MATS UNIVERSITY

NAAC GRADE A+

MATS CENTRE FOR DISTANCE & ONLINE EDUCATION

Database Technologies

Master of Computer Applications (MCA)

Semester - 1

SELF LEARNING MATERIAL

MATS UNIVERSITY

www.matsuniversity.ac.in

NAAC GRADE A+ ACCREDITED UNIVERSITY

Master of Computer Applications

ODL MCA DSC-102-T

Database Technologies

COURSE DEVELOPMENT EXPERT COMMITTEE

COURSE COORDINATOR

Prof. (Dr.) Omprakash Chandrakar, Professor and Head, School of Information Technology, MATS University, Raipur, Chhattisgarh

COURSE PREPARATION

Prof. (Dr.) Omprakash Chandrakar, Professor and Head, School of Information Technology, MATS University, Raipur, Chhattisgarh

March, 2025

ISBN: 978-93-49916-20-3

Printed & Published on behalf of MATS University, Village-Gullu, Aarang, Raipur by Mr. Meghanadhudu Katabathuni, Facilities & Operations, MATS University, Raipur (C.G.)

Printed at: The Digital Press, Krishna Complex, Raipur-492001(Chhattisgarh)

Acknowledgement

The material (pictures and passages) we have used is purely for educational purposes. Every effort has been made to trace the copyright holders of material reproduced in this book. Should any infringement have occurred, the publishers and editors apologize and will be pleased to make the necessary corrections in future editions of this book.

COURSE INTRODUCTION

Databases play a crucial role in managing, storing, and retrieving structured information efficiently. This course provides a comprehensive understanding of database management systems (DBMS), covering fundamental concepts, relational modeling, SQL, transaction management, and object-oriented databases.

Block 1: Introduction to Database Management System

This Block lays the foundation by introducing database management systems, their evolution, key characteristics, advantages, and real-world applications. It explores different types of DBMS and their significance in modern computing environments.

Block 2: Relational Data Modeling and Database Design

A well-structured database starts with a robust design. This Block covers relational data modeling, entity-relationship (ER) diagrams, normalization techniques, and schema design principles to ensure data consistency and integrity.

Block 3: SQL and Procedural SQL

Structured Query Language (SQL) is the backbone of database interaction. This Block introduces fundamental SQL commands and extends into procedural SQL, covering stored procedures, triggers, and functions to enhance database operations.

Block 4: Transaction Management and Concurrency

Data consistency and reliability are essential in multi-user environments. This Block discusses ACID properties, transaction processing, concurrency control techniques, and recovery mechanisms to ensure data integrity in database systems.

Block 5: Object-Oriented Database

The evolution of data storage has led to object-oriented databases (OODB), which integrate object-oriented principles with database management. This Block explores OODB concepts, advantages, and their application in complex data structures.

By the end of this course, learners will have a strong grasp of database concepts, design methodologies, and practical SQL skills to manage and optimize databases efficiently.

Block 1: Introduction to Database Management System

Unit 1: Purpose of Database Systems

Structure

1.1: Introduction

In the modern digital era, organizations generate and manage vast amounts of data daily. Managing this data efficiently, securely, and reliably is the central purpose of database systems. A Database Management System (DBMS) provides an organized framework for storing, retrieving, and manipulating data while ensuring accuracy, consistency, and accessibility. Traditional file-based systems often suffer from redundancy, inconsistency, and lack of data security. Database systems overcome these limitations by integrating mechanisms for data integrity, concurrency control, and recovery.

Database systems enable structured data storage using relational, object-based, or semi-structured models, allowing users and applications to perform complex queries with ease. Their design emphasizes data independence, where applications remain unaffected by physical storage changes. Today, DBMSs such as MySQL, PostgreSQL, and MongoDB form the backbone of applications in diverse domains like banking, healthcare, e-commerce, and education—ensuring data-driven decision-making, scalability, and reliability.

1.2: Learning Outcomes

After completing this unit, students will be able to:

1. Explain the fundamental purpose and importance of database systems in modern information management.

2. Differentiate between traditional file systems and database systems in terms of efficiency, integrity, and security.
3. Describe the core functions of a DBMS, including data organization, retrieval, backup, and recovery.
4. Understand key database features such as data independence, data abstraction, and elimination of redundancy.
5. Identify real-world applications of database systems across various domains and technologies.

1.3: Purpose of Database Systems

It is software used to manage the efficient storage, retrieval, and manipulation of data. Its A2F architecture guarantees data integrity, security, and accessibility for all users and applications.

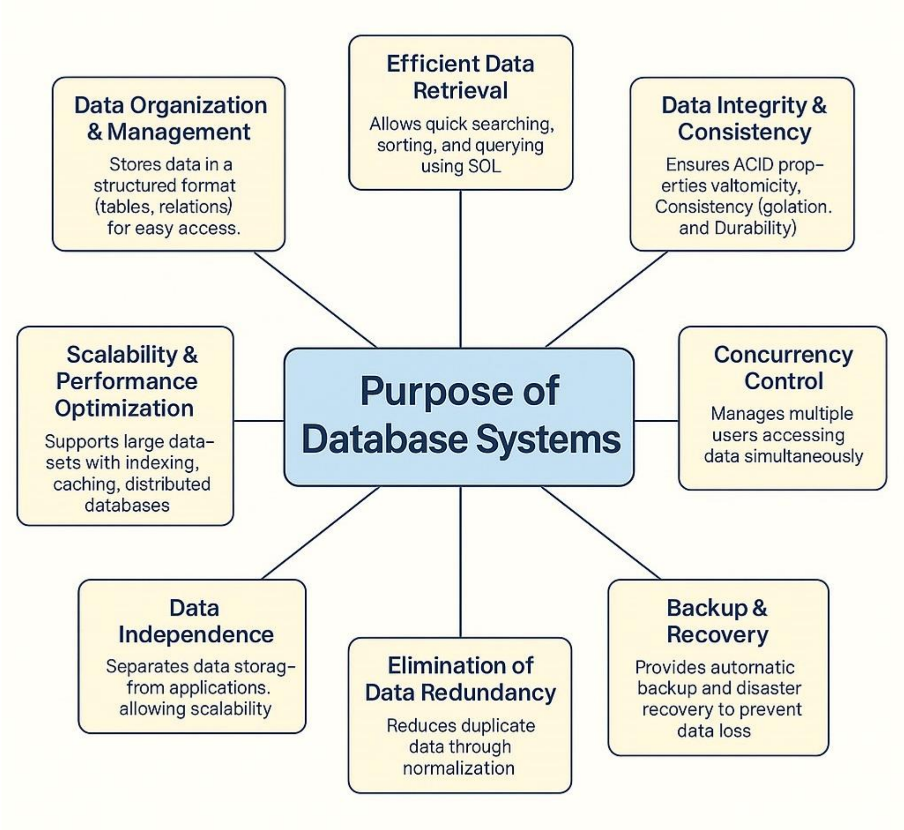

Figure 1.1 Purpose of Database Systems

Purpose of Database Systems

Purpose	Description
Data Organization & Management	Stores data in a structured format (tables, relations) for easy access.

Efficient Data Retrieval	Allows quick searching, sorting, and querying using SQL.
Data Integrity & Consistency	Ensures ACID properties (Atomicity, Consistency, Isolation, and Durability).
Data Security & Access Control	Restricts access using authentication & authorization (user roles, permissions).
Concurrency Control	Manages multiple users accessing data simultaneously.
Backup & Recovery	Provides automatic backup and disaster recovery to prevent data loss.
7. Data Independence	Separates data storage from applications, allowing scalability.
8. Elimination of Data Redundancy	Reduces duplicate data through normalization.
9. Scalability & Performance Optimization	Supports large datasets with indexing, caching, and distributed databases.

Example: Database Management System (DBMS)

A DBMS (e.g., MySQL, PostgreSQL, and MongoDB) helps in:

- Storing customer records in an e-commerce site.
- Managing bank transactions securely.
- Handling real-time analytics in businesses.

Modern applications such as banking, healthcare, e-commerce, and cloud computing demand efficient, secure, and scalable data management and that is where Database Systems comes into the picture**.**

1.4: View of Data: Data Abstraction, Instances and Schemas

A DBMS (database management system) is a software used for storing, retrieving and managing data. Databases utilize data abstraction, instances, and schemas to efficiently manage complex data and to organize and present data in the most effective way.

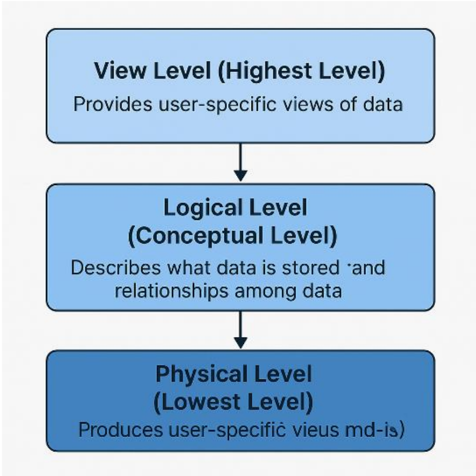

Figure 1.2 Levels of Data Abstraction

Data Abstraction

Data Abstraction means displaying only the relevant data while hiding the background details about how the data is stored and maintained. It contributes in handling large data bases effectively by dividing the data representation into three levels.

Table 1 Levels of Data Abstraction

Level	Description	Example
1. Physical Level (Lowest Level)	Describes how data is stored in memory (files, indexes, pointers).	Data stored as B-trees, Hash Tables, Blocks on Disk.
2. Logical Level (Conceptual Level)	Describes what data is stored and relationships among data.	Tables: Students (ID, Name, Course, Age)
3. View Level (Highest Level)	Provides user-specific views of the data.	A university student can see only his/her records, while an admin can access all student details.

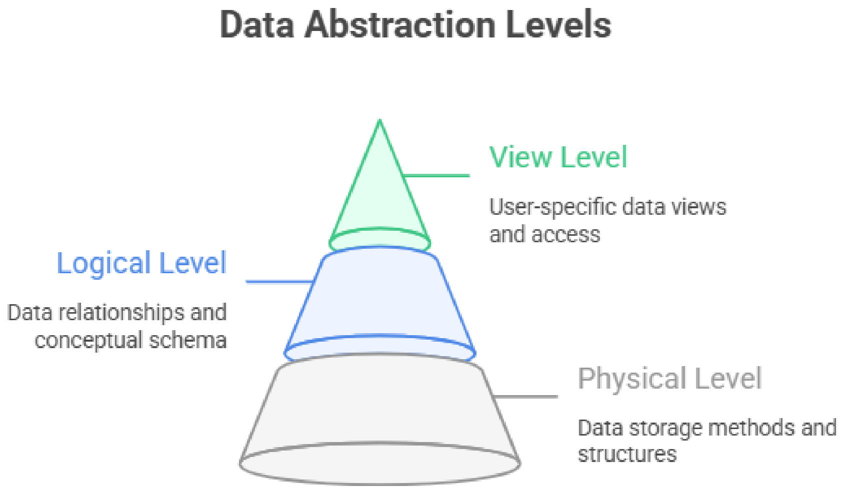

Figure 1.3 Levels of Data Abstraction

Example: In a banking system:

- Physical Level: Data is stored as indexed files on a disk.
- Logical Level: Tables store account details like Account_No, Name, Balance.
- View Level: A customer sees only their transactions, but the manager sees all accounts.

Instances and Schemas

Instance:

- Eg: The current state of the database at a specific point in time.
- Database keeps updating the instances as the data keeps changing.

Example:

A Students table contains:

- ID Name Age Course
- 101 Alex 21 CS
- 102 Emma 22 IT

• The Students table above is a snapshot of the Students table at this time.

• The instance changes when a new student joins.

Schema (Structure of the Database)

• Schema is the architecture of the database is stable.

• Describes tables, attributes, relationships, constraints.

For example: A schema for Students table:

CREATE TABLE Students (

Notes

ID INT PRIMARY KEY,
Name VARCHAR(50),
Age INT,
Course VARCHAR(50)
);

• All records share the same schema, even though we can add/delete records

Types of Schemas:

Schema Type	**Description**
Physical Schema	Defines storage details (indexes, partitioning).
Logical Schema	Defines tables, relationships, constraints.

Difference between Instance and Schema

Feature	**Instance**	**Schema**
Definition	Snapshot of data at a given moment	Blueprint or structure of the database
Changes	Frequently changes	Fixed unless modified by DBA
Example	Current rows in Students table	Table design (ID, Name, Age, Course)

- Data Abstraction facilitates easier management of database, separates data storage, structure and how user view.
- Instances contain the most current data, which changes continuously.
- Schemas dictate the architecture of a database, ownership and accessibility

Check Your Progress

1. Explain the major purposes of database systems and how they overcome the limitations of traditional file processing systems.

   ……………………………………………………………………
   ……………………………………………………………………
   ……………………………………………………………………
   ……………………………………………………………………

2. Describe the concept of data abstraction and discuss the roles of instances and schemas in database systems.
   ……………………………………………………………………
   ……………………………………………………………………
   ……………………………………………………………………

1.5: Summary

Database systems were developed to efficiently store, manage, and retrieve large volumes of data while ensuring consistency and integrity. Traditional file systems suffered from issues such as data redundancy, inconsistency, and difficulty in concurrent access. Database Management Systems (DBMS) resolve these issues through centralized management, reduced redundancy, and support for multiple users. The concept of data abstraction allows users to interact with data at different levels—physical, logical, and view—without worrying about storage details. A database schema defines the structure of data, while instances represent the data at a specific point in time. DBMSs also ensure data integrity, security, and concurrency control. They are used in various applications such as banking, healthcare, education, and e-commerce. Overall, database systems serve as the backbone of modern information management, providing reliability, scalability, and ease of access to data across diverse environments.

1.6: Exercises

Multiple Choice Questions:

1. **Which of the following best defines the main purpose of a Database Management System (DBMS)?**
   a) To design operating systems
   b) To store and manage data efficiently with integrity and security
   c) To create computer hardware
   d) To perform arithmetic calculations
   Answer: b) To store and manage data efficiently with integrity and security

2. **Which of the following ensures that multiple users can access data simultaneously without conflict?**
   a) Data Redundancy

Notes

b) Concurrency Control

c) Data Mining

d) Data Warehousing

Answer: b) Concurrency Control

3. **What does the term "Data Independence" refer to in database systems?**

   a) Freedom from using any data at all

   b) Separating data storage from application programs

   c) Backing up data automatically

   d) Removing duplicate data manually

   Answer: b) Separating data storage from application programs

4. **Which of the following is NOT an advantage of using a database system over traditional file systems?**

   a) Elimination of redundancy

   b) Increased data inconsistency

   c) Improved data security

   d) Enhanced backup and recovery

   Answer: b) Increased data inconsistency

5. **Which of the following is an example of a real-world application of database systems?**

   a) Text document creation in MS Word

   b) Online banking transaction management

   c) Drawing images using graphic software

   d) Playing music files from a local drive

   Answer: b) Online banking transaction management

Descriptive Questions

1. Explain the primary purpose of database systems and how they differ from traditional file-based systems.
2. Discuss the major functions of a Database Management System (DBMS) with suitable examples.
3. Define and elaborate on the concepts of data integrity, data security, and data independence in database systems.
4. Describe how database systems contribute to efficient and reliable data management in modern applications.

1.7: References and Suggested Readings

- Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson Education.
- Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database System Concepts* (7th ed.). McGraw Hill Education.
- Connolly, T., & Begg, C. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.
- Coronel, C., & Morris, S. (2019). *Database Systems: Design, Implementation, and Management* (13th ed.). Cengage Learning.

Block 1: Introduction to Database Management System

Unit 2: Data Models

Structure

2.1: Introduction

Data models form the conceptual foundation of every database system. They define how data is logically structured, stored, and related within a database. A data model provides a framework for organizing data elements and standardizing how different data items interact with one another.

The Relational Model, introduced by E. F. Codd, is the most widely used model in modern database systems, representing data in tables (relations) with rows and columns. Other models, such as the Entity–Relationship (ER) Model, help designers visually map entities, their attributes, and relationships before implementation. Additionally, Object-Based and Semi-Structured Data Models have evolved to handle complex, hierarchical, and schema-less data commonly used in multimedia applications, web services, and NoSQL systems.

Database languages like DDL (Data Definition Language), DML (Data Manipulation Language), DCL (Data Control Language), and TCL (Transaction Control Language) operate on these data models to define, manipulate, and control access to data.

In essence, understanding data models is essential for effective database design, data integrity, and efficient query processing in both traditional and modern database environments.

2.2: Learning Outcomes

After completing this unit, students will be able to:

1. Define and explain the concept and purpose of data models in database design.

2. Differentiate among various data models—Relational, Entity-Relationship, Object-Based, and Semi-Structured.
3. Illustrate the components of the ER Model, including entities, attributes, relationships, and cardinalities.
4. Explain different database languages (DDL, DML, DCL, TCL) and their functions.
5. Identify appropriate data models for different types of applications and datasets.

2.3: Data Models

Relational Model, Entity-Relationship Model, Object-Based Data Model, semi structured Data Model, Database Languages

Data models are abstract models that organize the elements of data and how they relate to one another and to the properties of real-world entities. Here's a snapshot of the data models and database languages you listed:

Data Models

1. **Relational Model:**
   - **Description:** In contrast, the relational model stores data in one or more tables (or 'relations') that consist of rows and columns, where each row is uniquely identified by a key. Rows are referred to as records or tuples, and columns as attributes or fields.
     - Tables, Rows (Tuples), Columns (Attributes)
     - Primary Keys and Foreign Keys
     - Relationships between tables
     - Normalization (removing redundancy and ensuring consistency)
   - **Pros:**
     - **Easy to use** – simple table structure.
     - **Reliable** – strong data integrity with keys and constraints.
     - **Flexible** – supports a wide variety of queries.
     - **Powerful** – can handle complex joins and operations.

Notes

- **Examples:** MySQL, PostgreSQL, Oracle.

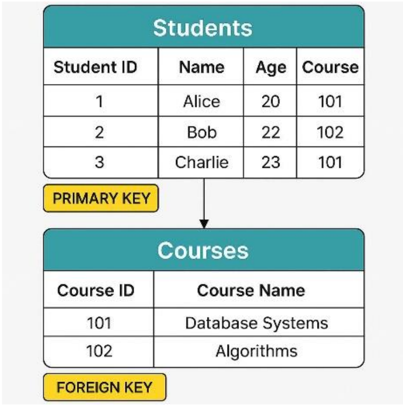

Figure 2.1 Relational Model Example

2. **Entity-Relationship Model (ER Model):**
   - **Definition:** The ER model is a high-level data model that provides a conceptual representation of the data structure of a database. This use ER diagrams for representation of your entities (tables), attributes (columns), and associations.
   - **Keywords:** Entities, attributes, relationships, cardinality and participation constraints.
   - **Entities**

     Meaning: Entities are the real-world objects, concepts, or things about which data is stored in a database. They can be tangible (like a student) or intangible (like a course).

     Example:

       Student: A student in a university database.

       Course: A course offered by the university (e.g., "Database Systems").

     Scenario: In a college database, both **Student** and **Course** are entities.
   - **Attributes**

     Meaning: Attributes describe the properties, characteristics, or fields of an entity. Each attribute stores a specific piece of data about the entity.

     Example (for Student entity):

       StudentID (unique identifier)

       StudentName (e.g., "Anita Sharma")

       Age (e.g., 21)

       City (e.g., Raipur)

Example (for Course entity):

- CourseID (e.g., C101)
- CourseName (e.g., "Database Systems")
- Credits (e.g., 4)

- **Relationships**

  Meaning: A relationship represents how two or more entities are connected to each other.

  Example:

  - **"Enrolled In"**: A student can enroll in many courses.
  - If Anita Sharma (Student) takes "Database Systems" (Course), then the relationship between **Student** and **Course** is "Enrolled In."

- **Cardinality**

  Meaning: Cardinality specifies the number of instances of one entity that can or must be associated with the number of instances of another entity.

  Types of cardinality with examples:

  One-to-One (1:1): One student has one ID card.

  One-to-Many (1:N): One course can have many students enrolled.

  Many-to-Many (M:N): A student can enroll in many courses, and each course can have many students.

- **Participation Constraints**

  Meaning: These define whether all or only some instances of an entity participate in a relationship.

  Types with examples:

  Total Participation: Every instance of the entity must participate in the relationship.

  Example: Every student must have a StudentID card → Student ↔ ID_Card.

  Partial Participation: Some instances of the entity may not participate in the relationship.

  Example: Not every student enrolls in a hostel → Student ↔ Hostel.

- **Strengths**: Simple to comprehend and picture, benefits database design.

  - **Easy to visualize** – Simple diagrams for complex systems.

- **Great for design** – Helps in planning before implementation.
- **Improves communication** – Everyone (even non-technical users) can understand the database structure.

- **Example(s)--** Usually you will use this during the design phase before creating a relational database, you might design an ER diagram:
  - **Entity:** `Student` with attributes (StudentID, Name, Age)
  - **Entity:** `Course` with attributes (CourseID, CourseName)
  - **Relationship:** `EnrolledIn` (Student ↔ Course) with cardinality (many students can enroll in many courses).

Entity-Relationship Diagram for a University Database

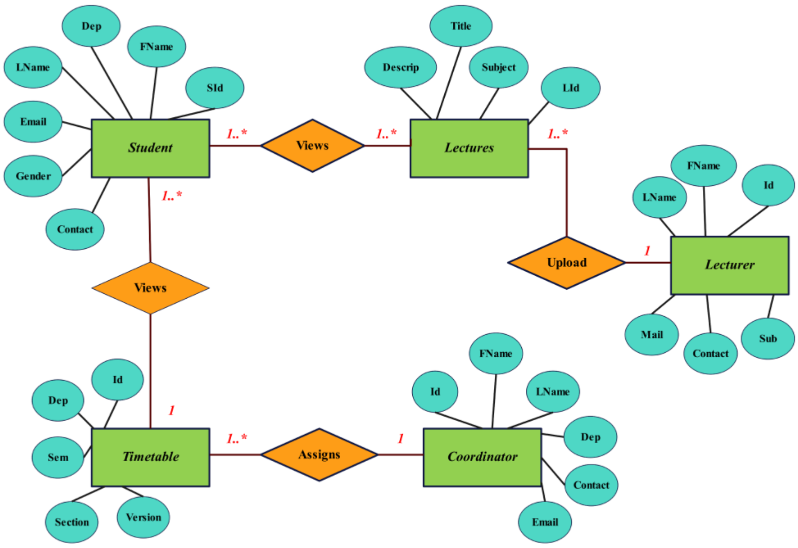

Figure 2.2 Relational Model Example

3. Object-Based Data Model:

- **Description:**

  The Object-Based Data Model is an extension of the relational model that incorporates object-oriented concepts. It supports complex data types, encapsulation, inheritance, and polymorphism—making it ideal for representing real-world objects in a database.

- **Key Concepts:**

- **Objects** – Real-world entities represented as objects with state and behaviours.
- **Classes** – Blueprints that define the structure and behaviours of objects.
- **Inheritance** – Classes can inherit attributes and methods from parent classes.
- **Encapsulation** – Data and methods are bundled together.
- **Polymorphism** – Same method or operation can behave differently based on the object.

- **Benefits:**
  - **Handles complex data schemas** easily (multimedia, spatial, or hierarchical data).
  - **Better compatibility** with object-oriented programming languages like Java, C++, or Python.
  - **Reusability and modularity** thanks to inheritance and encapsulation.
- **Examples:**
  - **PostgreSQL** (supports object-relational features like custom types and inheritance)
  - **Oracle Database** (supports object types and methods)

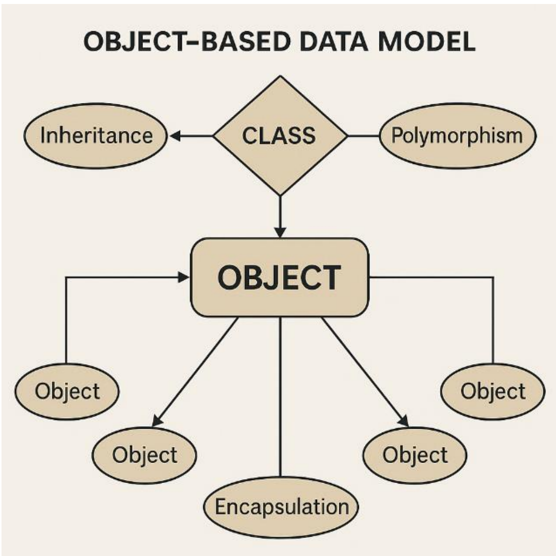

Figure 2.3 Object Based Data Model

4. **Semi-Structured Data Model:**
   - **Definition:**
     The semi-structured data model is used when data cannot be arranged into rigid tables like in relational databases. It has no fixed schema and supports nesting of data elements, making it highly flexible for representing irregular or evolving data structures.
   - **Keywords:**
     - **Tags** – Markers (like in XML or JSON) to identify elements.
     - **Elements** – Data items within tags or keys.
     - **Nesting** – Data structures can contain other data structures.
     - **Flexible** – Schema-less or self-describing data.
   - **Pros:**
     - **Flexible management** of heterogeneous (mixed) data.
     - **Easy integration** with web-based or semi-structured sources (APIs, documents).
     - **Ideal for rapidly changing requirements** (no need to redesign schemas).
   - **Examples:**
     - **MongoDB** (stores data as JSON-like documents)
     - **Couchbase**
     - **Any NoSQL database**
     - **XML** and **JSON** files commonly used in web services.

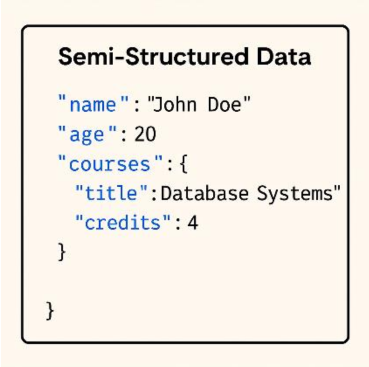

Figure 2.4 Semi-Structured Data

2.4: Database Languages

1. **Data Definition Language (DDL):**
   - **Description:**
     DDL (Data Definition Language) is used to define, modify, or remove the structure of database objects. It deals with the schema and structural changes in a database rather than the data itself. You can use DDL commands to create new tables or other objects, alter their definitions, or remove them entirely.
   - **Main operations:**

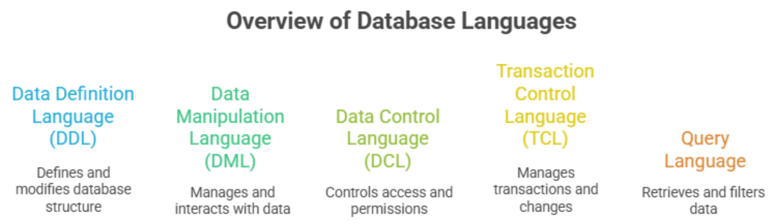

Figure 2.5 Types of Database Languages

     - **Create** database objects (tables, indexes, schemas, views, etc.)
     - **Alter** existing database objects (add/remove columns, change data types)
     - **Drop** or delete objects
     - **Truncate** tables (remove all rows quickly while keeping structure)
   - **Examples of DDL commands:**
     - **CREATE** – Creates new objects (e.g., CREATE TABLE students (...);)
     - **ALTER** – Modifies an existing object's structure (e.g., ALTER TABLE students ADD age INT;)
     - **DROP** – Deletes an object completely (e.g., DROP TABLE students;)
     - **TRUNCATE** – Removes all data from a table but keeps its structure (e.g., TRUNCATE TABLE students;)
2. **Data Manipulation Language (DML):**
   - **Description:**
     DML (Data Manipulation Language) is used to manage and interact with the data stored in database objects (like tables). It allows you to insert new data, update existing data, delete

unwanted data, and retrieve data as needed. Unlike DDL, which affects the structure, DML works on the actual records inside the structure.

- **Main operations:**
  - **Insert** new records into a table
  - **Update** existing records in a table
  - **Delete** records from a table
  - **Select** (retrieve) records from one or more tables

- **Examples of DML commands:**
  - **SELECT** – Retrieves data (e.g., SELECT * FROM employees;)
  - **INSERT** – Adds new rows (e.g., INSERT INTO employees (id, name) VALUES (1, 'John');)
  - **UPDATE** – Modifies existing rows (e.g., UPDATE employees SET name = 'John Doe' WHERE id = 1;)
  - **DELETE** – Removes rows (e.g., DELETE FROM employees WHERE id = 1;)

3. **Data Control Language (DCL):**
   - **Definition**: DCL is a language used to control accessibility of the data in the database. It has commands to add and remove permissions
   - **Definition:**
     DCL (Data Control Language) is used to control access to data stored in the database. It allows database administrators to grant or revoke permissions on database objects, ensuring only authorized users can perform certain actions.
   - **Main operations:**
     - **Grant** permissions to users or roles
     - **Revoke** permissions from users or roles
   - **Examples of DCL commands:**
     - **GRANT** – Gives specific privileges to a user or role *(e.g., GRANT SELECT, INSERT ON employees TO user1;)*
     - **REVOKE** – Removes previously granted privileges *(e.g., REVOKE INSERT ON employees FROM user1;)*
4. **Transaction Control Language (TCL):**

- **Description:**
  TCL (Transaction Control Language) is used to manage transactions in a database. It works closely with DML operations to control how changes are saved or undone, ensuring data integrity and consistency. TCL commands help you confirm, cancel, or temporarily mark points within a transaction.
- **Main operations:**
  - **Commit** a transaction (save changes permanently)
  - **Rollback** a transaction (undo changes since the last commit)
  - **Savepoint** within a transaction (set a point to which you can roll back later)
- **Examples of TCL commands:**
  - **COMMIT** – Makes all changes in the current transaction permanent
    (e.g., COMMIT;)
  - **ROLLBACK** – Undoes changes made in the current transaction
    (e.g., ROLLBACK;)
  - **SAVEPOINT** – Creates a named savepoint to roll back to if needed
  -
    (e.g., SAVEPOINT sp1; then later ROLLBACK TO sp1;)

5. **Query Language:**

- A query language is a data access language used to make queries in databases and information systems.
  It enables users to retrieve, filter, and organize data according to specific conditions. SQL (Structured Query Language) is the most widely used query language in relational database systems.
- **Examples of SQL Clauses (used in queries):**
  - SELECT – Specify the columns to retrieve
  - FROM – Specify the table(s) to query
  - WHERE – Apply conditions to filter records
  - GROUP BY – Group rows sharing a property
  - HAVING – Filter groups based on conditions
  - ORDER BY – Sort the result set

Notes

- **Data Models supported by Query Languages:**
  - **Relational Model:** Data is organized in **tables** with rows and columns, and relationships between them.
  - **Object-Based Data Model:** Incorporates **object-oriented features** (objects, classes, inheritance).
  - **Semi-Structured Data Model:** Supports flexible, schema-less representation, such as JSON or XML.

- **Database Languages include:**
  - **Data Definition Language (DDL)** – Define and modify structure
  - **Data Manipulation Language (DML)** – Manage data in tables
  - **Data Control Language (DCL)** – Control access and permissions
  - **Transaction Control Language (TCL)** – Manage transactions and changes

Check Your Progress

1. Define a data model and explain its significance in database design and management.

   ……………………………………………………………
   ……………………………………………………………
   ……………………………………………………………
2. Compare and contrast hierarchical, network, and relational data models with suitable examples.

   ……………………………………………………………
   ……………………………………………………………
   ……………………………………………………………

2.5: Summary

Data models provide a systematic way to represent, organize, and manipulate data in a database system. They define how data is structured, related, and constrained. The hierarchical model arranges data in a tree-like structure, whereas the network model allows more flexible many-to-many relationships. The relational model, proposed by E. F. Codd, organizes data into tables with rows and columns and uses keys to establish relationships. Advanced models, such as object-based and semi-structured data models, handle complex and unstructured data effectively. The Entity–Relationship (ER) model helps in conceptual design by representing real-world entities and their relationships graphically. Database languages like DDL, DML, DCL, and TCL support interaction with the data model. Understanding various data models is crucial for choosing the right representation and improving database efficiency, integrity, and maintainability.

2.6: Exercises

Multiple Choice Questions:

1. **Which of the following data models organizes data into tables with rows and columns?**
   a) Hierarchical Model
   b) Network Model
   c) Relational Model
   d) Object-Based Model
   Answer: c) Relational Model

2. **In an Entity–Relationship (ER) Model, what does a 'relationship' represent?**
   a) A type of attribute
   b) A connection between two or more entities
   c) A physical file on disk
   d) A constraint in the database
   Answer: b) A connection between two or more entities

3. **Which of the following is NOT a database language?**
   a) DML
   b) DCL
   c) HTML
   d) TCL
   Answer: c) HTML

4. **Which of the following features is unique to Object-Based Data Models?**
   a) Tables and rows
   b) Inheritance and encapsulation
   c) Foreign key constraints

d) Use of normalization

Answer: b) Inheritance and encapsulation

5. **Which data model is best suited for handling flexible, irregular, or nested data structures like JSON or XML?**
a) Relational Model
b) Object-Based Model
c) Semi-Structured Model
d) Hierarchical Model
Answer: c) Semi-Structured Model

Descriptive Questions

1. Define a data model. Explain its importance in the design and implementation of database systems.
2. Compare and contrast the Relational, Entity–Relationship, Object-Based, and Semi-Structured data models with suitable examples.
3. Describe the key components of the Entity–Relationship Model and explain their roles in database design.
4. Discuss the different types of database languages and their significance in managing and controlling data.

2.7: References and Suggested Readings

- Date, C. J. (2004). *An Introduction to Database Systems* (8th ed.). Pearson Education.
- Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson Education.
- Coronel, C., & Morris, S. (2019). *Database Systems: Design, Implementation, and Management* (13th ed.). Cengage Learning.
- Kroenke, D. M., & Auer, D. J. (2012). *Database Concepts* (6th ed.). Pearson Education.

Block 1: Introduction to Database Management System

Unit 3: Database Architecture, Storage, and Administration

Structure

3.1: Introduction

Database architecture defines the structure, components, and interactions among users, applications, and the database management system (DBMS). It acts as a blueprint for how data is stored, accessed, and managed efficiently in a computing environment. Modern database architectures—ranging from single-tier to multi-tier—enable scalability, performance optimization, and secure data access in both centralized and distributed systems.

Data storage is another critical component that determines how information is physically stored and retrieved. Databases use various storage mechanisms such as primary (main memory), secondary (disk storage), and tertiary (backup storage) to ensure quick access, durability, and reliability. Techniques like indexing, clustering, and query optimization enhance retrieval speed and minimize storage overhead.

Database administration involves managing database systems to ensure their performance, availability, and security. Database Administrators (DBAs) handle installation, configuration, backup and recovery, user access control, and performance tuning. Together, these concepts form the backbone of an efficient data management environment—enabling organizations to process and analyze large volumes of data seamlessly and securely.

Notes

3.2: Learning Outcomes

After completing this unit, students will be able to:

1. Explain the basic architecture and components of database systems.
2. Describe different types of database architectures (1-tier, 2-tier, and 3-tier).
3. Understand how data is stored, indexed, and queried efficiently.
4. Identify the key responsibilities of database administrators and other database users.
5. Explain the importance of database backup, recovery, and performance tuning in maintaining data integrity and reliability.

3.3: Data Storage and Querying, Database Architecture

1. Data Storage and Querying

Data Storage

In order to access and obtain data quickly, data is stored efficiently on various storage structures in a database. The two most commons types of storage are:

Storage Type	**Description**	**Examples**
Primary Storage (Main Memory)	Stores frequently accessed data in RAM for quick access.	Cache memory, Buffer pool
Secondary Storage (Disk Storage)	Stores large amounts of data persistently.	Hard Disk (HDD), SSD
Tertiary Storage	Used for long-term backups and archival data.	Magnetic tapes, Cloud storage

How Databases Store Data?

- Heap Storage – Stores unordered records (slow for searches).
- Indexed Storage – Uses B-Trees, Hash Indexes for fast lookup.
- Clustered Storage – Groups related data together for efficiency.

Querying in Databases

A query is a request to retrieve, insert, update, or delete data. Queries are written using SQL (Structured Query Language).

Example SQL Queries:

```
-- Retrieve all students older than 20
SELECT * FROM Students WHERE Age > 20;
-- Insert a new student record
```

INSERT INTO Students (ID, Name, Age, Course) VALUES (103, 'John', 21, 'CS');
-- Update a student's course
UPDATE Students SET Course = 'AI' WHERE ID = 103;
-- Delete a student record
DELETE FROM Students WHERE ID = 103;

Query Optimization:

- Query optimization is the process of improving the performance of database queries so they run faster and use fewer resources.
- **Key Techniques in Query Optimization:**
  - **Use of Indexes:**
  - Speeds up data retrieval by avoiding full table scans. *(e.g., creating B-Tree or Hash indexes on frequently searched columns)*
  - **Query Rewriting:**
    Rewriting a query into an equivalent but more efficient form.
    (e.g., replacing subqueries with joins, simplifying conditions)
  - **Execution Plans:**
    The database's query optimizer evaluates multiple plans and chooses the most efficient one.

2. Database Architecture

Databases are designed based on different architectures, which define how users, applications, and database systems interact.

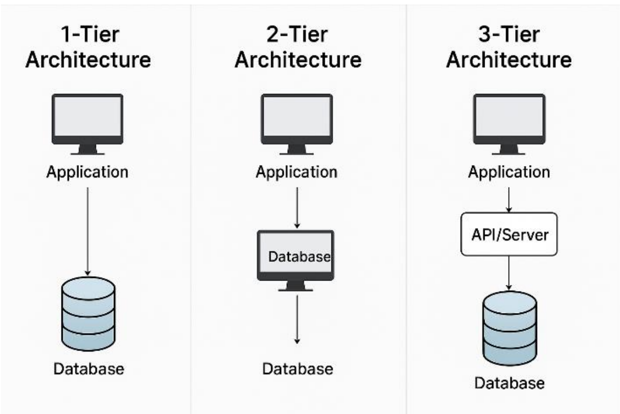

Figure 3.1 Tier Architecture

Architecture Type	Description	Examples
1-Tier Architecture	The database is directly accessed by the application.	Local file databases (MS Access)
2-Tier Architecture	Application connects to a central database (client-server model).	MySQL, PostgreSQL
3-Tier Architecture	Uses an intermediate layer (API/Server) between user and database.	Web applications (MySQL + Django/Node.js)

3.4: Components of Database Architecture

Component	Function
Database	Stores data in structured format.
DBMS (Database Management System)	Manages data, queries, and transactions.
Query Processor	Converts SQL queries into execution plans.
Storage Manager	Handles data retrieval, indexing, and optimization.
Transaction Manager	Ensures ACID properties (Atomicity, Consistency, Isolation, Durability).

Example: 3-Tier Web Application Architecture

1. Presentation Layer – Web UI (HTML, React)
2. Application Layer – Backend (Python, Java, Node.js)
3. Database Layer – DBMS (MySQL, MongoDB)

- **Data Storage**:
  - Databases are designed with indexing**,** sharding**,** replication, and caching strategies to handle large-scale data efficiently.
  - Proper storage design ensures:
  - **High performance** (faster queries)

Notes

- **Scalability** (handles growth in users/data)
- **Security** (controlled access, encrypted storage)

- For querying, we use SQL (statement for retrieving the data in SQL)
- Database Architecture that is highly secure, scalable, and efficient.

Designing a high performance application require a proper understanding of these concepts

3.5: Database Users and Administrators

In line with the database system, there are two kinds of roles, Users, and Administrators. On that note, here are five types of database users and administrators along with their responsibilities:

1. Database Administrators (DBAs)

• Responsibilities: Database Administrators (DBAs) oversee the database system's administration, upkeep, and performance.

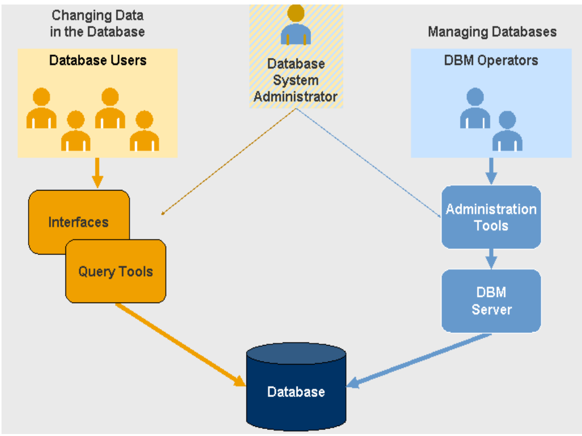

Figure 3.2 Database Administrator
(Source: https://maxdb.sap.com)

Responsibilities of a Database Administrator (DBA)

1. **Installation and Upgrades of Database Software**
   - The DBA is responsible for installing database software (such as Oracle, MySQL, SQL Server, or PostgreSQL) on servers.
   - They also ensure that patches, service packs, and major upgrades are applied in a timely manner to keep the database environment secure and up to date.
   - During upgrades, the DBA must carefully plan and test the migration to avoid downtime and data loss.

2. **Database Configuration and Optimization (Tuning)**
    - Once installed, the DBA configures the database to suit organizational requirements.
    - Configuration includes setting up parameters, memory allocation, storage options, and connection limits.
    - Optimization or tuning involves adjusting query execution plans, indexing, and caching strategies to improve database performance and ensure efficient utilization of resources.
3. **User Access and Security Management**
    - DBAs control who can access the database and what actions they can perform.
    - They manage user accounts, roles, and privileges, granting or revoking permissions as necessary.
    - Security measures may also include encrypting sensitive data, enforcing password policies, and monitoring suspicious activities to prevent unauthorized access.
4. **Backup and Recovery of Data**
    - One of the most critical DBA responsibilities is ensuring that data is backed up regularly.
    - DBAs design backup strategies such as full, incremental, and differential backups.
    - They also test recovery procedures to ensure that data can be restored quickly in case of system failure, corruption, or disaster.
    - For example, in Oracle, RMAN (Recovery Manager) may be used for backups, while SQL Server provides native backup utilities.
5. **Performance Monitoring and Issue Resolution**
    - DBAs continuously monitor the database for performance issues such as slow queries, locking, blocking, or resource bottlenecks.
    - They use monitoring tools to identify problems early and take corrective actions, such as adding indexes, tuning queries, or reallocating resources.
    - Regular health checks help in ensuring smooth database operations.

6. **Providing Appropriate Access Controls**
    - The DBA ensures that users only have access to the data and functions necessary for their role (principle of least privilege).
    - This prevents accidental or intentional misuse of data and enhances compliance with organizational and regulatory requirements (e.g., GDPR, HIPAA).
    - Access control may be implemented through roles, schemas, and permissions management.

Example: Tools Used by a Database Administrator (DBA)

A DBA not only relies on skills and knowledge but also on specialized tools to manage databases effectively. Two commonly used tools are **Oracle Enterprise Manager (OEM)** and **SQL Server Management Studio (SSMS)**.

1. **Oracle Enterprise Manager (OEM)**
    - OEM is a **centralized management console** provided by Oracle.
    - It allows DBAs to:
        - Monitor **database health** (CPU usage, memory utilization, query performance, sessions, and wait events).
        - Configure and schedule **automated backups**.
        - Apply **patches and upgrades** across multiple Oracle databases from a single dashboard.
        - Manage **user security**, including creating users, granting roles, and auditing activities.
        - Set **alerts and notifications** to proactively respond to performance or security issues.
    - Example: A DBA can use OEM to identify a slow-running query, check its execution plan, and tune it by adding an index.
2. **SQL Server Management Studio (SSMS)**
    - SSMS is a **graphical interface** provided by Microsoft for SQL Server databases.
    - It provides features such as:
        - Writing, testing, and executing **T-SQL queries**.

- Configuring **database security** (creating logins, assigning roles, and controlling access).
- Designing and modifying **schemas, tables, and indexes**.
- Monitoring **real-time performance** through Activity Monitor.
- Scheduling **jobs and maintenance plans** (backups, index rebuilds, and consistency checks).

  - Example: A DBA can use SSMS Activity Monitor to see which queries are consuming the most CPU time and then optimize them.

2. Database Designers

• Role: Also referred to as the database architect, the database designer is responsible for designing the database structure and schema.

• **Responsibilities**:

- Identifying user needs and converting them into a database schema.
- Develop ER diagrams and their corresponding relational schemas.
- Normalizing the database to avoid redundancy and make it more efficient.
- Establishing tables, links, conditions and indexes.
- **Example**: ER Diagrams and DB Schema Sophia (damn every time I use Sophia feels so real to me) is a database designer, she can use ERwin or Lucidchart to create ER Diagram and Design DB Schema

3. End Users

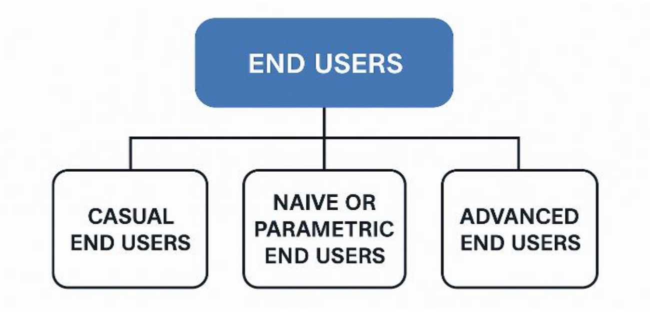

Figure 3.3 End User

Role: The End users are system users who enter into the database using different applications to retrieve, insert, update, or delete data.•

Types of End Users:

- Casual End Users: Use query languages (for example, SQL) to access the database on an occasional basis.
- Naive or Parametric End Users: Use existing applications or forms to access the database (e.g. ATMs, online shopping carts).
- Advanced End Users: Use specialized tools such as data analysis software or craft sophisticated queries.

• Responsibilities:

- Platforms that utilize the database for their work (e.g., interrogating data,c producing reports).
- Ensuring that data the data inputted into the system is accurate and complete.

• BOT: AN EXAMPLE We have a sales manager querying the database to generate a sales report.

4. **Application** • Role: Application programmers design and

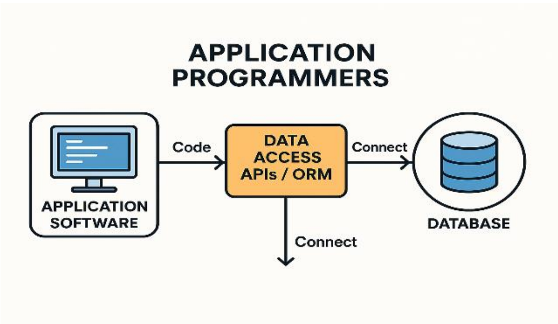

Figure 3.4 Application Programmer

create software applications that will communicate with the database.

• Responsibilities:

- Coding into applications to enable the database.
- Connect to the database using Data Access APIs (e.g., JDBC, ODBC, etcSimple API) or ORM (Object-Relational Mapping) tools
- Application logic ensuring data consistency and security.
- {Debugging and optimizing database queries in application.

• For instance, a programmer could use SQLAlchemy within a Python script to pull data from a PostgreSQL database

5. **System Analysts**

Notes

Role: Systems analysts are the link between end users and the database system. They understand what the user needs and optimize the database accordingly.

• **Responsibilities:**

- Collecting and Evaluating User Requirements
- Collaborating with database designers to confirm that the

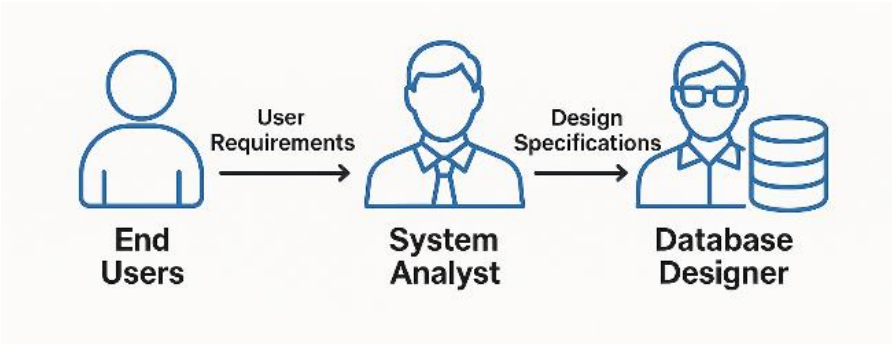

Figure 3.5 System Analysts

design aligns with user applications.

- Audit of the DB system to validate the functional & performance requirements.
- Preparing system specifications and user manuals.

• example: A system analyst helps a healthcare provider design a database for patient records.

Role	Primary Responsibility
Database Administrator	Manages and maintains the database system (e.g., performance, security, backups).
Database Designer	Designs the database schema and structure (e.g., ER diagrams, normalization).
End Users	Interact with the database to perform tasks (e.g., querying, updating data).
Application Programmers	Develop applications that interact with the database (e.g., APIs, ORM tools).
System Analysts	Analyze user requirements and ensure the database meets those needs.

Check Your Progress

1. Explain the different types of database architectures and discuss their advantages.

…………………………………………………………………

…………………………………………………………………

…………………………………………………………………

2. Describe the roles and responsibilities of a Database Administrator (DBA) in managing storage, security, and recovery.

   …………………………………………………………………

   …………………………………………………………………

   …………………………………………………………………

3.5: Summary

Database architecture defines how data is stored, accessed, and managed across a system. Common architectures include one-tier (standalone), two-tier (client-server), and three-tier (web-based) models, each offering different levels of abstraction and scalability. A DBMS consists of several components including the query processor, storage manager, and transaction manager. Efficient data storage relies on proper organization using indexing, file systems, and access methods. The Database Administrator (DBA) plays a critical role in maintaining the database environment—ensuring backup, recovery, performance tuning, and user security. Data is stored across primary, secondary, and tertiary storage media, each offering trade-offs in speed and cost. As databases grow in complexity, administrators use advanced tools to automate tasks and enforce policies. A well-designed architecture ensures reliability, data integrity, and optimal performance across enterprise systems

3.6: Exercises

Multiple-Choice Questions:

1. **Which component of a database architecture is responsible for converting SQL queries into executable plans?**

   a) Storage Manager

   b) Query Processor

   c) Transaction Manager

   d) Database Administrator

   Answer: b) Query Processor

2. **In a three-tier architecture, which layer handles business logic and connects the user interface to the database?**
   a) Presentation Layer
   b) Application Layer
   c) Database Layer
   d) Storage Layer
   Answer: b) Application Layer

3. **Which of the following is NOT a responsibility of a Database Administrator (DBA)?**
   a) Backup and recovery management
   b) Query optimization
   c) Application coding and interface design
   d) Access control and security management
   Answer: c) Application coding and interface design

4. **Which type of storage is typically used for long-term data backup and archiving?**
   a) Primary storage
   b) Secondary storage
   c) Tertiary storage
   d) Cache memory
   Answer: c) Tertiary storage

5. **Which of the following ensures that transactions in a database maintain Atomicity, Consistency, Isolation, and Durability (ACID)?**
   a) Query Processor
   b) Transaction Manager
   c) Data Dictionary
   d) Storage Manager
   Answer: b) Transaction Manager

Descriptive Questions

1. Explain the different types of database architectures and their real-world applications.
2. Describe how data is stored and retrieved in a database system. Discuss the role of indexing and query optimization.

3. Discuss the key functions and responsibilities of a Database Administrator (DBA).
4. Explain the importance of database backup, recovery, and performance monitoring in ensuring data integrity and system reliability.

3.7: References and Suggested Readings

1. Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database System Concepts* (7th ed.). McGraw Hill Education.
2. Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (3rd ed.). McGraw Hill.
3. Mullins, C. S. (2012). *Database Administration: The Complete Guide to DBA Practices and Procedures* (2nd ed.). Addison-Wesley Professional.
4. Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson Education

Block 2: Relational Data Modeling and Database Design

Unit 4: Relational Model and Constraints

Structure

4.1: Introduction

The Relational Model, introduced by Dr. E. F. Codd, revolutionized data management by organizing information into tables (relations) consisting of rows (tuples) and columns (attributes). Each table represents a specific entity, and relationships among entities are established using keys. This tabular structure ensures data consistency, easy manipulation, and logical organization.

The relational model enforces integrity constraints to maintain data accuracy and reliability. These include domain constraints, key constraints, entity integrity, and referential integrity, which collectively ensure that the data stored remains valid and consistent across related tables. Understanding these principles helps in designing structured databases that eliminate redundancy and maintain data integrity, forming the foundation of all modern relational database systems such as MySQL, PostgreSQL, and Oracle.

4.2: Learning Outcomes

After completing this unit, students will be able to:

1. Explain the basic concepts of the relational model, including relations, attributes, and tuples.
2. Identify and describe different types of keys—super key, candidate key, primary key, and foreign key.
3. Understand and apply various integrity constraints in relational databases.
4. Differentiate between entity integrity and referential integrity constraints.

5. Design relational schemas ensuring data accuracy, minimal redundancy, and consistency.

4.3: Relational Model Concepts

Super Key, Candidate Key and In the Relational Model, data is organized in tables (relations) consisting of tuples (rows) and attributes (columns).

Each table represents an entity, and relationships among entities are established through keys.

Primary Key

A Primary Key is an attribute (or a combination of attributes) that uniquely identifies each tuple **(row)** in a table.

- No two rows can have the same primary key value.
- A primary key cannot contain NULL values.

Role in the Relational Model:

- **Ensures data integrity:** Each row is uniquely identifiable.
- **Reduces redundancy:** Prevents duplicate records.
- **Enables relationships:** Used as a reference by foreign keys in other tables.
- **Supports efficient storage & retrieval:** SQL queries rely on keys for quick lookups and joins.

Example:

StudentID (PK)	**Name**	**Course**
101	Sophia	DBMS
102	Alex	Networks

Here, **StudentID** is the **Primary Key**, ensuring each student record is unique.

Super Key

A Super Key is any set of one or more attributes (columns) that can uniquely identify a tuple (row) in a relation (table).

- It may consist of a single attribute or a combination of attributes.
- Every table must have at least one super key.

- A primary key is always a super key, but not every super key is a primary key (some may contain extra attributes that are not necessary for uniqueness).

Key points about Super Keys:

- **Uniqueness**: No two rows can have the same values for a super key.
- **Minimality not required**: Unlike a candidate key or primary key, a super key can have redundant attributes and still be unique.

Example:

Consider a Student table with attributes:

{StudentID, Email, Name, Phone}

Possible Super Keys:

{StudentID} (uniquely identifies each student)

{Email} (assuming each email is unique)

{StudentID, Name} (still unique, but not minimal)

{StudentID, Phone} (also unique, but includes extra attribute)

Candidate Key

A Candidate Key is a minimal set of one or more attributes that uniquely identifies each tuple (row) in a table.

- **Minimality:** A candidate key has no redundant attributes—if you remove any attribute from it, it will no longer uniquely identify rows.
- **Uniqueness:** No two rows can share the same values for a candidate key.
- **Multiple Candidate Keys:** A table can have more than one candidate key.

Among all candidate keys, one is chosen as the Primary Key, while others remain as alternate keys.

Example:

Consider a Student table with attributes:

{StudentID, Email, Name, Phone}

Possible Candidate Keys (assuming uniqueness):

{StudentID} (unique and minimal)

{Email} (unique and minimal)

Non-Candidate Example:

{StudentID, Email} (still unique but not minimal, because StudentID alone is enough)

Foreign Key

A Foreign Key is an attribute (or a set of attributes) in one table that refers to the Primary Key (or a unique key) in another table.

It is used to establish and enforce a link between the data in the two tables.

Key Characteristics:

- **Maintains referential integrity:** Every value in the foreign key column must either match a value in the referenced primary key column or be NULL.
- **Defines relationships between tables:** One-to-many or many-to-one relationships are often implemented using foreign keys.
- **Restricts invalid data:** The database will prevent inserting or updating a foreign key value that doesn't exist in the referenced table.

Example:

Suppose you have two tables:

Students Table

StudentID (PK)	Name
101	Sophia
102	Alex

Enrollments Table

EnrollmentID (PK)	StudentID (FK)	Course
1	101	DBMS
2	102	Networks

Here, StudentID in Enrollments is a Foreign Key that references StudentID in Students.

It ensures every enrollment record belongs to a valid student.

Notes

4.4: Constraints: Domain, Key, Entity & Referential Integrity

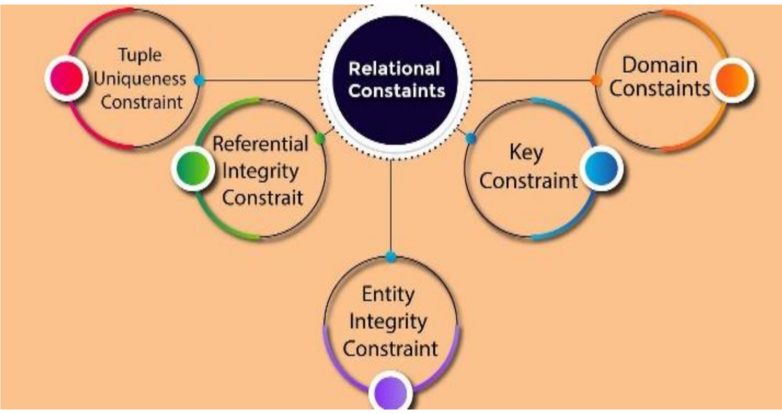

Figure 4.1 Relational Model and Constraints
(Source: https://medium.com)

Types of Integrity Constraints in RDBMS: Entity, Referential, Domain, and Key Integrity Constraints This include basically rules that are used by tables in relational databases in order to verify if the data inserted to these tables have a sense, when they are updated, deleted, etc. They limit the values that can be placed within tables and disallow bad values being entered. The main types of Constraints in a relational model are Domain Constraints Key Constraints Entity Integrity Constraints Referential Integrity Constraints

1. Domain Constraints

Domain Constraints restrict the values that a column (attribute) can take in a table. It validates that it stores only valid data types and values in the database.

Example: Assuming you have "Student" table with the following schema (Student_ID, Name, Age, and Email).

- Age is an example of a domain constraint, suppose Age is integer and Age is allowed only from 18 to 60 that is Age =17 or Age = 65 would violate the domain constraint.
- As another example, if Email is defined as a string that matches the format "@domain. i.e., student_email.@school. com" would be rejected

2. Key Constraints

Table: Key Constraint Ensures Each Row is Unique. This implies that two rows cannot possess the same value in a key attribute. Super Keys Candidate Keys Primary Keys

Example: Consider an "Employee" table having attributes (Employee_ID, Name, Email) and it is having Employee_ID as the Primary Key.

- If we attempt to add duplicate Employee_ID (like two employees with Employee_ID = 101), the Therewould be no insertion due to key constraint.

3. Entity Integrity Constraint

The Entity Integrity constraint ensures that the primary key of a table will never have NULL values. This Rule keeps each row in the table unique and into its own entity.

Example: In a “Product” table with attributes (Product_ID, Name, Price), the Product_ID is the Primary Key.

- The new product cannot be inserted with NULL as Product_ID, the entry will be rejected by database system, as primary key can be NULL.

4. Referential Integrity Constraint

A Referential Integrity Constraint is a set of rules that ensures that relationships between tables remain consistent. Foreign keys in one table must reference an existing primary key in another table or be NULL.

• **Definition:** A foreign key is a column whose data must match a primary key in another table **Example:** Two Tables, Orders, Customers.

Primary-foreign key relationship constraints (also known as referential integrity constraint) maintain the consistency between the two tables. It either has to refer to an existing primary key or NULL.

For example, imagine two tables, "Orders" and "Customers".

- **"Customers" Table:**

Customer_ID	Name	Email
C101	Alice	alice@email.com
C102	Bob	bob@email.com

- **"Orders" Table:**

Order_ID	Customer_ID	Product
O201	C101	Laptop
O202	C102	Phone
O203	C105	Tablet

- The *Customer_ID* column in the "Orders" table is a Foreign Key referencing the *Customer_ID* in the "Customers" table. If an order is placed with *Customer_ID = C105*, but no such customer exists in the "Customers" table, the database system will prevent the insertion to maintain referential integrity.

Constraints ensure data consistency and reliability within a relational database.

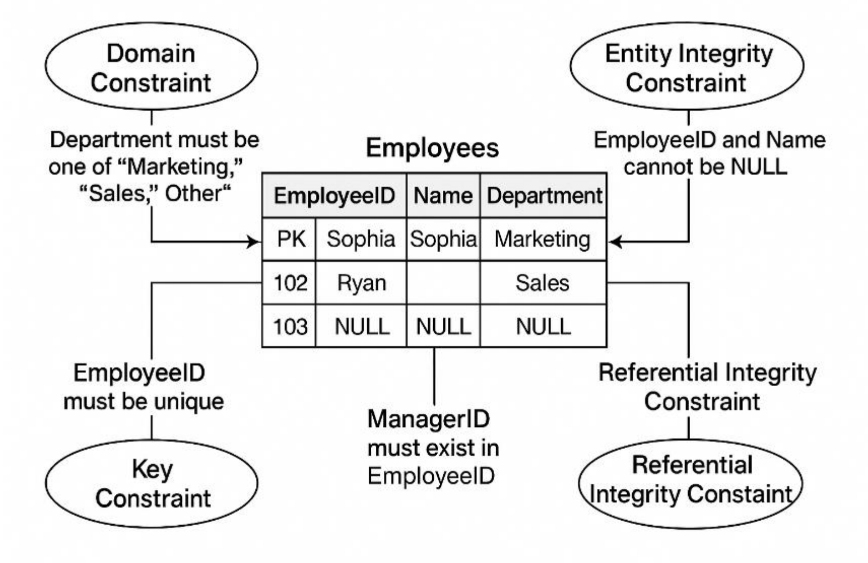

Figure.4.2: Constraints in Database

Domain Constraints – Maintain Proper Data Types & Values

- **Explanation:**
  Domain constraints ensure that the values entered into a column (attribute) of a table are valid, consistent, and within the acceptable range. Each attribute in a table is defined with a specific data type (e.g., INT, VARCHAR, DATE), and only values matching that type can be stored. Additionally, constraints like CHECK can restrict values further (e.g., age should be greater than 18).
- **Example:**
  If a column Age is defined as INT CHECK (Age >= 18), then only integer values greater than or equal to 18 are allowed. Inserting a string value like 'Twenty' or a number less than 18 would violate the domain constraint.

2. Key Constraints – Uniquely Identify Records

- **Explanation:**
  Key constraints are rules applied to attributes (or a set of

attributes) that help uniquely identify records in a table. Without key constraints, duplicate or ambiguous records could exist, making it difficult to distinguish one record from another. Common key constraints include **Primary Key, Unique Key, and Candidate Key**.

- **Example:**
  In a Student table, the column RollNo can be a key constraint. No two students should have the same RollNo, ensuring that each record is unique and identifiable.

3. Primary Keys & Entity Integrity Constraints – No NULL Values

- **Explanation:**
  A **Primary Key** is a special key constraint that uniquely identifies each record in a table. The **Entity Integrity Constraint** ensures that a primary key attribute cannot contain NULL values, because if it were NULL, the record could not be uniquely identified.
- **Example:**
  In a Customer table, the CustomerID is set as the primary key. Every customer must have a valid, non-null CustomerID. If a record were inserted with a NULL value for CustomerID, it would violate the entity integrity constraint.

4. Referential Integrity Constraints – Enforce Valid Relationships Between Tables

- **Explanation:**
  Referential integrity ensures that relationships between tables remain consistent. It is implemented using **Foreign Keys**. A foreign key in one table refers to the primary key in another table. This constraint ensures that a record in the child table cannot reference a non-existent record in the parent table.
- **Example:**
  Consider two tables:
  - Orders(OrderID, CustomerID, OrderDate)
  - Customers(CustomerID, Name, City)
    Here, CustomerID in Orders is a foreign key referencing CustomerID in Customers. Referential integrity ensures that an order cannot be placed by a customer who does

not exist in the Customers table. If you try to insert an Order with CustomerID = 999 (and no such customer exists), the database will reject it.

Database Constraints Summary

Constraint	Definition	Rule Enforced	Example
Domain Constraints	Restrict the values stored in a column to valid data types and acceptable ranges.	Attribute values must match the defined data type and any specified condition (e.g., CHECK).	Age INT CHECK (Age >= 18) → Only integer values ≥ 18 allowed.
Key Constraints	Ensure that one or more attributes uniquely identify a record in a table.	No two rows can have the same key value.	In Student(RollNo, Name), RollNo must be unique for each student.
Entity Integrity (Primary Key)	Primary key uniquely identifies each record and must not be NULL.	Primary key columns cannot contain NULL values.	In Customer(CustomerID, Name), CustomerID as Primary Key → cannot be NULL.
Referential Integrity (Foreign Key)	Maintains consistency between related tables using foreign keys.	A foreign key value must match a primary key value in the parent table or be NULL.	In Orders(CustomerID) referencing Customers(CustomerID) → an order cannot exist for a non-existing customer.

These constraints play a crucial role in preventing data anomalies such as duplication, inconsistency, and invalid entries. By enforcing rules on data types, uniqueness, primary keys, and relationships between tables, they ensure that the stored data remains accurate, reliable, and meaningful. As a result, the database maintains data integrity, supports consistent query results, and provides a dependable foundation for decision-making and application development.

Check Your Progress

1. Explain the structure of the relational model and describe the types of keys used to maintain data integrity.

   …………………………………………………………………
   …………………………………………………………………
   …………………………………………………………………
2. Discuss different types of integrity constraints in the relational model with examples.

   …………………………………………………………………
   …………………………………………………………………
   …………………………………………………………………

4.5: Summary

The relational model represents data as a collection of tables, known as relations, where each row corresponds to a record and each column represents an attribute. Keys—such as primary, candidate, super, and foreign—play an essential role in ensuring data uniqueness and establishing relationships between tables. Integrity constraints, including entity, domain, and referential integrity, maintain data correctness and validity. The relational model promotes data independence, reduces redundancy, and supports efficient data manipulation through SQL. Its simplicity, flexibility, and strong theoretical foundation make it the most widely used data model today. By enforcing constraints and normalization rules, the relational model ensures consistent and reliable data, serving as the foundation for modern database systems used across all industries

4.6: Exercises

Multiple-Choice Questions:

1. **Which of the following uniquely identifies a record in a relational table?**

   a) Foreign Key

   b) Candidate Key

Notes

c) Primary Key

d) Alternate Key

Answer: c) Primary Key

2. **A set of one or more attributes that can uniquely identify a tuple in a relation is called:**

   a) Super Key

   b) Foreign Key

   c) Secondary Key

   d) Composite Attribute

   Answer: a) Super Key

3. **Which constraint ensures that no primary key attribute can have a NULL value?**

   a) Domain Constraint

   b) Entity Integrity Constraint

   c) Referential Integrity Constraint

   d) Key Constraint

   Answer: b) Entity Integrity Constraint

4. **Referential integrity ensures that:**

   a) All tuples have unique primary keys

   b) Foreign key values must match existing primary key values

   c) Data types are valid and consistent

   d) Columns have unique values

   Answer: b) Foreign key values must match existing primary key values

5. **Which of the following statements about the relational model is true?**

   a) Data is stored as unstructured objects

   b) Data is represented as interconnected tables

   c) Tables cannot have relationships

   d) Constraints are optional in relational design

   Answer: b) Data is represented as interconnected tables

Descriptive Questions

1. Explain the concept of the relational model and its significance in database design.
2. Describe the different types of keys used in a relational database with suitable examples.
3. What are integrity constraints? Explain domain, key, entity, and referential integrity constraints.
4. Discuss how relational constraints ensure accuracy and consistency in databases.

4.7: References and Suggested Readings

- Date, C. J. (2004). *An Introduction to Database Systems* (8th ed.). Pearson Education.
- Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson.
- Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database System Concepts* (7th ed.). McGraw Hill.
- Connolly, T., & Begg, C. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.

Block 2: Relational Data Modeling and Database Design

Unit 5: Theoretical Foundations of Relational Databases

Structure

5.1: Introduction

The theoretical foundations of relational databases are built upon the principles introduced by **Dr. E. F. Codd**, who formulated the **relational model** and proposed the **12 rules** for defining a true Relational Database Management System (RDBMS). These rules ensure that data is logically organized, consistently stored, and accessible through a structured query language (SQL).

Another fundamental concept is **Functional Dependency (FD)**, which defines relationships between attributes within a relation and helps maintain data integrity. **Armstrong's Axioms**—a set of inference rules—are used to derive and validate all functional dependencies. Together, these theories form the mathematical and logical backbone of database normalization, schema design, and optimization, ensuring reliability and consistency in data-driven systems.

5.2: Learning Outcomes

After completing this unit, students will be able to:

1. Understand and explain E. F. Codd's 12 rules for RDBMS.
2. Define functional dependency and explain its significance in database design.
3. Apply Armstrong's inference rules to derive new functional dependencies.
4. Identify candidate keys using attribute closure.
5. Recognize the role of functional dependencies in normalization and data integrity.

5.3: E.F. Codd's Rule

In the context of relational databases, Dr. Edgar F. Codd, the father of the relational database model, defined 12 rules (actually 13, including Rule 0) to qualify a system as a true relational database management system (RDBMS) of 1985. In relational databases these rules help keep data integrity, data consistency and help organize data for better management.

Figure 5.1: Codd's 12 Rule

Rule 0: Foundation Rule

Else, an application can be called RDBMS if it manages data completely based on relational capabilities. It is actually support for relational structures (tables, rows and columns), and agree with all other rules.

Rule 1: Information Rule

Every data in a relational database can only be stored in tables as rows and columns.

For Example: A "Student" table storing data in structured rows and columns.

Violation: Unstructured file-based data storage (e.g., text documents and spreadsheets)..

Rule 2: Guaranteed Access Rule

All data (value) needs to be retrievable by a combination of the table name, primary key, and the column name.

For instance in order to get a student email, you can do:

SQL

SQL Query: SELECT Email FROM Students WHERE Student_ID = 101;

Violation: When we have access to some data using physical addresses or pointers but not using SQL queries.

Rule 3: Systematic Treatment of NULL Values

NULL values should be handled consistently across the database as to indicate missing, unknown, or inapplicable data.

For instance: Suppose a student's phone number is not known, then the database must permit NULL in the "Phone" column.

Violation: Replacing NULL with arbitrary negativity (i.e. -1 or 99999).

Rule 4: Dynamic Online Catalog (Metadata Rule)

A relational database must use tables (SQL) to access metadata (schema, constraints, data types).

Sample: Fetching table structure with Inserting/fetching Create table:

SQL

PARSE_NO_STD_ERRORS_074001=095BD429L2+045C06%BQ+018F+0B6=2_SELECT * FROM INFORMATION_SCHEMA.TABLES;

Incorrect: When metadata is part of external files or system logs rather than tables..

Rule 5: Comprehensive Data Sub-language Rule

At least one complete language (SQL, for example) for data access, manipulation, and control must be provided by the system.

e.g.: SQL can insert, delete, update, and retrieve data.

Not even a devil (Can database that have different languages for different operation, one for query other for updates, etc.)

Rule 6: View Updating Rule

In case a view (virtual table) is constructed from base tables it should be updatable.

Example:

SQL

CREATE VIEW StudentEmails AS

-- 2. Retrieving specific columns (SELECT): SELECT Student_ID, Email FROM Students;

The underlying table must change if we change the StudentEmails view.

Violation: When views are read only and don't provide updates.

Rule 7: High-Level Insert, Update, Delete

Since inserts, updates, and deletes are performed on RDBMS using set-based operations and not row-based operations.

Example Updating Multiple Rows in One Query

SQL

Break: The update will excessive looping through each row

Rule 8: Physical Data Independence

We should be able to change our physical storage (where on disk, the way we handle indexing, etc.) but this should not affect our ability to access our data with an SQL query.

e.g Moving data from a disk to another disk should not break SQL queries

Violation: Moving data means re-writing application code.

Rule 9: Logical Data Independence

Logical structure change (adding/removing columns) should not affect the exist applications

Adding a column like Date_of_Birth — should not break existing queries that do not use it.

Violation: When applications break due to schema changes.

Have you watched "The Fall of the House of Usher"?

Rule 10: Integrity Independence

The integrity constraints (e.g., the primary key, the foreign key, NOT NULL) must be stored directly in the database and not in application code.

Example: Primary key constraint in SQL:

SQL

ALTER TABLE Employees ADD CONSTRAINT pk_emp PRIMARY KEY (Emp_ID);

Violation: When you enforce uniqueness in your application logic instead of letting the DB do it.

Rule 11: Distribution Independence

A characteristic requirement of RDBMS is support for distributed databases without changing of SQL statements.

For example, a query should be functional regardless if your data lives in one server or is sharded across multiple servers.

Violation: If you need to rewrite queries every time you move data between different locations.

Rule 12: Non-Subversion Rule

No mechanism to access the data (e.g., system-level commands) is allowed that bypasses relational security and integrity constraints.

For example: Direct database access through scripts must still respect constraints (such as NOT NULL, FOREIGN KEY)

Violation: When backend scripts allow a user to enter invalid data.

E.F. Codd's 12 rules guarantee a database adheres to the relational model. Most modern relational databases (e.g., MySQL, PostgreSQL, SQL Server, Oracle) follow most of these rules, though some systems (e.g., NoSQL databases) do not abide by all of them. With these very rules the relational databases became reliable and efficient for structured data management, as they allow the database management system (DBMS) to maintain the well-defined integrity rules that guarantees the correctness of data as well as independent from all applications and usable format of data.

5.4: Functional dependency, Armstrong's Inference rules

1. Functional Dependency (FD) in Relational Databases

What is Functional Dependency? Functional Dependency (FD) in a relational database design is an important concept as it describes the relationship between attributes in a relation. 3NF (TEACHER, HEAD_DEPARTMENT, HEAD_DEPARTMENT) or 3NF: A functional dependency from attributes X to attributes Y in a relation R is a possibility that X can functionally determine Y. Understanding this model is essential for maintaining data integrity to prevent redundancy as well as database normalization.

What Functional Dependency means

Functional Dependency is denoted as:

X→YX \rightarrow YX→Y

where:

- X (determinant): one or more set of attributes.
- Y (dependent) would be another (or group of) attribute.
- There is exactly one Y for every unique value of X.Example of Functional Dependency

Consider a "Student" table:

Student_ID	Name	Course	Department
101	Alice	DBMS	CS
102	Bob	OS	CS
103	Charlie	DBMS	IT
104	David	OS	CS

Functional Dependencies in this relation:

1. Student_ID → Name, Course, Department

- If we know the Student_ID, we can determine Name, Course, and Department.

2. Course → Department
   - If we know the Course, we can determine the Department.

Types of Functional Dependencies

1. Trivial Functional Dependency
   - If X→YX \rightarrow YX→Y, and Y⊆XY \subseteq XY⊆X, it is called trivial.
   - Example: {Student_ID, Name} → Name (Here, Name is already part of the left-hand side).
2. Non-Trivial Functional Dependency
   - If X→YX \rightarrow YX→Y, and YYY is not a subset of XXX, it is non-trivial.
   - Example: Student_ID → Name (Name is not part of Student_ID).
3. Completely Non-Trivial Dependency
   - If X→YX \rightarrow YX→Y, and X and Y do not overlap, it is completely non-trivial.
   - Example: Course → Department.

Importance of Functional Dependency in Normalization

- Used to identify candidate keys.
- Helps in decomposing tables while preserving dependencies.
- Essential for eliminating anomalies in database design.

Functional Dependencies in this relation:

Student_ID → Name | Course | Department

o We can find out Name, Course & Department if we know the Student_ID.

Course → Department

o We can even find out the Department if we have the Course

Functional Dependencies Types

Trivial Functional Dependency

If X→YX \rightarrow YX→Y, and Y⊆XY \subseteq XY⊆X, it is trivial.

o For instance: {Student_ID, Name} → Name (Where Name is already included in the LEFT SIDE).

Non-Trivial Functional Dependency

o That is, if X→YX \rightarrow YX→Y is non-trivial, and Y is not (a subset of) X (here: YXYXYXYY⊆XXX)

o Example: Student_ID → Name (Name does not lie in Student_ID).

Non-Trivial Dependency You ban the initial k elements.

o If X→YX \rightarrow YX→Y and not-overlapping X and Y then it's completely non-trivial.

o E.g., Course → Department.

Role of Functional Dependency in Normalization

• It is used to identify the candidate keys.

• Assists decomposition of tables such that dependencies are preserved.

• Crucial for removing anomalies from the design of the database.

2. Armstrong's Axioms (Inference Rules for Functional Dependencies)

Armstrong's Axioms (proposed by William W. Armstrong in 1974) are a set of inference rules used to derive all functional dependencies in a relational schema. These axioms form the basis for closure computation and normalization in relational databases.

Armstrong's Inference Rules

1. Reflexivity (Trivial Dependency Rule)
    - If Y is a subset of X, then X → Y holds.
    - Example: {Student_ID, Name} → Name (since Name is part of {Student_ID, Name}).
2. Augmentation Rule
    - If X → Y, then XZ → YZ (Adding more attributes does not affect dependency).
    - Example: If Student_ID → Name, then (Student_ID, Course) → (Name, Course).
3. Transitivity Rule
    - If X → Y and Y → Z, then X → Z.
    - Example: If Student_ID → Course and Course → Department, then Student_ID → Department.

Additional Derived Rules (Based on Armstrong's Axioms)

4. Union Rule
    - If X → Y and X → Z, then X → YZ.
    - Example: If Student_ID → Name and Student_ID → Course, then Student_ID → (Name, Course).
5. Decomposition Rule

- If X → YZ, then X → Y and X → Z separately.
- Example: If Employee_ID → (Employee_Name, Salary), then:
  - Employee_ID → Employee_Name
  - Employee_ID → Salary.

6. Pseudotransitivity Rule
   - If X → Y and WY → Z, then WX → Z.
   - Example: If Student_ID → Course and (Course, Department) → Professor, then (Student_ID, Department) → Professor.

Closure of Functional Dependencies (F+)

The closure of a set of functional dependencies is the complete set of dependencies that can be derived using Armstrong's Axioms.

- Given F = {A → B, B → C}, the closure F+ includes:
  1. A → B (Given)
  2. B → C (Given)
  3. A → C (By Transitivity)

Example of Computing Closure of an Attribute Set

Given: F = {A → B, B → C, C → D}, find A+ (Closure of A).

1. Start with A+ = {A}.
2. Since A → B, add B → A+ = {A, B}.
3. Since B → C, add C → A+ = {A, B, C}.
4. Since C → D, add D → A+ = {A, B, C, D}.
5. Final result: A+ = {A, B, C, D}.

Finding Candidate Keys Using Closure

- If A+ = All Attributes in Relation, then A is a candidate key.

Example: In R(Student_ID, Name, Course, Department) with FDs: Student_ID → Name, Course → Department,

- Closure of Student_ID: {Student_ID, Name, Course, Department} → Student_ID is a Candidate Key.

• Functional Dependency It specifies how attributes relate to maintain the integrity of the database.

• Using Armstrong's Axioms, one can derive all the dependencies and use this for schema normalization.

• \Database design\ : FD Closures and Candidate Key identification

Let's understand the concept of functional dependency using another real world example. Consider the following table:

EMPLOYEE Table

EmpID	EmpName	DeptID	DeptName
E101	Anjali	D01	HR
E102	Ramesh	D02	Finance
E103	Sohan	D01	HR
E104	Meena	D03	IT

Functional Dependencies in this Table

1. **EmpID → EmpName, DeptID, DeptName**
   - EmpID uniquely identifies the employee's name and their department.
   - If two rows have the same EmpID, they must also have the same EmpName, DeptID, and DeptName.
2. **DeptID → DeptName**
   - Every department ID corresponds to only one department name.
   - Example: D01 always means HR.
   - This prevents anomalies like assigning two different names (HR and Human Resources) to the same department ID.

Not a Functional Dependency

- **EmpName → EmpID** ❌
  This is invalid because two employees might share the same name (e.g., two employees named "Ramesh"). So EmpName cannot uniquely determine EmpID.

Why it Matters

- Functional Dependencies like **DeptID → DeptName** help us understand how to organize data.
- If DeptName depends only on DeptID, we should separate Department details into another table → this leads to **Normalization**.

In this chapter, we present the basic ideas, examples, and application for Functional Dependencies and Armstrong's Rules.

Check Your Progress

1. Define functional dependency and explain its importance in relational database design.

   …………………………………………………………………

   …………………………………………………………………

   …………………………………………………………………

2. Describe Armstrong's axioms and their role in deriving new functional dependencies.

   …………………………………………………………………

   …………………………………………………………………

   …………………………………………………………………

5.5: Summary

The theoretical foundations of relational databases provide the mathematical framework necessary for designing efficient and consistent data structures. Functional dependency (FD) expresses relationships between attributes in a relation, ensuring that certain attributes uniquely determine others. E. F. Codd's 12 rules set the principles for an ideal RDBMS, emphasizing logical consistency and independence. Armstrong's axioms—reflexivity, augmentation, and transitivity—form the basis for deriving new FDs. Concepts like attribute closure and canonical covers help identify candidate keys and remove redundancy. Understanding these principles ensures that database design adheres to integrity and efficiency standards. The theory bridges the gap between conceptual modeling and practical implementation, making it fundamental to database normalization and optimization.

5.6: Exercises

Multiple-Choice Questions:

1. **Who is known as the father of the relational database model?**

   a) Charles Bachman

   b) Edgar F. Codd

c) Michael Stonebraker
d) Jim Gray
Answer: b) Edgar F. Codd

2. **Which of the following correctly represents a functional dependency?**
   a) X ↔ Y
   b) X → Y
   c) X = Y
   d) X + Y
   Answer: b) X → Y

3. **Which of the following Armstrong's axioms states that if X → Y, then XZ → YZ?**
   a) Reflexivity
   b) Transitivity
   c) Augmentation
   d) Union
   Answer: c) Augmentation

4. **According to Codd's rules, which rule ensures that all data is accessible using a combination of table name, primary key, and column name?**
   a) Information Rule
   b) Guaranteed Access Rule
   c) Physical Data Independence
   d) Integrity Independence
   Answer: b) Guaranteed Access Rule

5. **Functional dependencies are mainly used in:**
   a) Transaction management
   b) Backup and recovery
   c) Database normalization and schema design
   d) Query execution
   Answer: c) Database normalization and schema design

Descriptive Questions

1. Discuss E. F. Codd's 12 rules for defining a relational database system.
2. Define functional dependency. Explain its types with suitable examples.
3. What are Armstrong's Axioms? Illustrate each rule with an example.
4. How do functional dependencies assist in identifying candidate keys and normalization?

5.7: References and Suggested Readings

- Ullman, J. D. (1988). *Principles of Database and Knowledge-Base Systems* (Vol. 1). Computer Science Press.
- Date, C. J. (2003). *Database in Depth: Relational Theory for Practitioners*. O'Reilly Media.
- Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson Education.
- Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (3rd ed.). McGraw Hill.

Block 2: Relational Data Modeling and Database Design

Unit 6: Decomposition and Normalization

Structure

6.1: Introduction

As databases grow in size and complexity, redundant data and anomalies can arise during insertion, deletion, and updates. **Normalization** is the systematic process of organizing data to reduce redundancy and improve data integrity. It involves decomposing large, unstructured tables into smaller, well-structured relations while maintaining meaningful relationships among them.

A good decomposition must satisfy two important properties: **Lossless Join** and **Dependency Preservation**. Various normal forms—**1NF, 2NF, 3NF, BCNF, and higher**—provide guidelines for ensuring minimal redundancy, logical consistency, and optimal storage. Normalization not only enhances query performance but also simplifies maintenance and improves scalability in relational databases

6.2: Learning Outcomes

After completing this unit, students will be able to:

1. Understand the concept and purpose of decomposition and normalization.
2. Explain the properties of good decomposition: lossless join and dependency preservation.
3. Apply normalization techniques (1NF, 2NF, 3NF, BCNF) to eliminate redundancy and anomalies.
4. Differentiate between various normal forms and identify when each is applicable.
5. Design efficient, normalized database schemas ensuring data integrity and performance.

6.3: Decomposition of Relations

Lossless Join and Dependency Preservation property

1. Decomposition of Relations

Decomposition is necessary in database design to address issues such as:

- Data Redundancy(duplicating the same data several times).
- Anomalies of Insertion, Deletion and Update (inconsistencies when we modify data).
- Integrity of Data (avoid the inconsistency in data).

Normalization is the process of organizing the columns (attributes) and tables (relations) of a database to minimize data redundancy and improve data integrity, and 1NF, 2NF, 3NF, and BCNF normalization techniques involve decomposing a relation into smaller relations so that functional dependencies are preserved, and joins are lossless..

Example:

Let us assume the relation R(Student_ID, Name, Course, Instructor, Department) with the following functional dependencies:

- Student_ID → Name, Course
- Course → Instructor, Department

Here Course → Instructor,Department is BCNF violating, so we decompose R into:

1. R1(Student_ID, Name, Course)
2. R2(Cours e, In structor, De part ment)

2. Lossless Join Property

Also, when relations decomposed, they can be combined back to the original collection of tuples without any loss of information: A Lossless Join Decomposition.

Definition:

A decomposition of a relation RRR into R1,R2,...,RnR1, R2,..., RnR1,R2,...,Rn is lossless join if:

RR1 \bowtie R2 \bowtie... \bowtie Rn = R R R 1 ⋈ R 2 ⋈... ⋈ R n = R.Strings are based on the way they are built or arranged.

So it had to not introduce any false (additional) tuples, and not lose any source data either.

Lossless Join Condition:

A lossless decomposition R1R1R1 and R2R2R2 of relation RRR is said to be lossless if:

Notes

More formally, the common attributes need to be a key for at least one of the newly generated relations.

Example of Lossless Join Decomposition

Let us consider a relation R(Employee_ID, Name, Department, Manager) with the following functional dependencies:

- Employee_ID | Name | Department
- Department → Manager

In order to comply with BCNF we break it into:

- R1(Employee ID, Name, Department))
- R2(Department, Manager)

Note that the common attribute Department in R1∩R2R1 \cap R2R1∩R2 is a key in R2, so this gives us lossless join.

R1⋈R2=R(No information is lost)R1 \bowtie R2 = R \quad \text{(No information is lost)}R1⋈R2=R(No information is lost)

But what if our decomposition is wrong and we write, for example:

• R1(Employee_ID, Name)

• R2(Employee_ID, Department, Manager)

The join between Department and Manager becomes lossy as we lose the mapping between them.

3. Dependency Preservation Property

We say a decomposition is dependency preserving, if all functional dependencies of the original relation are enforceable in the decomposed relations (without doing joins).

Definition:

A decomposition R1,R2,...,RnR1, R2,..., RnR1,R2,...,Rn of RRR is said to be dependency preserving if:

(F1∪F2∪...∪Fn)+=F+(F1∪F2∪...∪Fn)+

=(F1∪F2∪...∪Fn)+=F+(F1∪F2∪...∪Fn)+=F+

where:

- FFF is the initial set of functional dependencies.
- F1, F2,... Fn F1,F2,...FnF1,F2,...Fn are functional dependencies in decomposing relation
- F+F^{+}F+ is the closure of FFF, that is, all derived dependencies.

Why is Dependency Preservation Important?

• It provides a way to enforce integrity constraints (functional dependencies) in specific tables without the need for potentially expensive joins.

• If a decomposition loses dependencies, we may have to enforce some constraints by joins, resulting in loss of efficiency.

Checking for Dependency Preservation

To determine whether decomposition is dependency preserving, we find the closure of the union of the dependencies in decomposed relations and check if it is equal to the original closure.

Example of Dependency Preservation

Let's assume you have the relation R(A, B, C) and with the dependencies

1. A → B
2. B → C

Decomposing into:

• R1(A, B)

• R2(B, C)

Checking closures:

• F1 = {A → B}

• F2 = {B → C}

• Closure (F1 ∪ F2) + (F1 ∪ F2)^{+}(F1 ∪ F2) + contains A → B → C, hence all dependencies preserved.

Dependency preserved

Non-Dependency Preservation Example

Assuming we decompose R(A, B, C) into:

• R1(A, B)

• R2(A, C)

Since B → C is lost here, we need to perform joins to uphold this.

Not dependency preserving

4. Combining Lossless Join and Dependency Preservation

Ideal Decomposition

A good decomposition should satisfy both properties:

1. No Loss → Should not lose any data or add extra tuples.
2. Dependency Preservation → The functional dependencies must enforceable without costly joins.

However, in some cases, achieving both simultaneously may not be possible.

Trade-off Example

Suppose we have a relation R(A, B, C, D, E) Functional Dependencies:

• A → B
• B → C
• C → D, E

The BCNF decomposition would yield:
• R1(A, B)
• R2(B, C)
• R3(C, D, E)

The first option is lossless but not dependency preserving (because C → D, E spans multiple tables).

• Normalizations of database and removal of duplicate is only possible by means of decomposition.
• Lossless Join means no information is lost if relations are rejoined.
• Dependency Preservation assumes that the functional dependencies can be enforced without computing joins.
• The ideal decomposition preserves both properties, but compromises are sometimes necessary.

These tools serve as important checks for not only the viability of the proposed database schema but also for its performance: by ensuring that a schema maintains lossless join and dependency preservation, database designers are able to implement an optimal, normalized, high-performing database schema.

6.4: Normalization: First, Second, Third, BCNF, PJNF

Normalization is a systematic design process used in Relational Database Management Systems (RDBMS) to organize data into well-structured tables (relations) and columns (attributes).

The main objectives of normalization are:

1. To eliminate data redundancy (repeated storage of the same data).
2. To avoid update, insertion, and deletion anomalies.
3. To ensure data integrity and consistency.

Normalization is carried out in stages called Normal Forms (NF). Each stage applies a set of rules to improve the structure of the database.• Query performance (decreases data redundancy and anomalies)

1. First Normal Form (1NF)

A relation is in 1NF(First Normal Form) if:

1. They all contain atomic (or indivisible) values for each attribute.
2. No multi valued attributes: Each column holds a single value for each row.
3. This is a primary key requirement as each row must be uniquely identifiable.

Example of a Non-1NF Table:

Student_ID	Name	Courses	Phone Numbers
101	Alice	DBMS, OS	9876543210, 1234
102	Bob	OS, Networks	5556677889

Issues in Non-1NF Table:

• Multi-valued attributes: "Courses" and "Phone Numbers" have multiple values in a single column.

• Repeating groups: Some students have multiple phone numbers in a single field

Converting to 1NF (Atomic Values & Unique Rows):

Student_ID	Name	Course	Phone Number
101	Alice	DBMS	9876543210
101	Alice	OS	1234
102	Bob	OS	5556677889
102	Bob	Networks	5556677889

Now:

- Each column contains atomic values.
- No multi-valued attributes.
- Every single row can be uniquely identified.

2. Second Normal Form (2NF)

Definition:

Second Normal Form (2NF) if − A relation is in

1. It is already in 1NF.
2. partial dependencies). All non-primary attributes are fully functionally dependent on the primary key (no)

Notes

Example of a Non-2NF Table:

Order_ID	Product_ID	Product_Name	Quantity	Order_Date
O101	P01	Laptop	2	2024-01-01
O102	P02	Mouse	5	2024-01-02

Issues in Non-2NF Table:

- Primary Key = (Order_ID, Product_ID) (Composite Key).
- Partial Dependency:
  - Product_Name functionally depends on Product_ID, but not on Order_ID

Converting to 2NF (Eliminating Partial Dependencies):

Order Table:

Order_ID	Order_Date
O101	2024-01-01
O102	2024-01-02

Product Table:

Product_ID	Product_Name
P01	Laptop
P02	Mouse

Order_Details Table:

Order_ID	Product_ID	Quantity
O101	P01	2
O102	P02	5

Now:

No partial dependency (all non-key attributes depend on its primary key, fully).

Data is form correctly into lower tables.

3. Third Normal Form (3NF)

Definition:

A relation is in Third Normal Form(3NF) if:

1. It is already in 2NF.
2. No transitive dependency (a non-key attribute must not dependent on other non-key attributes).
3.

Example of a Non-3NF Table:

Student_ID	Name	Course	Instructor	Instructor_Phone
101	Alice	DBMS	Dr. John	9876543210
102	Bob	OS	Dr. Smith	5556677889

Issues in Non-3NF Table:

- Transitive Dependency:
    - Instructor_Phone depends on Instructor, not on Student_ID.

Converting to 3NF (Eliminating Transitive Dependency):

Student Table:

Student_ID	Name	Course	Instructor
101	Alice	DBMS	Dr. John
102	Bob	OS	Dr. Smith

Instructor Table:

Instructor	Instructor_Phone
Dr. John	9876543210
Dr. Smith	5556677889

Now:

No transitive dependency.

colspan="2" Attributes are directly dependent on the primary key.

4. Boyce-Codd Normal Form (BCNF)

Definition:

A relation is in BCNF if:

1. It is already in 3NF.
2. All determinants would be candidate key (We have no partial or transitive dependency now

Example of a Non-BCNF Table:

Employee_ID	Department	Manager
101	IT	John
102	HR	Sarah
103	IT	John

Issues in Non-BCNF Table:

- Manager depends on Department, not on Employee_ID (violating BCNF).

Converting to BCNF:

Department Table:

Department	Manager
IT	John
HR	Sarah

Employee Table:

Employee_ID	Department
101	IT
102	HR

None of the functional dependencies violates BCNF..

5. Projection-Join Normal Form (PJNF or 5NF)

Definition:

A relation is in 5NF (PJNF) if and only if:

1. It is already in BCNF.
2. There is no join dependency that cannot be enforced by decomposition of the relation

Example:

Consider a Supplier-Parts-Project relation:

Supplier_ID	Part_ID	Project_ID
S1	P1	J1
S1	P2	J2
S2	P1	J1

If we decompose into:

- Supplier-Part
- Part-Project
- Supplier-Project

We must ensure that recombining these tables retains all original data. PJNF eliminates join dependencies and guarantees that there's no more lossless decomposition to be had.

• 1NF → No multivalues attributes.

• 2NF → No partial dependency.

• 3NF → No transitive dependency.
• BCNF → Each determinant is a candidate key.
• PJNF (5NF) → All join dependencies.

The process of normalization adheres to a set of specific criteria, resulting in efficient databases that are scalable and free from logical conflicts.

Check Your Progress

1. Explain the need for normalization in database design and discuss the different normal forms.
   ………………………………………………………………
   ………………………………………………………………
   ………………………………………………………………
2. What are the properties of good decomposition? Explain with suitable examples.
   ………………………………………………………………
   ………………………………………………………………
   ………………………………………………………………

6.5: Summary

Normalization is a systematic approach used to eliminate redundancy and prevent anomalies in relational databases. It involves decomposing a relation into smaller, well-structured tables while preserving data integrity. The process is based on normal forms—1NF, 2NF, 3NF, and BCNF—each addressing specific types of redundancy and dependency issues. A good decomposition must be lossless, ensuring no information is lost, and dependency-preserving, maintaining all necessary constraints. Denormalization, on the other hand, is sometimes applied to improve query performance. Normalization enhances data consistency, scalability, and ease of maintenance. It is a crucial step in logical database design, ensuring that data remains organized, efficient, and reliable during transactions and updates.

6.6: Exercises

Multiple-Choice Questions:

1. **What is the main purpose of normalization in database design?**
   a) To increase redundancy

b) To remove redundancy and anomalies
c) To store data in multiple locations
d) To reduce query execution speed
Answer: b) To remove redundancy and anomalies

2. **Which property ensures that decomposed relations can be joined back without losing any information?**
a) Dependency Preservation
b) Lossless Join
c) Referential Integrity
d) Entity Integrity
Answer: b) Lossless Join

3. **A relation is in Second Normal Form (2NF) if it is in 1NF and:**
a) Has no partial dependency
b) Has no transitive dependency
c) Has a single attribute as key
d) Has no foreign key
Answer: a) Has no partial dependency

4. **Which normal form removes transitive dependencies?**
a) First Normal Form (1NF)
b) Second Normal Form (2NF)
c) Third Normal Form (3NF)
d) Boyce–Codd Normal Form (BCNF)
Answer: c) Third Normal Form (3NF)

5. **In normalization, dependency preservation ensures that:**
a) No data is lost during decomposition
b) Functional dependencies can still be enforced without joins
c) Every attribute depends on the primary key
d) All anomalies are eliminated
Answer: b) Functional dependencies can still be enforced without joins

Descriptive Questions

1. Define normalization. Explain its importance in relational database design.

2. What are the properties of good decomposition? Discuss lossless join and dependency preservation.
3. Explain different normal forms (1NF, 2NF, 3NF, and BCNF) with suitable examples.
4. Describe how normalization helps in eliminating data anomalies and improving database performance.

6.7: References and Suggested Readings

- Date, C. J. (2004). *An Introduction to Database Systems* (8th ed.). Pearson Education.
- Connolly, T., & Begg, C. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Pearson.
- Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson.
- Coronel, C., & Morris, S. (2019). *Database Systems: Design, Implementation, and Management* (13th ed.). Cengage Learning.

Block 3: SQL and Procedural SQL

Unit 7: Control Flow in SQL

Structure

7.1: Introduction

Control flow in SQL enables conditional and iterative execution of statements within stored programs such as stored procedures, functions, and triggers. Similar to programming languages, SQL control flow constructs help manage logical decision-making and repetition of tasks based on specific conditions.

Common control flow statements include IF…ELSE, CASE, WHILE, LOOP, REPEAT, and LEAVE. These statements enhance SQL's ability to perform procedural operations, automate decisions, and handle exceptions effectively within the database environment.

By mastering control flow, database developers can design robust stored programs that respond dynamically to varying input conditions, thus improving efficiency, maintainability, and accuracy of business logic executed at the database level.

7.2: Learning Outcomes

After completing this unit, students will be able to:

1. Explain the purpose and role of control flow statements in SQL.
2. Implement conditional logic using **IF**, **CASE**, and **ELSE** statements.
3. Utilize loops such as **WHILE**, **LOOP**, and **REPEAT** to manage repetitive operations.
4. Apply control flow constructs within stored procedures and functions.

5. Develop decision-based SQL programs that enhance automation and logical control.

7.3: Overview of SQL and Procedural SQL in MySQL

Structured Query Language (SQL) is the standard language used to interact with relational databases such as MySQL. SQL is primarily a **declarative language**, meaning the user specifies *what* needs to be done, and the database engine determines *how* to execute the query. Typical SQL operations include retrieving data using SELECT, inserting new records with INSERT, updating records using UPDATE, and deleting records with DELETE. SQL also includes commands for creating and altering database objects like tables, views, and indexes.

While SQL is powerful for data manipulation and definition, it is limited in terms of control flow and procedural logic. This is where **Procedural SQL** comes into play. In MySQL, Procedural SQL is implemented through stored routines such as **Stored Procedures, Functions, Triggers, and Events**. These allow programmers to embed procedural constructs like variables, loops, conditional statements, and error handling within SQL. With Procedural SQL, developers can write complex business logic directly inside the database, reducing reliance on external application code.

Merits of SQL (Declarative SQL)

1. **Simplicity and Readability:** SQL syntax is simple and closer to natural language, making it easy for beginners to learn and use.
2. **Data Independence:** SQL queries focus on *what* data is needed, not *how* it is fetched. The database optimizer decides the most efficient execution plan.
3. **Standardization:** SQL is an ANSI/ISO standard, and although implementations vary across platforms, the core features remain consistent.
4. **Powerful Data Manipulation:** SQL provides powerful operations such as joins, aggregations, grouping, and subqueries that simplify complex data retrieval tasks.
5. **Portability:** SQL code can often be migrated across different relational database systems with minimal changes.

Notes

Demerits of SQL

1. **Limited Procedural Capabilities:** SQL lacks constructs like loops and conditional branching, which restricts its ability to handle complex logic.
2. **Vendor Differences:** Although standardized, different databases (MySQL, Oracle, PostgreSQL) implement SQL with variations, leading to portability issues.
3. **Performance Concerns:** Poorly written queries can result in significant performance issues, as developers rely on the database engine for optimization.
4. **Steep Learning Curve for Complex Queries:** While basic SQL is easy, advanced queries involving multiple joins or nested subqueries can be difficult to understand and maintain.

Merits of Procedural SQL (MySQL Stored Routines)

1. **Enhanced Functionality:** Procedural SQL adds control structures like IF, CASE, WHILE, and LOOP, making it possible to write sophisticated logic inside the database.
2. **Performance Gains:** Business logic executed at the database layer often reduces network traffic and increases efficiency, as fewer calls are made between the application and the database.
3. **Reusability:** Stored procedures and functions can be reused across multiple applications, reducing redundancy and improving consistency.
4. **Security:** Access to sensitive operations can be controlled by granting permissions on procedures rather than directly on tables.
5. **Maintainability:** Changes in business logic can be managed at the database level without modifying multiple application programs.

Demerits of Procedural SQL

1. **Portability Issues:** Procedural SQL is not standardized across databases; MySQL's implementation may differ from Oracle PL/SQL or SQL Server T-SQL.
2. **Complexity:** Writing and debugging procedural code inside the database can be more difficult compared to application-level programming.

3. **Performance Bottlenecks:** Overuse of procedural SQL for tasks better suited to applications can overload the database server.
4. **Limited Tool Support:** Compared to general-purpose programming languages, Procedural SQL lacks advanced debugging, testing, and IDE support.
5. **Scalability Concerns:** Heavy reliance on stored routines can create scalability challenges in distributed systems where application servers are better suited for business logic.

SQL and Procedural SQL complement each other in MySQL. SQL excels at handling **data-oriented tasks**, while Procedural SQL adds **programming logic** to enhance functionality. While Procedural SQL provides performance, reusability, and security advantages, it also introduces complexity and portability issues. An effective database design balances both approaches: use SQL for straightforward data operations and Procedural SQL for encapsulating business rules where it improves efficiency and consistency.

7.4: Conditional statements and Iterative statements

Conditional statements

Conditional statements in PL/SQL allow your block to make decisions and execute different code paths depending on conditions.

Types of Conditional Statements:

- IF…THEN

Executes a block only if a condition is true.

```
IF <condition> THEN
-- statements
END IF;
```

- IF…THEN…ELSE

Chooses between two paths.

```
IF <condition> THEN
-- statements if true
ELSE
-- statements if false
END IF;
```

- IF…THEN…ELSIF…ELSE

Chooses among multiple conditions.

```
IF condition1 THEN
```

```
-- statements
ELSIF condition2 THEN
-- statements
ELSE
-- statements if none are true
END IF;
```

- CASE Statement

Used to evaluate expressions and choose one path:

```
CASE v_grade
WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Good');
ELSE DBMS_OUTPUT.PUT_LINE('Needs Improvement');
END CASE;
```

Example:

```
DECLARE
salary NUMBER := 60000;
BEGIN
IF salary > 50000 THEN
DBMS_OUTPUT.PUT_LINE('High Salary');
ELSE
DBMS_OUTPUT.PUT_LINE('Normal Salary');
END IF;
END;
```

Example:

```
DELIMITER //

CREATE PROCEDURE check_marks(IN student_marks INT, OUT
result VARCHAR(20))
BEGIN
   IF student_marks >= 75 THEN
      SET result = 'Distinction';
   ELSEIF student_marks >= 50 THEN
      SET result = 'Pass';
   ELSE
      SET result = 'Fail';
   END IF;
```

END $$

DELIMITER ;

How to Call the Procedure

```
-- Declare a variable for the output
SET @status = '';

-- Call the procedure with input marks
CALL check_marks(82, @status);

-- Display the result
SELECT @status AS 'Result';
```

Expected Output

If input marks = 82 →

Result

Distinction

If input marks = 55 →

Result

Pass

If input marks = 35 →

Result

Fail

2. Iterative Statements (Loops for Repetition)

Iterative Statements in PL/SQL

Iterative statements (loops) are used to repeat execution of a block of statements as long as a condition holds or for a fixed number of iterations.

Types of Loops:

- **LOOP…EXIT**
  A basic loop that repeats until you explicitly EXIT.
  ```
  LOOP
    -- statements
  ```

```
  EXIT WHEN condition;
END LOOP;
```

- **WHILE Loop**

  Repeats while a condition remains true

  ```
  WHILE condition LOOP
    -- statements
  END LOOP;
  ```

- **WHILE Loop**

  Repeats while a condition remains true.

  ```
  WHILE condition LOOP
    -- statements
  END LOOP;
  ```

- **FOR Loop**

  Repeats for a known range

  ```
  FOR counter IN start..end LOOP
    -- statements
  END LOOP;
  ```

Example:

```
DECLARE
  i NUMBER := 1;
BEGIN
  WHILE i <= 5 LOOP
    IF MOD(i,2)=0 THEN
      DBMS_OUTPUT.PUT_LINE('Even Number: ' || i);
    ELSE
      DBMS_OUTPUT.PUT_LINE('Odd Number: ' || i);
    END IF;
    i := i + 1;
  END LOOP;
END;
```

Output:

```
Odd Number: 1
Even Number: 2
Odd Number: 3
Even Number: 4
Odd Number: 5
```

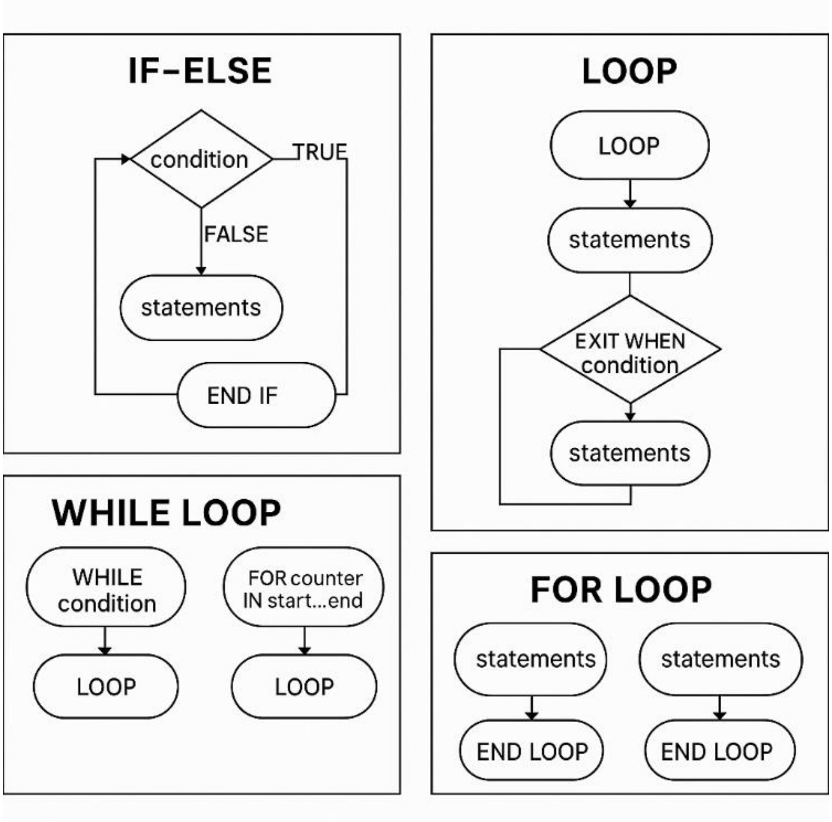

Figure.7.1: Conditional statements and Iterative statements

CHECK YOUR PROGRESS:

1. Explain how control flow statements like IF, CASE, and LOOP are used in SQL programming.

----

2. Explain how control flow statements like IF, CASE, and LOOP are used in SQL programming.

7.5: Summary

Control flow in SQL enables developers to add procedural logic to database operations. Statements such as IF, CASE, WHILE, and LOOP allow conditional execution and iterative processing within SQL programs. These constructs make SQL more powerful and flexible, allowing decision-making and repetitive task automation directly in the database layer. Control flow enhances procedural programming within SQL, improving efficiency and reducing dependency on external applications. It allows developers to manage complex workflows, data validation, and business rules effectively. When combined with stored procedures and functions, control flow promotes modularity and reusability. Thus, control flow forms the backbone of procedural SQL, helping bridge declarative and procedural paradigms.

7.6: Exercises

Multiple Choice Questions:

1. Which SQL control structure allows execution of statements based on a condition?

a) WHILE
b) IF…ELSE
c) LOOP
d) DECLARE

Answer: b) IF…ELSE

2. Which control statement is used to execute repetitive tasks in SQL?

a) CASE
b) LOOP
c) IF
d) EXIT

Answer: b) LOOP

3. The CASE statement in SQL is used for:

a) Creating tables
b) Data filtering
c) Conditional value selection
d) Declaring variables

Answer: c) Conditional value selection

4. Which of the following is used to immediately exit a loop in SQL?

a) RETURN
b) LEAVE
c) CONTINUE

d) BREAK

Answer: b) LEAVE

5. Which statement type is used for branching logic in stored procedures?

a) GOTO

b) CASE

c) DECLARE

d) ROLLBACK

Answer: b) CASE

Descriptive Questions:

1. Define control flow in SQL and explain its importance in stored programs.
2. Discuss different control flow statements in SQL with suitable examples.
3. Explain the difference between the IF…ELSE and CASE statements.
4. How can loops be used in SQL to automate repetitive operations?

7.7: References and Suggested Reading

- Beaulieu, A. (2009). *Learning SQL* (2nd ed.). O'Reilly Media.
- Feuerstein, S. (2014). *Oracle PL/SQL Programming* (6th ed.). O'Reilly Media.
- Stephens, R., & Plew, R. (2015). *Database Design and SQL for DBAs*. Sams Publishing.
- Groff, J. R., & Weinberg, P. N. (2014). *SQL: The Complete Reference* (3rd ed.). McGraw Hill Education.

Block 3: SQL and Procedural SQL

Unit 8: User Defined Function and Stored Procedure

Structure

8.1: Introduction

User-Defined Functions (UDFs) and Stored Procedures are powerful features in SQL that allow users to encapsulate frequently used operations and business logic within the database. A stored procedure is a precompiled block of SQL statements that can perform multiple operations such as data manipulation, control flow, and error handling. On the other hand, user-defined functions return a single value or a table as output and can be used directly in SQL expressions. Functions improve modularity, maintainability, and reusability of code, while procedures enhance performance and reduce network traffic by executing complex logic on the server side.

Together, UDFs and stored procedures promote structured programming in SQL, enabling developers to build scalable, secure, and efficient database applications.

8.2: Learning Outcomes

After completing this unit, students will be able to:

1. Define and differentiate between stored procedures and user-defined functions.
2. Create and execute stored procedures using SQL commands.
3. Develop scalar and table-valued user-defined functions.
4. Pass and handle parameters within functions and procedures.
5. Understand the benefits of modular programming and code reusability in SQL.

8.3: User-defined functions

A function in PL/SQL is a named block of code that performs a task and returns a single value. It can be called in SQL queries, PL/SQL blocks, or other functions/procedures.

Key Features:

- Must return exactly one value using RETURN.
- Can have parameters.
- Can be used in SELECT, WHERE, or HAVING clauses.

Syntax:

```
CREATE OR REPLACE FUNCTION function_name (param1
datatype, param2 datatype)
RETURN return_datatype
IS
  -- variable declarations
BEGIN
  -- function body
  RETURN value;   -- must return a value
END;
```

Example:

```
CREATE OR REPLACE FUNCTION get_bonus(salary NUMBER)
RETURN NUMBER
IS
BEGIN
  RETURN salary * 0.1;
END;
/
-- Calling the function
DECLARE
  bonus NUMBER;
BEGIN
  bonus := get_bonus(50000);
  DBMS_OUTPUT.PUT_LINE('Bonus: ' || bonus);
END;
/
```

Output:

Bonus: 5000

Types of User-Defined Functions in PL/SQL

In PL/SQL, user-defined functions are generally categorized based on how they are used in SQL statements.

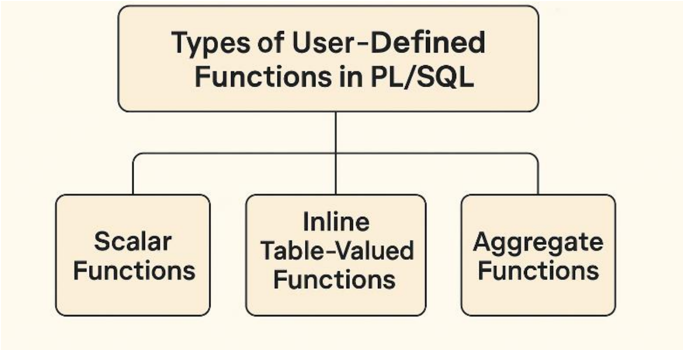

Figure.8.1: statements and Iterative statements

According to the SQL/PLSQL standards, there are three main types:

1. Scalar Functions

Return a single value for each call.

Typically used in SELECT, WHERE, HAVING, or in calculations.

Example:

```
DELIMITER //

CREATE FUNCTION get_bonus(salary DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
  RETURN salary * 0.1;
END //

DELIMITER ;
```

Explanation

DECIMAL(10,2) is used instead of NUMBER (Oracle type).

DETERMINISTIC means the function always returns the same result for the same input.

MySQL doesn't support CREATE OR REPLACE FUNCTION, so you must DROP first if it already exists.

```
DROP FUNCTION IF EXISTS get_bonus;
```

```
-- Example usage:
SELECT get_bonus(50000) AS Bonus;
```

Output:

Bonus
5000.00

When to use:

- To compute and return a single value (like bonus, tax, percentage, etc.).

8.4: Stored Procedures, Parameter types: IN, OUT and INOUT

Introduction to Stored Procedures

A Stored Procedure is a compiled SQL statement collection stored in the database that can be reused. Adan also helps in performing complex queries and operation efficiently and enhances performance, security and code reusability.

Why Use Stored Procedures?

Better performance – The SQL statements are compiled once and then executed multiple times.

Code Reusability – Means no need to write SQL queries again and again.

Security – Enforcement of access control can restrict changes to underlying tables.

Less Network Traffic – You send a procedure call instead of multiple SQL queries.

2. Creating a Stored Procedure

Basic Syntax (MySQL Example):

```
DELIMITER //
CREATE  PROCEDURE procedure_name(
BEGIN
-- SQL Statements
END //
DELIMITER;
```

• DELIMITER // sets a new statement terminator (because the procedure contains;).

• CREATE PROCEDURE defines (creates) the procedure.

• SQL logic is within BEGIN... END

• DELIMITER; resets the default terminator

3. Calling a Stored Procedure

Syntax:

CALL procedure_name();

Example:

```
DELIMITER //
CREATE PROCEDURE GetEmployees()
BEGIN
SELECT * FROM Employees;
END //
DELIMITER;
CALL GetEmployees();
```

Explanation:

• When you execute this procedure, it retrieves all the records from the Employees table.

• Call GetEmployees(); statement runs the procedure.

4. Parameter Types in Stored Procedures

We can also pass parameters to the stored routines.

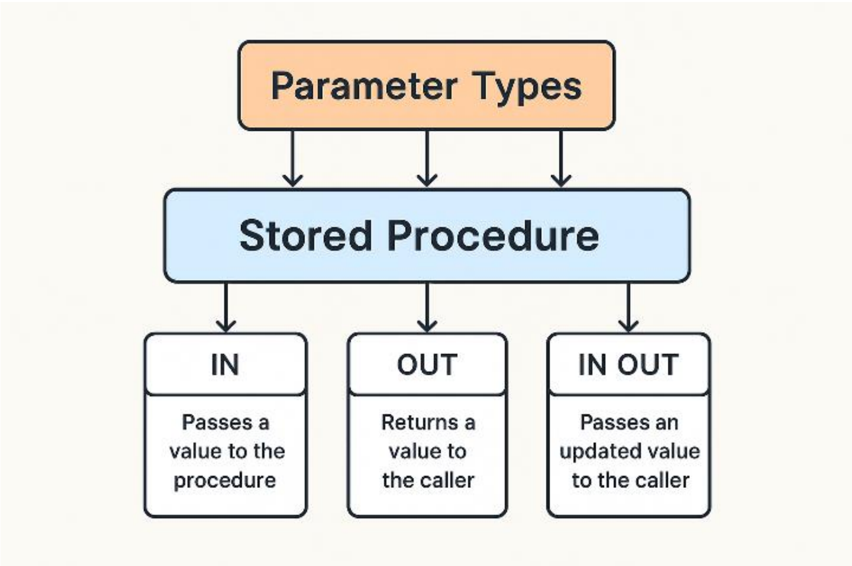

Fig.8.2: Parameter Types in Stored Procedures

Types of parameters are 3 types.

1. IN – For passing input values to the procedure.
2. OUT – Used for returning values from the procedure.

3. INOUT – Input and Output.

IN Parameter (Passing Input to Procedure)

• An IN parameter passes a value into the stored procedure.

• An IN parameter cannot be modified by the procedure.

Syntax:

DELIMITER //

CREATE PROCEDURE GetEmployeeByID(IN emp_id INT)

BEGIN

Roughly, if you were to be running SQL commands, your prompt input would be something like:

END //

DELIMITER;

Calling the Procedure:

CALL GetEmployeeByID(101);

Explanation:

• IN emp_id INT → Takes in an integer (Employee_ID) as input.

• Fetches employee information for a specific employee ID.

OUT Parameter (Returning a Value)

- OUT parameter returns a value.
- THE PROCEDURE CHANGES THE VALUE OF OUT PARAMETER

For Example: Number Of Employees Rehired

DELIMITER //

DELIMITER $$ CREATE PROCEDURE GetEmployeeCount(OUT total INT)$$ DELIMITER;

BEGIN

SELECT COUNT(*) INTO total FROM Employees;

END //

DELIMITER;

Calling the Procedure:

CALL GetEmployeeCount(@count);

SELECT @count; -- Show the value returned

Explanation:

• OUT total INT stores total employees.

• SELECT COUNT(*) INTO total saves the result to total.

• The CALL GetEmployeeCount(@count); saves the result to @count.

INOUT Parameter (Both Input and Output)

- The INOUT modifier is for directly changing value and returning.

Example: Modify Salary and Retrun updated Value

```
DELIMITER //
CREATE PROCEDURE UpdateSalary( INOUT emp_salary
DECIMAL(10,2), IN emp_id int)
BEGIN
UPDATE Employees SET Salary = emp_salary WHERE
Employee_ID = emp_id;
SELECT Salary INTO emp_salary FROM Employees WHERE
Employee_ID=emp_id
END //
DELIMITER;
```

Calling the Procedure:

```
SET @salary = 50000;
CALL UpdateSalary(101, @salary);
SELECT @salary; -- Updated salary
```

Explanation:

• INOUT emp_salary → input and output salary value
• Modifies the salary, and then reads back the new value.
• SET @salary = 50000; initializes the value
• The new salary is returned and stored in @salary.

5. Dropping a Stored Procedure

- To remove a procedure you are no longer interested in, use:

```
DROP PROCEDURE procedure_name IF EXISTS;
```

Example:

```
DROP PROCEDURE IF EXISTS GetEmployeeByID;
```

They optimized database operations by efficiency, security, and reusability.

- IN Parameters → Get input but cannot be changed.
- OUT Parameters ⇒ Return from procedures
- INOUT Parameters → Re-route values by modifying and returning.

Stored procedures help manage the database more quickly and securely in an effective and structural way, thus making them one of the key functions of all modern RDBMS systems.

Scenario based Problem Statement demonstrating IN, OUT, INOUT

Here is one scenario based problem statement and its solution that demonstrate the usage of IN, OUT, and INOUT parameter in real world.

A university maintains a student table containing student details. The administration often needs to:

1. **Search students by city (IN parameter use case).**
   - They want to input a city name and see how many students are from that city.
2. **Fetch student's name using their ID (OUT parameter use case).**
   - They provide the student's ID, and the procedure should return the student's name.
3. **Update a student's marks (INOUT parameter use case).**
   - They provide the student's ID and current marks as input.
   - The system adds 10 bonus marks and returns the updated marks back to the user.

Database Setup

```
-- Create table
CREATE TABLE student (
   id INT PRIMARY KEY,
   name VARCHAR(50),
   city VARCHAR(50),
   marks INT
);

-- Insert sample data
INSERT INTO student (id, name, city, marks) VALUES
(1, 'Amit', 'Raipur', 75),
(2, 'Sneha', 'Bhilai', 82),
(3, 'Rohan', 'Raipur', 68),
(4, 'Priya', 'Durg', 90);
```

Stored Procedure with IN, OUT, and INOUT

```
DELIMITER $$
```

```
CREATE PROCEDURE manage_student_data(
    IN in_city VARCHAR(50),       -- IN parameter
    IN in_id INT,                 -- IN parameter
    OUT out_name VARCHAR(50),     -- OUT parameter
    INOUT inout_marks INT         -- INOUT parameter
)
BEGIN
    -- 1. Using IN parameter: Count students from a given city
    SELECT COUNT(*) AS total_students
    FROM student
    WHERE city = in_city;

    -- 2. Using OUT parameter: Fetch student name by ID
    SELECT name INTO out_name
    FROM student
    WHERE id = in_id;

    -- 3. Using INOUT parameter: Update marks with bonus
    SET inout_marks = inout_marks + 10;

    UPDATE student
    SET marks = inout_marks
    WHERE id = in_id;
END$$

DELIMITER ;
```

Execution & Testing

Step 1: Declare variables to hold OUT and INOUT values

```
SET @city = 'Raipur';
SET @sid = 1;          -- Student ID
SET @sname = '';       -- OUT variable
SET @smarks = 75;      -- INOUT variable (initial marks)
```

Step 2: Call the procedure

```
CALL manage_student_data(@city, @sid, @sname, @smarks);
```

Step 3: Check results

```
-- OUT parameter result
SELECT @sname AS StudentName;
```

-- INOUT parameter result
SELECT @smarks AS UpdatedMarks;

Expected Output

1. From **IN parameter** (city = 'Raipur')
2. total_students
3. 2
4. From **OUT parameter** (student with id = 1)
5. StudentName
6. Amit
7. From **INOUT parameter** (original marks = 75, bonus = 10)
8. UpdatedMarks
9. 85

And in the table, Amit's marks will be updated to **85**.

Check Your Progress:

1.Define user-defined functions and stored procedures and explain their advantages in database programming.

2.Differentiate between scalar and table-valued functions with examples.

8.5: Summary

User-defined functions (UDFs) and stored procedures enhance modular programming within SQL databases. Stored procedures are precompiled SQL blocks that perform specific operations and can be reused across applications. User-defined functions return a value or table and can be invoked within SQL expressions. Both improve performance by reducing network traffic and execution time. They also promote code reusability, maintainability, and security by centralizing business logic within the database. UDFs and procedures support input and output parameters, control structures, and error handling mechanisms. Together, they help in building scalable, efficient, and well-organized database applications, ensuring consistent implementation of business logic and validation rules.

8.6: Exercises

Multiple Choice Questions:

1. Which SQL command is used to create a stored procedure?

a) CREATE FUNCTION
b) CREATE PROCEDURE
c) CREATE PROGRAM
d) DEFINE PROCEDURE

Answer: b) CREATE PROCEDURE

2. A user-defined function in SQL must always:

a) Return a value
b) Modify a table
c) Commit transactions
d) Drop indexes

Answer: a) Return a value

3. Which of the following can be used inside a stored procedure but not inside a user-defined function?

a) SELECT statement
b) DML statements (INSERT, UPDATE, DELETE)
c) RETURN statement
d) Variable declaration

Answer: b) DML statements (INSERT, UPDATE, DELETE)

4. Which keyword is used to execute a stored procedure in SQL?

a) EXEC
b) RUN
c) START
d) CALL

Answer: a) EXEC (or CALL in MySQL)

5. Which of the following is an advantage of using stored procedures?

a) Slower execution speed
b) Increased network traffic

c) Better performance and security

d) Lack of modularity

Answer: c) Better performance and security

Descriptive Questions:

1. Define stored procedures and explain their advantages with examples.
2. Differentiate between user-defined functions and stored procedures.
3. Explain the syntax and steps involved in creating a user-defined function.
4. Describe the concept of modular programming in SQL and its benefits.

8.7: References and Suggested Reading

- Hernandez, M. J. (2013). *Database Design for Mere Mortals* (3rd ed.). Addison-Wesley Professional.
- Feuerstein, S. (2014). *Oracle PL/SQL Programming* (6th ed.). O'Reilly Media.
- Groff, J. R., & Weinberg, P. N. (2014). *SQL: The Complete Reference* (3rd ed.). McGraw Hill.
- Stephens, R. (2016). *Beginning SQL* (3rd ed.). Wrox.

Block 3: SQL and Procedural SQL

Unit 9: Triggers

Structure

9.1: Introduction

A **trigger** is a special kind of stored program that automatically executes in response to specific database events such as **INSERT**, **UPDATE**, or **DELETE** operations on a table. Triggers are used to enforce business rules, maintain data integrity, and perform automatic auditing or logging of data changes.

Each trigger is associated with a table and can be defined to execute **BEFORE** or **AFTER** a specified event. Triggers operate transparently to users and ensure that critical tasks such as validation, synchronization, and backup occur without manual intervention.

By using triggers, database designers can automate workflows, enhance consistency, and ensure that organizational rules are always upheld at the data level.

9.2: Learning Outcomes

After completing this unit, students will be able to:

1. Explain the concept and purpose of triggers in database systems.
2. Differentiate between BEFORE and AFTER triggers.
3. Describe the events that can activate a trigger (INSERT, UPDATE, DELETE).
4. Design and implement simple triggers to automate database operations.
5. Understand the role of triggers in enforcing data integrity and business rules.

9.3: Triggers: Introduction, Needs, Before trigger and After trigger

1. Introduction to Triggers

Triggers are special types of stored procedures in a database that are automatically executed when an event occurs on a table. The events can be INSERT, UPDATE, or DELETE operations.

Business rules, data integrity, automation, and security controls are all uses of triggers. Triggers are similar to stored procedures in that you cannot call them manually; they run automatically in response to the event with which they are associated.

Features of Triggers:

Automatic Execution – Automatically fires on occurrence of referenced Event.

Event-Driven – Triggers on INSERT, UPDATE, or DELETE.

Validates Business Rules – Prevents Invalid Data Churn

Data Integrity – Ensures consistent data across tables.

Makes Unauthorized Changes Impossible – Data Access and Validation

2. Need for Triggers

We will use triggers in the database. Triggers are very important in removing data constraints, audit log creation, or enforce a business rule automatically. Some key use cases include:

Enforcing Business Rules

Example: Ensuring employees don't pay salaries under minimum wage.

Maintaining Data Integrity

Example: automatically updating child records if any parent record is updated to maintain a foreign key constraint.

Auditing and Logging Changes

Example: Audit trail for changes to sensitive tables like financial transactions.

Preventing Invalid Transactions

Example: Not allowing account balance updates for negative amounts.

Automating Actions

Example: Send an email notification every time a new user is added to the system.

3. Types of Triggers

Triggers are classified based on when they execute relative to the event:

Type	Executes Before/After	Applies to INSERT, UPDATE, DELETE	Use Case
BEFORE Trigger	Before the event	YES	Validation & Prevention
AFTER Trigger	After the event	YES	Logging &Post-processing

4. BEFORE Triggers

Definition:

A BEFORE Trigger runs before an INSERT, UPDATE, or DELETE operation. Its common on use is to verify data and prevent incorrect alterations.

Example 1: BEFORE INSERT Trigger (Preventing Invalid Salary Entry)

DELIMITER //

The order of the two is the subject of this post.CREATE TRIGGER Before_Insert_Employee

BEFORE INSERT ON Employees

FOR EACH ROW

BEGIN

IF NEW. Salary= 30000) THEN SET MESSAGE_TEXT = 'Salary should be atleast 30000';

END IF;

END //

DELIMITER;

Explanation:

- BEFORE INSERT – Executes before inserting data into the Employees table.
- NEW.Salary – Refers to the salary value being inserted.
- SIGNAL SQLSTATE '45000' – Throws an error if salary is less than 30,000.

Calling the Trigger:

INSERT INTO Employees (Employee_ID, Name, Salary)VALUES (101, 'Alice', 25000);

Output:

ERROR Salary Must be Greater Than 30,000

To ensure no invalid salary can be inserted the trigger can be used.

Example 2: BEFORE UPDATE Trigger (Restricting Price Reduction by More Than 50%)

```
DELIMITER //
< ` CREATE  TRIGGER Before_Update_Product
BEFORE UPDATE ON Products
FOR EACH ROW
BEGIN
IF NEW. Price< (OLD. Price * 0.5) THEN
SIGNAL SQLSTATE '45000'
SET MESSAGE_TEXT = 'Reduced price can't be  more than 50%';
END IF;
END //
DELIMITER;
```

Explanation:

- BEFORE UPDATE – Executes before product prices are updated.
- OLD. Price – Refers to the current price.
- NEW. Price – The new price that is being updated
- The trigger throws an error if the new price is less than 50% of the old price.

Calling the Trigger:

UPDATE Products SET Price = 20 WHERE Product_ID = 1; -- Previous price 100

Output:

vbnet

The trigger saves us from having to make a drastic price cut.

5. AFTER Triggers

Definition:

An AFTER Trigger runs after INSERT, UPDATE, or DELETE statement. It is often used to log, audit, and update reference tables.

Example 1: AFTER INSERT Trigger (Logging New Employee Addition)

```
DELIMITER //
SQL --  CREATE TRIGGER After_Insert_Employee
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
```

"INSERT INTO Employee_Log (Employee_ID, Action, Timestamp)
VALUES (NEW. Insert into Table (Employee_ID, 'Inserted',
NOW());
END //
DELIMITER;

Explanation:

- AFTER INSERT – Trigger works after inserting an employee.
- NEW. Employee_ID – Gets the new employee's Identification.
- NOW() – Saves the current date and time.
- Outputs log entry into Employee_Log table

Calling the Trigger:

INSERT INTO Employees VALUES (102, 'Bob', 50000);
Employee_Log Table (Post Trigger Execution):
Log_ID Employee_ID Action Timestamp
Row Status | ID | Data | Date/Time | 1 Inserted | 2024-03-09 12:30:00
|

The trigger logs the new joins automatically.

Example 2: AFTER DELETE Trigger (Archiving Deleted Orders)

```
DELIMITER //
CREATE TRIGGER After_Delete_Order
AFTER DELETE ON Orders
FOR EACH ROW
BEGIN
INSERT INTO Order_Archive (Order_ID, Customer_ID,
Order_Date) VALUES
VALUES (OLD. Order_ID, OLD. Customer_ID, OLD. Order_Date);
END //
DELIMITER;
```

Explanation:

- AFTER DELETE – Triggered post deletion of the order.
- OLD. Order_ID – The Order which has been deleted.
- Deletes orders into an archive table (Order_Archive).

Calling the Trigger:

The trigger also keeps deleted orders in the archive table.

6. Dropping a Trigger

To drop a trigger, which may be, no longer needed:

DROP TRIGGER IF EXISTS trigger_name;

Example:

DROP TRIGGER IF EXISTS Before_Insert_Employee;

Triggers increase database automation, security, and integrity by enforcing business rules at the database level.

• Triggers BEFORE → Validate on execution (BEFORE INSERT/UPDATE/DELETE).

• AFTER Triggers → Execute after the execution of actions (AFTER INSERT/UPDATE/DELETE).

Triggers are a major component of data management in relational databases, allowing automated checks on data, logging, fraud prevention, and ensuring data consistency.

9.4: Scenario-Based Problems with Solutions

Let's take some **scenario-based problems with solutions** to demonstrate **MySQL Procedure, Function, and Trigger**. Each example will include a problem statement, the MySQL solution, and explanation.

1. MySQL Procedure Example

Scenario-Based Problem Statement:

A company maintains an Employee table. The HR department wants to **update the salary of an employee** by a given percentage whenever there is a performance appraisal. The operation should be done using a stored procedure.

Employee Table:

```
CREATE TABLE Employee (
   EmpID INT PRIMARY KEY,
   EmpName VARCHAR(50),
   Salary DECIMAL(10,2)
);
```

Solution – Procedure

```
DELIMITER //

CREATE PROCEDURE UpdateSalary(
   IN p_EmpID INT,
   IN p_Percent DECIMAL(5,2)
)
BEGIN
   UPDATE Employee
```

```
   SET Salary = Salary + (Salary * p_Percent / 100)
   WHERE EmpID = p_EmpID;
END //

DELIMITER ;
```

Usage:

```
CALL UpdateSalary(101, 10); -- Increases salary of EmpID 101 by 10%
```

Explanation:

- IN parameters are used to pass employee ID and percentage.
- Procedure encapsulates logic for reusability.
- Avoids writing update statements repeatedly in application code.

2. MySQL Function Example

Scenario-Based Problem Statement:

A school maintains a Student table. The school wants to calculate **grade points** based on marks for each student. Marks to grade points mapping is:

- Marks ≥ 90 → 10
- Marks ≥ 80 → 9
- Marks ≥ 70 → 8
- Marks ≥ 60 → 7
- Marks < 60 → 6

The calculation should be implemented as a function.

Student Table:

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

Solution – Function

```
DELIMITER //

CREATE FUNCTION GetGrade(Marks INT) RETURNS INT
BEGIN
   DECLARE grade INT;
```

```
    IF Marks >= 90 THEN
        SET grade = 10;
    ELSEIF Marks >= 80 THEN
        SET grade = 9;
    ELSEIF Marks >= 70 THEN
        SET grade = 8;
    ELSEIF Marks >= 60 THEN
        SET grade = 7;
    ELSE
        SET grade = 6;
    END IF;

    RETURN grade;
END //

DELIMITER ;
```

Usage:

```
SELECT Name, Marks, GetGrade(Marks) AS Grade FROM Student;
```

Explanation:

- Function takes Marks as input and returns grade points.
- Can be used in SELECT statements to calculate grades on-the-fly.
- Ensures consistency in grading logic.

3. MySQL Trigger Example

Scenario-Based Problem Statement:

A bank maintains an Accounts table. Whenever a **new transaction** is inserted in the Transactions table, the **account balance should automatically update**. This should be achieved using a trigger.

Accounts Table:

```
CREATE TABLE Accounts (
    AccountID INT PRIMARY KEY,
    AccountHolder VARCHAR(50),
    Balance DECIMAL(10,2)
);

CREATE TABLE Transactions (
    TransactionID INT PRIMARY KEY,
```

```
   AccountID INT,
   Amount DECIMAL(10,2),
   TransactionType ENUM('Credit','Debit'),
   FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)
);
```

Solution – Trigger

```
DELIMITER //

CREATE TRIGGER UpdateBalance
AFTER INSERT ON Transactions
FOR EACH ROW
BEGIN
   IF NEW.TransactionType = 'Credit' THEN
      UPDATE Accounts
      SET Balance = Balance + NEW.Amount
      WHERE AccountID = NEW.AccountID;
   ELSEIF NEW.TransactionType = 'Debit' THEN
      UPDATE Accounts
      SET Balance = Balance - NEW.Amount
      WHERE AccountID = NEW.AccountID;
   END IF;
END //

DELIMITER ;
```

Usage:

```
INSERT INTO Transactions (TransactionID, AccountID, Amount, TransactionType)
VALUES (1, 101, 5000, 'Credit');
```

Explanation:

- Trigger automatically updates Accounts.Balance whenever a new transaction is inserted.
- Ensures **data integrity** between Accounts and Transactions.
- No need for manual balance updates from application code.

9.5: Summary of Use Cases:

Feature	Use Case
Procedure	Perform repetitive operations like updating salaries
Function	Return computed values like grade points in queries
Trigger	Automatically enforce rules like updating account balance on transaction

Check your Progress:

1. What is a database trigger? Explain its types and typical use cases.

----

2. Discuss the advantages and potential challenges of using triggers in database management.

9.6: Summary

Triggers are special database programs that automatically execute in response to specific events such as INSERT, UPDATE, or DELETE operations. They help maintain data integrity, enforce business rules, and automate system-level tasks. Triggers can be classified as BEFORE or AFTER, depending on their execution timing relative to the triggering event. They are commonly used for auditing, validation, and maintaining derived data. While triggers enhance automation and consistency, excessive or poorly designed triggers can affect performance and complicate debugging. Proper design and testing ensure that triggers complement other database constraints. In modern systems, triggers are integral for implementing event-driven logic and enforcing data consistency across tables

9.7: Exercises

Multiple Choice Questions:

1. **A trigger in SQL is automatically executed when:**

a) A specific event occurs in the database
b) The user runs a stored procedure
c) The database starts
d) A backup is performed

Answer: a) A specific event occurs in the database

2. **Which of the following statements can activate a trigger?**

a) SELECT
b) COMMIT
c) INSERT, UPDATE, or DELETE
d) ROLLBACK

Answer: c) INSERT, UPDATE, or DELETE

3. **A trigger that executes before a data modification is called:**

a) Pre-Trigger
b) BEFORE Trigger
c) PRIOR Trigger
d) INITIAL Trigger

Answer: b) BEFORE Trigger

4. **Which command is used to remove an existing trigger in SQL?**

a) DELETE TRIGGER
b) REMOVE TRIGGER
c) DROP TRIGGER
d) DISABLE TRIGGER

Answer: c) DROP TRIGGER

5. **Triggers are most commonly used to:**

a) Create new tables
b) Automate data validation and auditing
c) Execute external scripts
d) Backup database files

Answer: b) Automate data validation and auditing

Descriptive Questions:

1. Define triggers and explain their purpose in SQL.
2. Differentiate between BEFORE and AFTER triggers with examples.

3. Explain the types of triggering events and their use cases.
4. Discuss the advantages and potential drawbacks of using triggers in databases.

9.8: References and Suggested Reading

- Feuerstein, S. (2014). *Oracle PL/SQL Programming* (6th ed.). O'Reilly Media.
- Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (3rd ed.). McGraw Hill.
- Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson.
- Groff, J. R., & Weinberg, P. N. (2014). *SQL: The Complete Reference* (3rd ed.). McGraw Hill.

Block 4: Transaction Management and Concurrency

Unit 10: Transactions

Structure

10.1: Introduction

A transaction is a logical unit of work that consists of one or more SQL operations performed on a database. The main purpose of a transaction is to ensure that the database remains in a consistent and reliable state even in the presence of system failures or concurrent access.

Each transaction follows the ACID properties — Atomicity, Consistency, Isolation, and Durability — which guarantee reliability and correctness of data operations. Transactions can be initiated, committed, or rolled back, depending on whether all operations complete successfully or an error occurs.

Transaction management is an essential aspect of Database Management Systems (DBMS) because it ensures that complex operations are processed safely and efficiently, especially in multi-user environments where several transactions may execute simultaneously.

10.2: Learning Outcomes

After completing this unit, students will be able to:

1. Define the concept of transactions and their role in database systems.
2. Explain the ACID properties and their importance for maintaining data integrity.
3. Describe transaction states such as active, partially committed, failed, aborted, and committed.
4. Understand commit and rollback operations in SQL.
5. Appreciate the significance of transaction management in multi-user database systems.

10.3: Transaction: Introduction, Transaction Model

A transaction is a series of one or more SQL operation from a database that is executed as one logical unit of work. Transactions help maintain database consistency and reliability in the case of system crashes or power failures, as well as concurrent user operations. In multi-user database environments, when several users at the same time perform operations, data integrity has to be maintained. If transactions were not handled correctly, partial operations could corrupt or render data inconsistent. In banking systems, if a customer transfers money from one account to another, both the debit and the credit operation has to be successful. If the debit is successful but the credit fails the money will be lost. Transactions allow you to group both operations so they either both complete or both fail to prevent this from happening.

1.Real-Life Example of a Transaction

For instance let's say a bank customer transfers $500 from Account A to Account B. The transaction involves two operations:

1. Update Accounts Set Balance = Balance - 500 Where Account_ID = 1; Deduct $500 from Account A.
2. Update Accounts: Add $500 in Account B (UPDATE Accounts SET Balance = Balance + 500 WHERE Account_ID = 2;).

For consistency, the rollback should be performed in case the second operation fails, so that no operation will have to revert back

2. Understanding the ACID Properties of Transactions

A transaction must follow four key properties, known as the ACID properties:

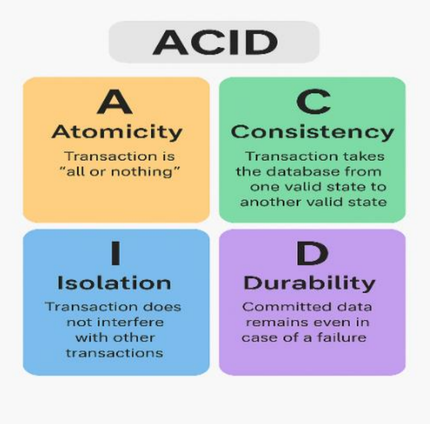

Figure: 10.1-ACID Properties

Atomicity (All or Nothing Rule)

- A transaction is either successfully completed or aborted.
- • All changes made in a transaction must either be completed successfully or rolled back if an error occurs.

- For example, if a payment fails after taking money out from an account, the amount has to be refunded.

Consistency (Maintaining Database Validity)

- Every transaction should transform the database from one valid state to another.
- Any changes made need to comply with the database constraints, rules, and integrity checks
- For example, if an order is placed in an e-commerce system, then the stock count must be decreased accordingly.

Isolation (Preventing Concurrent Transaction Interference)

- Concurrent transactions must not interfere with one another's execution.
- All transactions are executed sequentially and the final outcome must look like transactions were executed sequentially.
- Illustration: When two users book the last ticket to a movie, both should fail, but one should succeed.

Durability (Permanent Storage of Committed Data)

- Once a transaction commits, its changes have to be persisted even in case of a system crash.
- Example: After conducting an online banking transaction, new balance should not be lost after server crash

3. Transaction Lifecycle and States

A transaction progresses through multiple states during its execution:

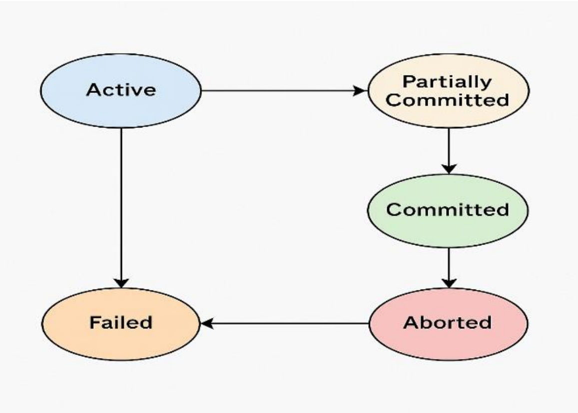

Fig. 10.2- Transaction Lifecycle and States

Transaction State	Description
Active	The transaction has started and is currently executing.

Partially Committed	All SQL operations are completed, but not yet permanently saved.
Committed	Changes are successfully stored in the database.
Failed	An error occurred, causing the transaction to fail.
Aborted (Rollback)	The transaction is undone, restoring the database to its previous state.

State Transitions in a Transaction

1. Execution of a transaction starts in the ACTIVE state.
2. If all operations succeed, changes to PARTIALLY COMMITTED.
3. Once the COMMIT statement is executed, the transaction is CONVERTED to COMMITTED, which saves the changes permanently.
4. If any step fails, the transaction will enter into FAILED state.
5. In the event of a failure, a ROLLBACK command restores the database, transitioning it to the ABORTED state.

Example of Transaction Lifecycle in SQL

ROLLBACK; -- Undo all changes
START TRANSACTION;
UPDATE Accounts Set Balance = Balance-500 WHERE Account_ID = 1; -- Decrease money
UPDATE Accounts SET Balance = Balance - 500 WHERE Account_ID = 1; -- Withdraw money
COMMIT; # Make changes permanent
If any error occurs before the COMMIT, we can roll it back

4. Transaction Models in Database Management Systems (DBMS)

A transaction model specifies the functioning of transactions in a database system with guarantee of ACID properties. They assist in managing transactions, concurrency, and failure recovery efficiently.

Flat Transactions (Simple Transactions)

- An abstract model for the simplest transaction in which a series of operations are performed as a single unit.
- Strictly follows ACID properties: Thus if there is a failure in a part of transaction, the complete transaction will be rolled back.
- for instance: Moving currency between bank accounts.

Nested Transactions (Transactions Inside Transactions)

• Sub-transactions which are executed independently within the main transaction.

• Supports rolling back a single part of a compound transaction if it is a sub-transaction and the parent transaction can still succeed.
• Example: In case of an online shopping system placing an order is:

1. Minusing money from the customer account (Sub-transaction 1)
2. Stock levels update (Sub-transaction 2)
3. Sending a confirmation mail (Sub-Transaction 3)

Whether Sub-transaction 3 fails or succeeds, the payment and update of stock will still be valid.

Long-Duration Transactions (Used in Batch Processing & Cloud Systems)

- Long-running transactions (hours or days)
- Used extensively in scientific computing, cloud applications, and batch data processing.
- Sample: Month-end for payroll processing for thousands of employees

5. Concurrency Control in Transactions

Data inconsistencies arise when two or more transactions access the same data and try to change it at the same time. Concurrency control mechanisms must be implemented in database systems to avoid conflicts.

Problems Caused by Concurrent Transactions

- lost update → two transactions updating the same data; one update gets lost.
- Dirty Read → This occurs in case if one transaction reads data that another transaction has not yet committed.
- Non-Repeatable Read → A transaction is read multiple times but value change from another transaction.

Techniques for Concurrency Control

- Locking Mechanisms (Shared & Exclusive Locks) → Avoid two transactions modifying the same data at the same time.
- Timestamp Ordering→ Guarantees the correct sequence of executing transactions.
- Optimistic Concurrency Control →Freely allows transactions to execute, checks for conflicts before committing.

6. Handling Failures and Recovery in Transactions

Transaction Failures can be attributed to:

System Crashes → Power failures, OS crashes.

Deadlock → Transaction(s) waiting infinitely for each other.

Concurrency Issues → when multiple transactions conflict with one another.

Recovery Mechanisms:

1. Undo (Rollback) – Reverts uncommitted changes to ensure data consistency
2. Redo (Reapply Changes) – it guarantees that committed transactions are recovered after a system crash.
3. Transaction ambiguity rules – Ensures only valid transactions are considered, managing rollback overhead.

• Transactions are guaranteed to execute reliably because of ACID properties.
• Transaction – states and models define the way transactions work.
• Concurrency control provides protection against concurrency conflicts (e.g., in a multi-user environment).
• Ensure atomicity of transactions even in case of system failures through failure recovery mechanisms.

Transaction handling is a critical aspect of any modern Database Management System (DBMS), ensuring that operations within a database are secure, efficient, and free from errors.

10.4: Properties of Transactions

A transaction, in a database, is a series of operations executed as one work unit. Transactions executed must adhere to ACID properties to ensure that they have data integrity, consistency, and reliability. These Properties guarantee that, even during power failures, crashes and concurrent transactions, the database could be returned to some previous valid state. The four fundamental properties of a transaction are:

1. Atomicity – Ensures that all operations in a transaction are executed completely or not at all.
2. Consistency – Ensures a transaction takes the database from one valid state to another.
3. Isolation – Guarantees that transactions do not disturb one another.
4. Durability – Guarantees that once a transaction has been committed, it will remain so, regardless of what may happen.

These properties together make the ACID model, which is the basis of a reliable Database Management System (DBMS).

1. Atomicity (All or Nothing Rule)

Atomicity guarantees that a transaction is treated as a single, indivisible unit. So, either all the operations of the transaction are performed successfully or none of them are performed at all. Any failure of any of these operations must trigger a rollback of the entire transaction to avoid partial updates.

Why is Atomicity Important?

In the absence of atomicity, a transaction may leave data half-completed in the database, creating corrupted and inconsistent data.

Example of Atomicity

Consider a bank transfer where Alice transfers $500 to Bob. The transaction consists of:

1. Deduct $500 from Alice's account
2. Add $500 to Bob's account

SQL Example:

```
START TRANSACTION;
UPDATE Accounts SET Balance =  Balance + 500 WHERE
Account_ID = 1; -- Add money
UPDATE Accounts SET  Balance = Balance - 50 WHERE
Account_ID = 1; -- Withdraw money
COMMIT;   -- Commit changes
```

If the second operation fails (say due to a database crash), atomicity guarantees the first operation will be undone by rolling back the transaction:

```
ROLLBACK;    -- undo everything
```

Effect: It is either fully committed or fully rolled back so no partial transfer.

2. Consistency (Maintaining Database Validity)

Consistency means that a transaction is valid with respect to any database constraint before and after running. The database must meet all conditions, rules, and relationships.

Why is Consistency Important?

To prevent creating corrupt or invalid data that breaks business rules and constraints through transactions, consistency is important.

Example of Consistency

Consider an e-commerce system where a customer places an order:

1. Deduct stock quantity from inventory
2. Generate an invoice for the order

The order should not be processed if the stock is not available, so the database should be consistent.

SQL Example (Consistency in Order Placing):

START TRANSACTION;

This SQL query deducts one stock for a product with Product_ID of 101, if there is stock available, Sequelize would be Genetrating a query similar to the one below.

INSERT INTO Orders (Order_ID, Product_ID, Customer_ID)
VALUES (5001, 101, 2001);
COMMIT;

If the stock quantity is zero, the transaction fails and does not place the order, maintaining consistency.

The consequence is that the database is always in a valid state during and after the transaction.

3. Isolation (Ensuring Independent Execution of Transactions)

Definition:

Isolation is what makes sure that when transactions are being processed, they do so without stepping on one another. Changes made by a transaction may not be visible to other transactions until the transaction is committed.

Why is Isolation Important?

Without isolation, concurrent transactions can lead to issues such as:

- Lost updates – One transaction overwrites changes made by another.
- Dirty reads – A transaction reads uncommitted data from another transaction.
- Non-repeatable reads – A transaction sees different results for the same query due to another transaction's modifications.

Example of Isolation

Consider two customers trying to book the last available flight seat at the same time:

1. Customer A initiates booking.
2. Customer B initiates booking at the same time.

If the database does not implement isolation, both customers can be assigned to the same seat, which will cause a conflic.

SQL Example (Using Isolation to Prevent Booking Conflicts):

SQL Example (Utilising Isolation to Avoid Overbooking)---

START TRANSACTION;
SELECT Seats_Available FROM Flights WHERE Flight_ID = 301
FOR UPDATE; // Locks the row
UPDATE Flights SET Seats_Available = Seats_Available - 1
WHERE Flight_ID = 301
COMMIT;

- The seat availability is locked due to the FOR UPDATE statement until that transaction is done.
- Then no other user will be able to access the seat until the transaction is committed.

Outcome: A single customer gets the seat, no conflicts.

4. Durability (Permanent Data Storage After Transaction Completion)

Definition:

Durability: Once some transaction has been committed, the updates made by that transaction should be permanent.

Why is Durability Important?

In the absence of durability, there could be a potential loss of committed transactions in the event of a power outage or a system crash/abrupt shutdown, resulting in data loss.

How is Durability Ensured?

• Write-Ahead Logging (WAL): This mechanism ensures that the database can recover after a crash by writing transaction logs before applying any changes.

• Commit Operation: Changes are available in persistent storage (disk, SSD, or cloud storage) after they are committed.

Example of Durability

Let us consider a customer who placed an order online, for instance:

1. Fill stock inventory by reducing stock quantity
2. An order with 'Confirmed' status

Persist the order once it is confirmed, the order should be stored permanently, even if the system crashes.

SQL Example (Ensuring Durability in Order Confirmation):

START TRANSACTION;

UPDATE Products SET Stock = Stock - 1 WHERE Product_ID = 102;

INSERT INTO Orders (Order ID, Product ID, Customer ID, Status) VALUES (6001, 102, 3001, 'Confirmed')INSERT INTO Orders (Order_ID, Product_ID, Customer_ID, Status) VALUES (6001, 102, 3001, 'Confirmed');

COMMIT;

• If the system fails after the COMMIT statement, the order will still be confirmed (when the system restarts).

• Committed changes are sustainable, even after failures, through database logging.

OUTPUT: the order is stored permanently (Durability).

These four properties, Atomicity the A, Consistency the C, Isolation the I, and Durability the D, ensure the reliability, integrity, and consistency of the database.

ACID Property	Ensures That...	Example
Atomicity	A transaction is fully completed or fully rolled back	Money transfer: Debit and credit both succeed or both fail
Consistency	The database remains valid before and after transactions	Preventing orders if stock is unavailable
Isolation	Transactions do not interfere with each other	Preventing two customers from booking the same flight seat
Durability	Committed transactions remain permanent	Orders stay confirmed even after a system crash

Databases ensure that applications like business applications, financial systems, e-commerce platforms, etc., work correctly without errors or inconsistencies.

10.5: Transaction isolation, Schedules: Serial, Non-Serial Schedules

1. Transaction Isolation in Databases

Transaction isolation prevents interference from concurrent transactions, preserving the database's consistency and integrity. Multiple versions of a row is a core concept in concurrency control, which avoids issues like dirty reads, lost updates, and inconsistent reading. A multi-user database allows multiple transactions to overlap in time. Promiscuous interaction may lead to corrupt, out-of-sync, or missing data. Isolation guarantees the correct outcome of each transaction as if it executed in isolation.

Example of Transaction Isolation

Consider two customers booking the last available train ticket simultaneously:

1. Transaction A queries availability and sees 1 seat available.
2. Transaction B, which checks availability at the same time, also sees 1 seat left.

Notes

3. Both the transactions book the seat.
4. Now they have two customers with the same seat and, hence, a conflict.

Transaction isolation mechanisms prevent both incorrect updates and ensure the consistency of data to avoid this situation.

2. Isolation Levels in Database Systems

The level of isolation between one transaction and other concurrent transactions is defined by different isolation levels. The more isolation you have, the more accurate (but slower) you will be, the less isolation, the faster (but potentially inaccurate) you will be.

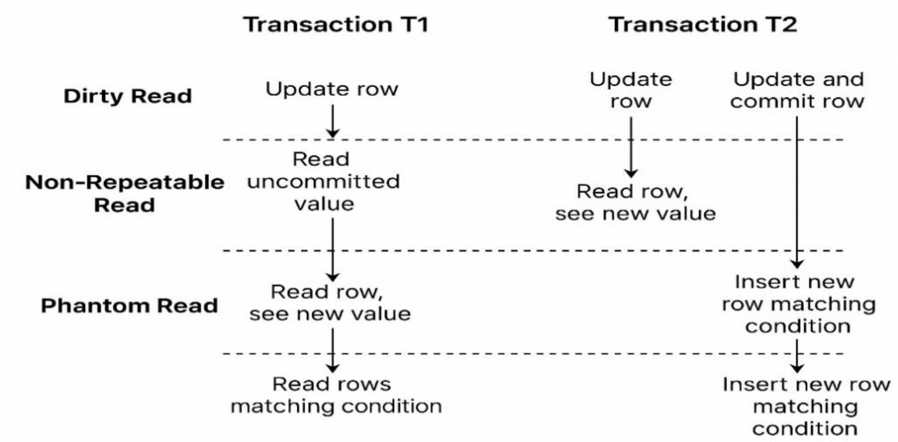

Fig. 10.3: Diagram illustrating transaction anomalies

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read	Use Case
Read Uncommitted	Allowed	Allowed	Allowed	Fastest, but least safe
Read Committed	Prevented	Allowed	Allowed	Standard for many databases
Repeatable Read	Prevented	Prevented	Allowed	Ensures consistent reads
Serializable	Prevented	Prevented	Prevented	Highest safety, but slowest

Common Problems in Isolation Levels

1. Dirty Read (Reading Uncommitted Data)

This happens when a transaction reads data that has been modified by yet another transaction but has not yet been committed

Example:

- Transaction A: Increase salary, not yet committed.
- Transaction B: Fetches the new salary.
- Transaction A: Rollback, reverting change.
- Transaction B: Is now corrupt and has the wrong data.

2. Non-Repeatable Read (Different Results in the Same Transaction)

When transaction reads the same row twice but gets different values due to another transaction updating it in between.

Example:

- Transaction A: Reads a product price to be $100.
- Transaction B: Changes the price to $120 and commits.
- Transaction A: Reads the price again: $120 instead of $100.

3. Phantom Read (New Rows Appearing in Subsequent Reads)

This happens when an inserted/deleted row is returned in a new read through the same transaction.

Example:

Transaction A : Get All Orders for Customer 101(5 records)

• Transaction B: Create new ORD for Customer 101 and commit.

• Transaction A: Reads again, there are 6 records now.

3. Transaction Schedules: Serial and Non-Serial Schedules

Scheduler is a way of executing multiple transaction in a database in a serial manner. This order of execution affects the consistency and correctness of the data.

Serial Schedule (Fully Isolated Transactions)

A serial schedule is one where transactions are executed one after another, with no overlaps.

Slow but certain — every transaction must wait for the previous one to complete.

Example:

Let us consider two transactions, T1 and T2:

- T1: Account balance update.
- T2: Reads account balance.

Serial Execution:

T1: Read Balance

T1: Update Balance

T1: Commit

T2: Read Updated Balance

T2: Commit

Pro: Guarantees consistency and free from concurrency issues.

X Drawback: Slow, because transactions do not overlap

Non-Serial Schedule (Concurrent Transactions)

In contrast, a non-serial schedule permits transactions to run concurrently, interleaving their operations.

Works great and boost performance but might create inconsistency.

Example:

T1: Read Balance

T2: Read Balance

T1: Update Balance

T2: Update Balance

T1: Commit

T2: Commit

Problem: Lost updates—T2 reading before T1 commits will overwrite T1's changes.

4. Types of Non-Serial Schedules

However, not all non-serial schedules are of concern. Others are accurate but perform worse.

Conflict Serializable Schedule

- Concurrent execution of transactions but final result matches a serial execution.
- Correctness is preserved and permits parallel execution.

• Utilized for optimistic concurrency control.

Example:

T1: Read Balance

T1: Update Balance

T2: Read Balance

T2: Update Balance

T1: Commit

T2: Commit

Since T1 finishes before the effects of T2's changes are seen, the outcome is the same as that of a serial schedule.

View Serializable Schedule

- Transaction produces a final result as in serial execution, although operations may differ.
- Much more permissive than conflict serializability.

Example:

For example, two transactions update a price list, but their final effect is correct: the operations are reordered.

5. Ensuring Correct Schedules: Concurrency Control

Databases use the following concurrency control techniques to avoid errors in non-serial schedules:

1. Two-Phase Locking (2PL) → Set locks earlier than accessing resources and eventuates in serializability.
2. Timestamp Ordering → Each transaction is assigned a timestamp, and according to their timestamp, transactions are executed.
3. Optimistic Concurrency Control (OCC) → No lock mechanism, transactions run freely then checked before committing. Transaction isolation means transactions execute properly without interfering with each other.

- READ_COMMITTED is Isolation levels (Read Uncommitted, Read Committed, Repeatable Read, Serializable) (if we want to allow how much concurrency?)
- A schedule is the order of execution of transactions, which has an impact on consistency.
- Serial schedules are always accurate, but slow, and non-serial schedules optimize the speed, but expect for conflicts.
- Concurrency control methods maintain data integrity in non-serialized schedules.

With a sound grasp of isolation levels and transaction schedules, database administrators can achieve a good balance of performance and consistency, enabling reliable database operations in multi-user settings.

Check Your Progress:

1. Define a transaction and explain its ACID properties with examples.

----

2. Discuss the various states of a transaction during execution.

10.6: Summary

A transaction represents a logical unit of work in a database system that ensures data consistency and integrity. Each transaction follows four essential ACID properties—Atomicity, Consistency, Isolation, and Durability. Atomicity ensures all operations execute completely or not at all, while consistency maintains valid database states. Isolation prevents concurrent transactions from interfering, and durability guarantees permanent results after a commit. Transactions pass through states like active, partially committed, failed, aborted, and committed. They ensure reliability in multi-user environments, supporting rollback and recovery mechanisms when failures occur. Transaction management thus safeguards data integrity, providing a foundation for dependable database operations.

10.7: Exercises

Multiple Choice Questions:

1. Which of the following is NOT one of the ACID properties of a transaction?

a) Atomicity

b) Consistency

c) Isolation

d) Accessibility

Answer: d) Accessibility

2. The property that ensures a transaction is treated as a single, indivisible unit is called:

a) Isolation

b) Atomicity

c) Durability

d) Consistency

Answer: b) Atomicity

3. Which SQL command is used to permanently save the changes made by a transaction?

a) SAVEPOINT

b) COMMIT

c) ROLLBACK

d) BEGIN

Answer: b) COMMIT

4. If a transaction fails after making some changes, the process of undoing those changes is called:

a) Recovery

b) Rollback

c) Logging

d) Validation

Answer: b) Rollback

5. Durability in transactions ensures that:

a) The transaction can be canceled anytime

b) The effects of a committed transaction are permanently stored

c) The transaction is isolated from others

d) Transactions execute sequentially

Answer: b) The effects of a committed transaction are permanently stored

Notes

Descriptive Questions:

1. Define a transaction and explain its role in database management systems.
2. Describe the ACID properties of transactions with suitable examples.
3. Explain the different states of a transaction during its execution.
4. Discuss the significance of commit and rollback operations in transaction processing

10.8: References and Suggested Reading

- Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database System Concepts* (7th ed.). McGraw Hill.
- Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson.
- Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (3rd ed.). McGraw Hill.
- Coronel, C., & Morris, S. (2019). *Database Systems: Design, Implementation, and Management* (13th ed.). Cengage Learning.

Block 4: Transaction Management and Concurrency

Unit 11: Serializability

Structure

11.1: Introduction

Serializability is a fundamental concept in concurrency control that ensures the correctness of transactions executed simultaneously in a multi-user database environment. A schedule (sequence of operations) is said to be serializable if its result is equivalent to that of a serial execution of the same transactions.

There are two major types of serializability — Conflict Serializability and View Serializability. Conflict serializability is based on the ordering of conflicting operations (such as read and write), while view serializability considers the final output and dependencies of operations.

Serializability guarantees that even when multiple transactions run concurrently, the database remains consistent as if the transactions were executed one after another. Understanding this concept is crucial for designing systems that balance performance with correctness

11.2: Learning Outcomes

After completing this unit, students will be able to:

1. Explain the concept of serializability in transaction scheduling.
2. Differentiate between serial, non-serial, and serializable schedules.
3. Understand conflict and view serializability.
4. Analyze schedules to determine whether they are serializable.
5. Appreciate the importance of serializability in maintaining consistency in concurrent databases.

11.3 Serializability, Conflict Serializability

1. Introduction to Serializability

Millions of transaction attempts are submitted to the database concurrently every second, and a vital concept you need to be aware of is something called serializability. If the effect of a schedule (the order in which transactions operations are carried out) is equivalent to the effect of some serial schedule then it is called as serializable schedule.

Why is Serializability Important?

- In multi-user databases, several transactions may be working concurrently to enhance performance.
- Inconsistent data, lost updates, or incorrect results may happen when transactions are not adequately controlled.
- Sequentializability guarantees correctness under concurrency and prevents execution at the same time conflicting

Example: Serial vs. Non-Serial Execution

Consider two transactions, T1 and T2:

- T1: Withdraws $100 from a bank account.
- T2: Checks the balance.

Serial Execution (Correct & Safe)

T1: Read Balance ($1000)
T1: Update Balance ($900)
T1: Commit
T2: Read Balance ($900)
T2: Commit

T2 sees the correct updated balance of $900.

Non-Serial Execution (Unsafe)

T1: Read Balance ($1000)
T2: Read Balance ($1000)
T1: Update Balance ($900)
T1: Commit
T2: Commit

T2 reads an incorrect balance of $1000 instead of $900!

Serializability defines a contract to ensure that, despite running concurrently, transactions will execute such that they are equivalent to a serial execution order, preventing this type of inconsistency.

2. Types of Serializability

. Conflict Serializability

- Ensures that transactions can be reordered into a serial schedule by checking for conflicts.

- Conflict serializability is verified using a precedence graph (or dependency graph).

View Serializability

- Ensures that final results of transactions match those of a serial execution, even if operations are reordered.
- More relaxed than conflict serializability.

11.4 Conflict Serializability

What is Conflict Serializability?

What is conflict serializable: A schedule is conflict serializable if it is possible to convert it into a serial one by exchanging non-conflicting operations without modifying the final outcome.

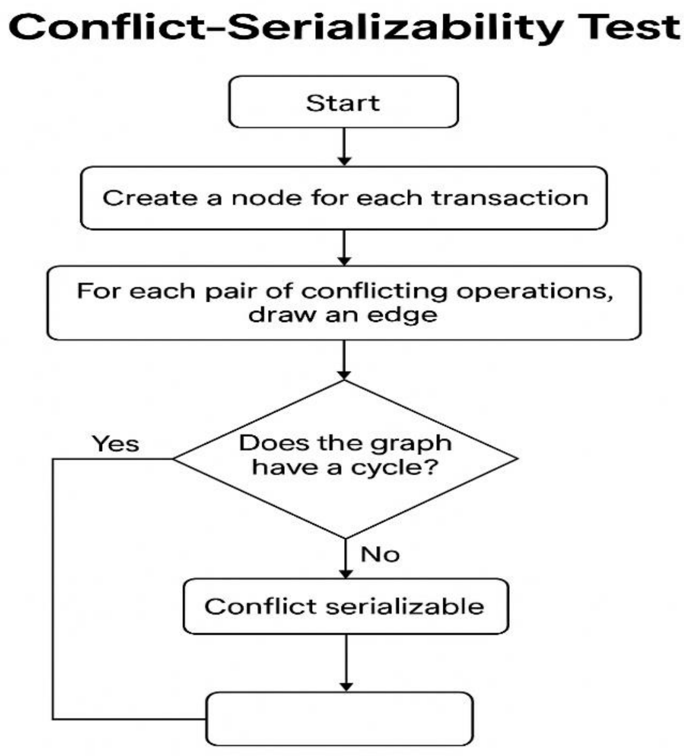

Fig. 11.1: flowchart illustrating the conflict-serializability test

What Causes a Conflict?

Two operations conflict if they:

1. Belong to different transactions (T1 and T2).
2. Operate on the same data item (e.g., X).
3. At least one of them is a WRITE operation.

Types of Conflicting Operations

Operation 1	Operation 2	Same Transaction?	Same Data Item?	At Least One WRITE?	Conflict?
Read(X)	Read(X)	No	Yes	No	No
Read(X)	Write(X)	No	Yes	Yes	Yes
Write(X)	Read(X)	No	Yes	Yes	Yes
Write(X)	Write(X)	No	Yes	Yes	Yes

Checking Conflict Serializability Using a Precedence Graph

A Precedence Graph (or Dependency Graph) is used to show if a schedule is conflict serializable.

Steps to Check Conflict Serializability:

1. Create a directed graph with transactions as nodes.
2. Add a directed edge from Ti to Tj if Ti performs an operation before Tj that conflicts.
3. Check for cycles in the graph:
    - If the graph has NO cycles, the schedule is conflict serializable.
    - If the graph has a cycle, the schedule is not conflict serializable.

4. Example of Conflict Serializability

Example 1: Conflict Serializable Schedule

Consider the following schedule:

Time	Transaction	Operation
1	T1	Read(X)
2	T2	Read(X)
3	T1	Write(X)
4	T2	Write(X)

Step 1: Identify Conflicts

- T1: Read(X) vs. T2: Read(X) → No conflict
- T1: Write(X) vs. T2: Read(X) → Conflict (T1 → T2)
- T1: Write(X) vs. T2: Write(X) → Conflict (T1 → T2)

Step 2: Build Precedence Graph

T1 → T2

- No cycle exists → The schedule is conflict serializable.

Step 3: Equivalent Serial Schedule

The transactions can be executed in the order T1 → T2.
The schedule is conflict serializable (and equivalent to the serial execution of T1 followed by T2).

Example 2: Non-Conflict Serializable Schedule

Consider this schedule:

Time	Transaction	Operation
1	T1	Read(X)
2	T2	Write(X)
3	T1	Write(X)

Step 1: Identify Conflicts

- T1: Read(X) vs. T2: Write(X) → Conflict (T1 → T2)
- T2: Write(X) vs. T1: Write(X) → Conflict (T2 → T1)

Step 2: Build Precedence Graph

T1 → T2

T2 → T1 (Cycle detected)

- A cycle exists → The schedule is not conflict serializable.

The schedule is not conflict serializable because T1 and T2 cannot be reordered into a serial sequence.

5. Conflict Serializability vs. View Serializability

Feature	**Conflict Serializability**	**View Serializability**
Definition	Transactions can be reordered into a serial schedule using conflict rules	Transactions produce the same final result as a serial execution
Check Method	Precedence Graph (Check for cycles)	Compare final results
More Restrictive?	Yes (Stronger condition)	No (More relaxed)
Practical Use	Most databases enforce conflict serializability	View serializability is rarely used

• Serializability verifies correct result of concurrent transaction is equivalent to that of serial execution.

Notes

- The conflict serializability is the most popular method that is used to ensure the safe concurrent executions.
- Conflict serializability of a schedule can be tested through Precedence Graphs.
- If a schedule has a cycle, it is NOT conflict serializable.
- Conflict serializability is stricter than view serializability, but simpler to implement.

Conflict serializability is a key concept in database management systems that ensures transactions are executed in a manner that preserves the desired properties of the database.

Check Your Progress:

1. Explain the concept of serializability and its role in ensuring correctness in concurrent transaction execution.

----

2. Differentiate between conflict and view serializability with examples.

11.5: Summary

Serializability ensures that the outcome of concurrent transaction execution is equivalent to a serial (non-overlapping) schedule. It is a key criterion for maintaining database consistency under concurrency. Two major types of serializability—conflict and view—are used to evaluate transaction schedules. Conflict serializability depends on the order of conflicting operations, while view serializability is based on the final outcome of transactions. Techniques like precedence graphs help detect whether a schedule is serializable. Serializability prevents problems like lost updates, temporary inconsistencies, and cascading rollbacks. It provides a theoretical basis for concurrency control mechanisms in DBMS. Ensuring serializability allows multiple transactions to execute efficiently without compromising the correctness of results.

11.6: Exercises

Multiple Choice Questions:

1. A schedule is serializable if:

a) It produces the same result as a serial schedule

b) It executes transactions in parallel

c) It violates ACID properties

d) It allows partial rollback only

Answer: a) It produces the same result as a serial schedule

2. Conflict serializability is based on:

a) Order of conflicting operations

b) Final output of transactions

c) Time of transaction start

d) Priority of users

Answer: a) Order of conflicting operations

3. Two operations conflict if they:

a) Belong to the same transaction

b) Access the same data item, and at least one is a write operation

c) Are executed at different times

d) Are both read operations

Answer: b) Access the same data item, and at least one is a write operation

4. Which of the following serializability types ensures equivalence based on final results?

a) Conflict Serializability

b) View Serializability

c) Logical Serializability

d) Recoverable Serializability

Answer: b) View Serializability

5. Which of the following is NOT a characteristic of serial schedules?

a) Transactions execute one after another

b) No interleaving of operations occurs

c) Provides maximum concurrency

d) Ensures consistency

Answer: c) Provides maximum concurrency

Descriptive Questions:

1. Define serializability. Why is it important in database systems?
2. Explain the difference between conflict serializability and view serializability.
3. Describe how dependency graphs are used to test for conflict serializability.
4. Compare serial, non-serial, and serializable schedules with examples.

11.7: References and Suggested Reading

- Date, C. J. (2004). *An Introduction to Database Systems* (8th ed.). Pearson Education.
- Ullman, J. D. (1988). *Principles of Database and Knowledge-Base Systems* (Vol. 1). Computer Science Press.
- Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database System Concepts* (7th ed.). McGraw Hill.
- Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (3rd ed.). McGraw Hill.

Block 4: Transaction Management and Concurrency

Unit 12: Concurrency Control & Deadlock Handling

Structure

12.1: Introduction

In multi-user database environments, multiple transactions often execute simultaneously, which can lead to data inconsistencies if not properly managed. **Concurrency control** is the mechanism used to ensure that transactions execute in a safe and consistent manner without interfering with one another.

Common concurrency control techniques include **Lock-Based Protocols**, **Timestamp Ordering**, and **Optimistic Concurrency Control**. These methods help maintain database consistency by preventing problems such as **lost updates**, **dirty reads**, and **unrepeatable reads**.

However, concurrency can sometimes lead to **deadlocks**, where two or more transactions wait indefinitely for each other to release resources. Effective **deadlock detection**, **prevention**, and **recovery** strategies are therefore essential to maintain system stability and performance.

12.2: Learning Outcomes

After completing this unit, students will be able to:

1. Explain the concept and need for concurrency control in databases.
2. Identify problems arising from concurrent transactions.
3. Understand and apply different concurrency control techniques.
4. Define deadlock and discuss methods for its detection, prevention, and recovery.

5. Appreciate the importance of balancing concurrency and consistency for optimal performance.

12.3: Concurrency Control

1. Introduction to Concurrency Control

Concurrency control refers to the methods used by a DBMS to ensure the correct operation of simultaneous transactions. It handles dirty read, lost update, and inconsistency problems when multiple users are trying to access the database at the same time.

Why is Concurrency Control Important?

In a multi-user database system, multiple transactions may execute concurrently, leading to potential conflicts. Concurrency control ensures that:

 Data integrity is maintained despite concurrent operations.

 ACID properties (Atomicity, Consistency, Isolation, Durability) are preserved.

 Correct execution order of transactions is maintained.

 Performance and throughput are optimized without sacrificing correctness.

Example Without Concurrency Control (Lost Update Problem)

Consider two transactions, T1 and T2, updating the same data item (bank balance = $1000):

Without Concurrency Control:

T1: Read Balance ($1000)

T2: Read Balance ($1000)

T1: Update Balance to ($900)

T2: Update Balance to ($950)

T1: Commit

T2: Commit

Final Balance = $950 instead of $900 (T1's update is lost).

With Concurrency Control:

T1: Read Balance ($1000)

T1: Update Balance ($900)

T1: Commit

T2: Read Balance ($900)

T2: Update Balance ($950)

T2: Commit

Final Balance = $950 (Correct result achieved).

2. Problems Due to Lack of Concurrency Control

Problem	Description	Example
Dirty Read	A transaction reads uncommitted changes made by another transaction.	T1 updates salary, T2 reads new salary before T1 commits, but T1 rolls back. T2 now has incorrect data.
Lost Update	One transaction overwrites another transaction's changes.	T1 and T2 read the same balance, T1 updates it, then T2 updates it, ignoring T1's change.
Non-Repeatable Read	A transaction reads the same row twice but gets different values due to another transaction's update.	T1 reads product price, T2 updates the price, T1 reads again and gets a different value.
Phantom Read	A transaction reads a set of rows, but another transaction inserts/deletes rows in between.	T1 counts total employees, T2 inserts a new employee, T1 re-executes and gets a different count.

3. Concurrency Control Techniques

To prevent the above problems, DBMSs implement concurrency control mechanisms that ensure correct transaction execution. The most widely used techniques are:

1. Lock-Based Protocols (Pessimistic Concurrency Control)
2. Timestamp-Based Protocols
3. Optimistic Concurrency Control (OCC)
4. Multiversion Concurrency Control (MVCC)

4. Lock-Based Concurrency Control (Using Locks)

What are Locks?

Locks mechanisms that prevent concurrent access to the same data by multiple transactions. Locks guarantee that a transaction has to relinquish a lock before another transaction can use the data item.

Types of Locks

Lock Type	Purpose	Example

Shared Lock (S-Lock)	Allows multiple transactions to read but not write.	T1 and T2 both read the same row at the same time.
Exclusive Lock (X-Lock)	Allows only one transaction to read and write at a time.	T1 updates a row, preventing T2 from accessing it.

Two-Phase Locking (2PL) Protocol

The Two-Phase Locking (2PL) protocol ensures conflict serializability by dividing transactions into two phases:

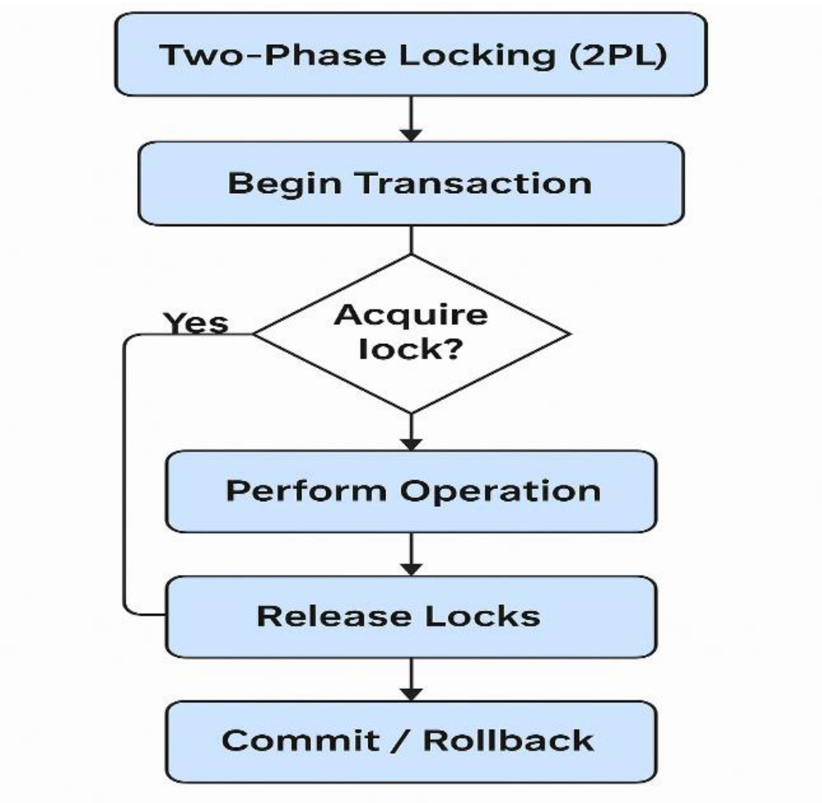

Fig.12.1: Flowchart of the Two-Phase Locking (2PL) protocol

1. Growing Phase:
    - A transaction acquires locks but does not release any locks.
2. Shrinking Phase:
    - A transaction releases locks but does not acquire any new locks.

Advantage: Ensures serializability.

Disadvantage: Can lead to deadlocks (two transactions waiting indefinitely for each other's locks).

Example of Two-Phase Locking

T1: Lock(X)

T1: Read(X)

T1: Lock(Y)

T1: Read(Y)

T1: Unlock(X)

T1: Update(Y)

T1: Unlock(Y)

Ensures correct execution by preventing lost updates and dirty reads.

5. Timestamp-Based Concurrency Control

What is Timestamp Ordering?

- Every transaction is assigned a unique timestamp when it starts.
- Transactions execute in order of their timestamps.

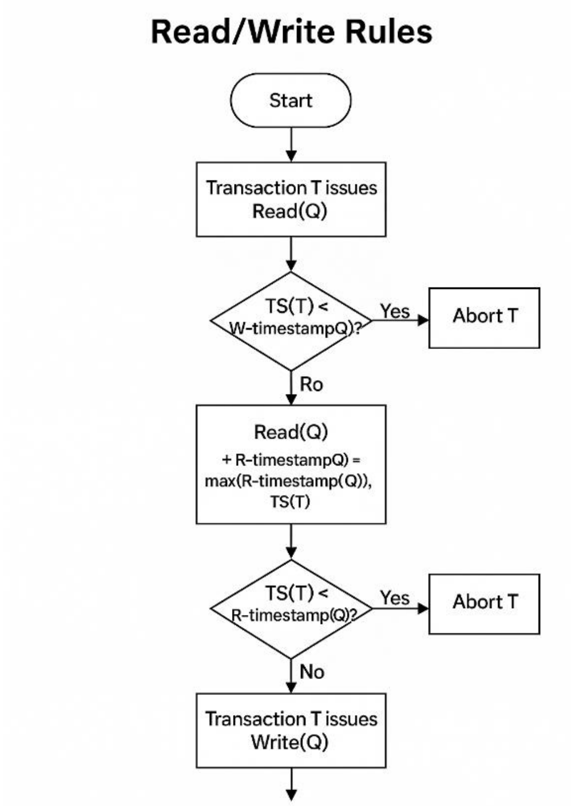

Figure.12.2: Concurrency Control

How Timestamp-Based Concurrency Control Works

Each data item has:

1. Read Timestamp (RTS): The largest timestamp of any transaction that has read the item.
2. Write Timestamp (WTS): The largest timestamp of any transaction that has written the item.

If a newer transaction tries to access an older version of data, it is aborted and restarted.

Notes

Advantage: Prevents deadlocks.

Disadvantage: Transactions may be aborted frequently, reducing performance.

6. Optimistic Concurrency Control (OCC)

What is OCC?

- OCC assumes transactions rarely conflict and allows them to execute freely.
- Before commit, the system checks if conflicts occurred.
- If a conflict is found, the transaction is aborted and restarted.

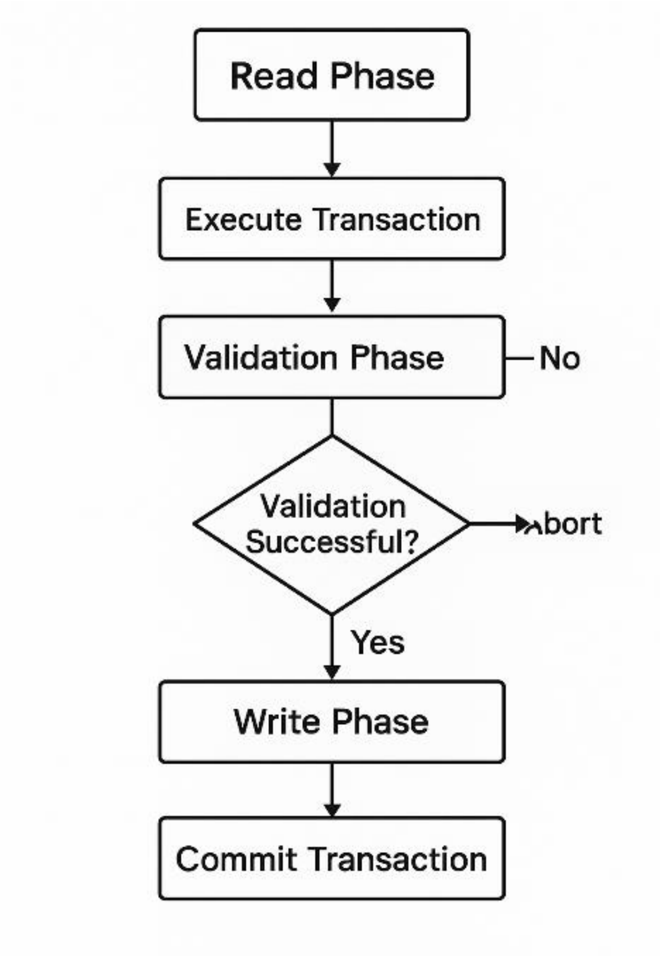

Figure.12.3: Flowchart of Optimistic Concurrency Control process

Phases in OCC

1. Read Phase: Transaction reads data without locking.
2. Validation Phase: Before committing, checks if another transaction modified the data.
3. Write Phase: If no conflict, changes are written to the database.

Advantage: Faster in systems with low conflicts.

Disadvantage: Rollback may happen frequently in high-concurrency environments.

Notes

7. Multiversion Concurrency Control (MVCC)

What is MVCC?

- MVCC stores multiple versions of data instead of locking it.
- Each transaction gets a consistent snapshot of the database at the time it starts.
- Readers don't block writers, and writers don't block readers.

How MVCC Works:

1. Read transactions get a snapshot of old data (ensuring consistent reads).
2. Write transactions create a new version of the data instead of modifying the old one.
3. Older versions are removed when no transactions need them.

Advantage: Eliminates locking overhead and increases performance. **Disadvantage**: Uses more storage because multiple versions of data are kept.

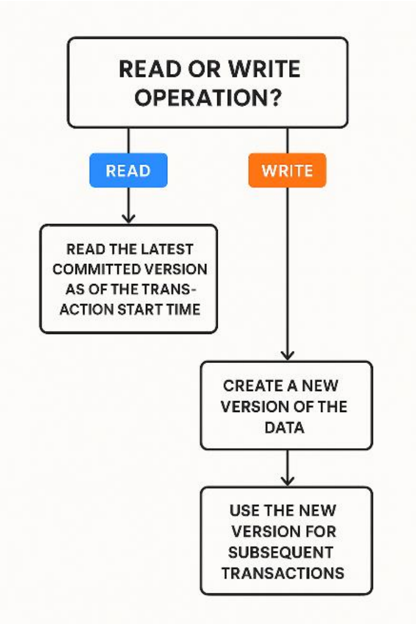

Figure.12.4: flowchart showing how MVCC handles reads and writes

8. Deadlock Handling in Concurrency Control

Deadlock happens when two or more transactions are keeping each other waiting indefinitely for each other to release locks The two most common ones are.

Deadlock Prevention Strategies:

1. Timeout: If a transaction waits too long, it is aborted.
2. Wait-Die Scheme: Older transactions wait; younger transactions restart.
3. Wound-Wait Scheme: Older transactions force younger ones to restart.

Notes

Deadlock handling ensures transactions do not block indefinitely. Concurrency control is necessary to ensure the correct, consistent, and efficient execution of transactions in a multi-user database.

Technique	Advantage	Disadvantage
Lock-Based Protocols (2PL)	Prevents lost updates & dirty reads	Can cause deadlocks
Timestamp Ordering	Ensures transactions execute in correct order	May abort transactions frequently
Optimistic Concurrency Control (OCC)	Best for low-conflict environments	Rollbacks may be frequent in high concurrency
MVCC	Improves performance (no blocking)	Uses more storage

Features of databases: [/heading] Through effective concurrency control, databases maintain a balance between consistency, isolation and performance, ensuring that multiple users can work on them simultaneously, without corrupting data.

12.4 Concurrency Control Protocols: Lock based and Timestamp based

Concurrency Control Protocols: Lock-Based and Timestamp-Based

1. Introduction to Concurrency Control Protocols

In Database Management Systems (DBMS), concurrency control is the process of managing simultaneous operations without conflicting with each other. They guarantee that the operations execute correctly and comply with isolation, consistency, and serializability.

Why Are Concurrency Control Protocols Needed?

In multi-user databases, several transactions execute simultaneously to enhance performance. Without adequate concurrency management, read anomalies return, which include dirty reads, lost updates, and inconsistent data reads.

Concurrency control protocols prevent conflicts by ensuring that transactions execute in a controlled manner.

Types of Concurrency Control Protocols

The two most commonly used concurrency control protocols are:

1. Lock-Based Protocols – Transactions acquire locks to control data access.
2. Timestamp-Based Protocols – Transactions are ordered using timestamps to ensure serial execution.

2. Lock-Based Concurrency Control Protocols

. What Are Lock-Based Protocols?

Lock-based protocols use locks to restrict multiple transactions from accessing the same data simultaneously.

Types of Locks

Lock Type	Description	Example
Shared Lock (S-Lock)	Allows multiple transactions to read the same data but prevents writes.	Multiple users can view a bank balance at the same time.
Exclusive Lock (X-Lock)	Allows only one transaction to read and write the data.	A user transferring money should prevent others from modifying the same account.

Shared Locks allow reading but prevent writing.

Exclusive Locks prevent all access except for the locking transaction.

12.5: Two-Phase Locking (2PL) Protocol

What is 2PL?

The Two Phase Locking (2PL) protocol is one of the most common methods to achieve serializability; it does so by separating the transaction into two distinct phases:

1. Growing Phase: A transaction acquires locks but does not release any.
2. Shrinking Phase: A transaction releases locks but does not acquire any new ones.

Notes

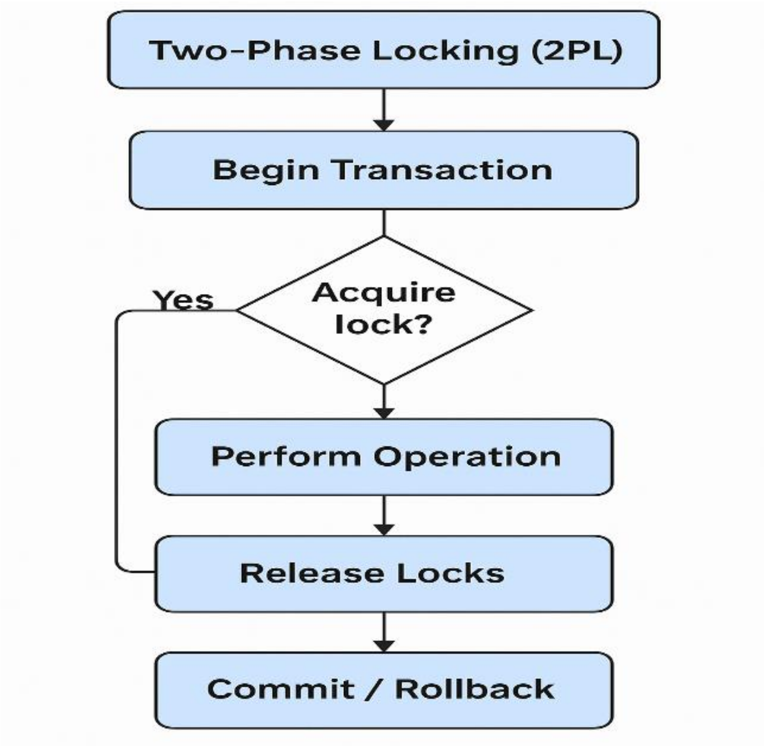

Fig.12.5: Flowchart of the Two-Phase Locking (2PL) protocol

Guarantees serializability.

Can lead to deadlocks if transactions wait indefinitely for each other's locks.

Example of Two-Phase Locking (2PL)

T1: Lock(A)

T1: Read(A)

T1: Lock(B)

T1: Read(B)

T1: Unlock(A)

T1: Write(B)

T1: Unlock(B)

Correct execution: Ensures consistent transaction execution.

Strict Two-Phase Locking (Strict 2PL)

- Locks are held until the transaction commits or aborts.
- Prevents cascading rollbacks (when an aborted transaction forces multiple rollbacks).
  Safer than basic 2PL because transactions only release locks after committing.

Deadlock and Starvation in Lock-Based Protocols

Problem	Description	Solution

Deadlock	Two or more transactions wait indefinitely for each other's locks.	Timeouts, Wait-Die, Wound-Wait schemes
Starvation	A transaction never gets a lock because other transactions always get priority.	Fair scheduling policies

Deadlocks occur when transactions form a circular wait.
Starvation happens when low-priority transactions never execute.

3. Timestamp-Based Concurrency Control Protocols

What Are Timestamp-Based Protocols?

Timestamp-based protocols order transactions based on their timestamps to ensure serializability.

How It Works

- Each transaction T is assigned a unique timestamp (TS) when it starts.
- Each data item has:
    1. Read Timestamp (RTS): Latest timestamp of a transaction that read the data.
    2. Write Timestamp (WTS): Latest timestamp of a transaction that wrote to the data.

Ensures that older transactions execute before newer ones.

Basic Timestamp Ordering Protocol

- If a transaction T wants to read X:
    - If TS(T) < WTS(X) → T is aborted (because a newer transaction already updated X).
    - Else, T reads X, and RTS(X) is updated.
- If a transaction T wants to write X:
    - If TS(T) < RTS(X) or WTS(X) → T is aborted (because older reads or writes exist).
    - Else, T writes X, and WTS(X) is updated.

Prevents dirty reads and lost updates.
Transactions may be aborted frequently, reducing performance.

12.6 Thomas's Write Rule (Optimized Timestamp Protocol)

- If TS(T) < WTS(X), ignore the write instead of aborting T.
  Reduces unnecessary transaction rollbacks.

Comparison: Lock-Based vs. Timestamp-Based Protocols

Feature	Lock-Based Protocols	Timestamp-Based Protocols
How It Works	Uses locks to control access	Uses timestamps to order transactions
Handling Concurrency	Prevents conflicts by locking resources	Allows transactions to execute but aborts if conflicts occur
Risk of Deadlock?	Yes	No
Risk of Starvation?	Yes	Yes (Frequent rollbacks)
Performance	Slower due to locks	Faster but can lead to frequent restarts
Best Used For	Systems with high contention (e.g., banking, ticketing)	Systems with high read-to-write ratio (e.g., analytics, reporting)

Lock-based protocols prevent conflicts but can cause deadlocks.
Timestamp-based protocols avoid deadlocks but may require frequent transaction rollbacks.

Concurrency control protocols are used for the correct execution of transactions in multi-user databases.

- Lock-based methods (2PL, Strict 2PL) avoid conflict and deadlock issues.
- Timestamp-based protocols (Basic Timestamp Ordering, Thomas's Write Rule) make serializability guarantee without deadlocks but may result in frequent rollbacks.
- Selecting appropriate protocols will be aligned with specific performance requirements as per transaction type.

Database systems are then able to efficiently leverage efficient concurrency control protocols to strike a balance between isolation, consistency, and performance, as multiple transactions are able to execute concurrently and safely.

12.7: Deadlock Handling: Detection and Prevention

This leads to a situation of circular dependency, in which processes cannot continue execution, and thus, parts of the system come to a halt. Deadlocks are one of the hardest problems in operating systems, database management systems, and distributed computing environments. Therefore, it is critical to understand, identify and resolve deadlocks because they may cause a noticeable drop in system performance, useless resource consumption or even system deadlocks that require manual restart of the system. This guide will explore the key principles behind deadlocks, the conditions that result in them, how they can be detected, prevented, avoided, and recovered from. We will also link to practical implementations in different contexts of computing, analyze the trade-offs of proposed solutions, and explore research directions for the evolution of deadlock avoidance/avoidance in modern paradigms of computing.

Fundamental Concepts of Deadlocks

Deadlock is a particular state in concurrent programming in which processes are forever blocked in their wait for resources, so it is a condition where, without outside intervention, the system enters a state it cannot recover from. The resource allocation systems where deadlocks happen need to be understood to fully comprehend this phenomenon. In these systems, processes request resources, use them to calculate and then release them to other processes. Resources can either be preemptable (can be taken away from a process) or non-preemptable (the holding process must explicitly release it). Deadlocks are mainly because of non-preemptable resources because pre-emptable resources can hardly lead to deadlock conditions. Resource Allocation Graphs The Resource allocation graph is another data structure that visually depicts resource allocation and requests in a system. In this directed graph, we have processes and resources as nodes, and the edges are allocated resources or requests. Haven't heard of deadlock detection? This description should give a better idea about the format of resource allocation graph and how swill be interpreted.

The Four Necessary Conditions for Deadlock

The Coffman conditionsE. G. Coffman: A look at Deadlock are four conditions that must hold for a deadlock to occur, they form the basis of understanding a deadlock situation and E. G. Coffman formalized this concept. The first condition for mutual exclusion states that at least one resource should be held in a non-sharable mode such that only one process can be using it at any specific interval. Deadlocks could never occur if every resource in the system could be shared among all processes at the same time. The other condition, hold and wait (or resource holding) arises when a process that is holding at least one

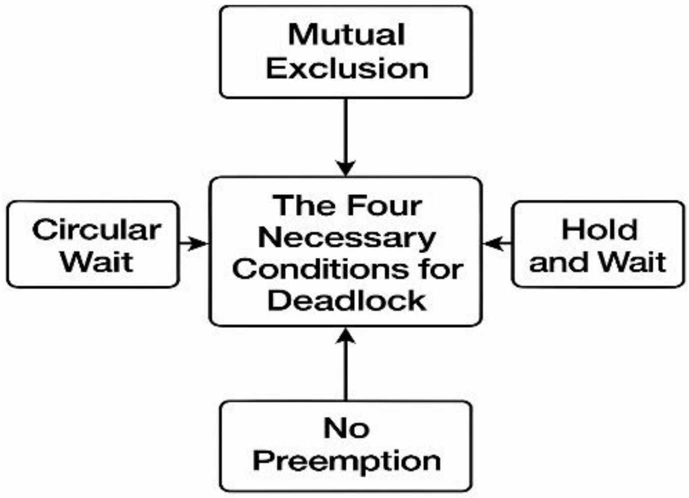

Figure.12.6: Deadlock Conditions

resource is waiting to attain additional resources that are held by other processes. This provides a scenario where processes will wait for other processes to release resources while already holding resources, thereby potentially paving the way for circular dependencies. The third condition, no preemption, says that resources cannot be forcibly removed from a process; the process that has the resource must explicitly give it up. The system could preempt resources to prevent deadlocks by reallocating them from a waiting process. The fourth condition, circular wait, occurs when there is a set of processes such that every process is waiting for a resource held by another process in the set, forming a circle of processes. A deadlock can occur when all four of the following conditions hold simultaneously. Alternatively, if all of these conditions are precluded the system can avoid deadlock completely. These insights lay the groundwork for a class of deadlock prevention schemes all of which attempt to eliminate one of the four conditions that are needed to allow deadlocks to occur within the system.

Resource Allocation Graphs and Deadlock Representation

A visual model to explain optimal resource allocation in based on resource allocation graphs (RAG) in a powerful way. A representational element of a resource allocation graph contains two kinds of nodes (circles, process nodes and squares or rectangles, resource nodes). Directed edges link these nodes, indicating resource requests or allocations. The edge from a process to a resource indicates that the process has requested that resource, but not yet been granted it. An edge from a resource to a process means that the resource has been allocated

Notes

to that process. Abstract resources(0): In systems where there are several instances of a single resource type, the representation is complex, for example it may to have to be notated the number of instances requested or allocated. Resource allocation graphs are not so much useful for detecting deadlock: a cycle in a resource allocation graph with only one instance of each resource type means a deadlock has occurred. But cycles are an essential yet not sufficient condition for deadlocks in most resource arrangement models.

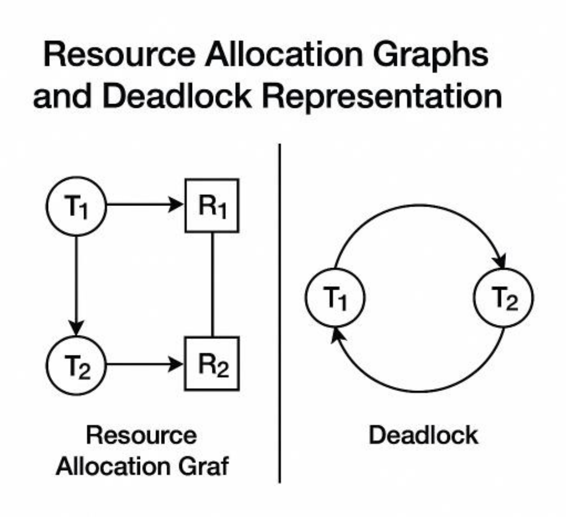

Figure.12.7: Diagram showing Resource Allocation Graphs

In such systems, special algorithms must be applied to determine whether a cycle actually represents a deadlock. Additionally, these resource allocation graphs can be dynamic since processes are able to request new allocations and release resources as needed. By tracking these changes and examining the structure of the resulting graph, systems can detect impending deadlocks before they completely manifest or can discover full deadlocks for resolution. Resource allocation graphs are especially helpful in visualizing and explaining deadlock states, making them a tool for understanding as well as education in concurrent systems.

Deadlock Detection Mechanisms

Deadlock detection describes algorithms and techniques allowing systems to detect when a deadlock has occurred. These will be required in systems where deadlock prevention or avoidance strategies are not implemented, or as a backup to fallback strategies that fail. Detection algorithms commonly check resource allocation state and process requests to search for circular wait states. If we consider a single-instance resource type, detection can be simple — it is equivalent to searching for cycles within the resource allocation graph. (N) To avoid deadlock in multi-instance resource systems, more complex methods needed, such as the banker's algorithm or derivatives thereof, exploring possible resource allocation paths to determine if safe sequences exist. Deadlock detection is done periodically or when certain events occur such as a resource is requested or allocation failed. Detections happen relatively infrequently; there is always a trade-off: with more frequent detections you get more overhead but an earlier detection and response, whereas with less frequent detections you get less overhead but potentially longer deadlocks.

As soon as a deadlock is detected, the system needs to follow recovery procedures that it has in place to break the deadlock and allow those processes involved in the deadlock to continue. Approaches such as those used by operating systems and database systems involve advanced detection methods that minimize false positives and negatives while providing timely responses that do not unduly degrade system performance.

Algorithms for Single-Instance Resource Deadlock Detection

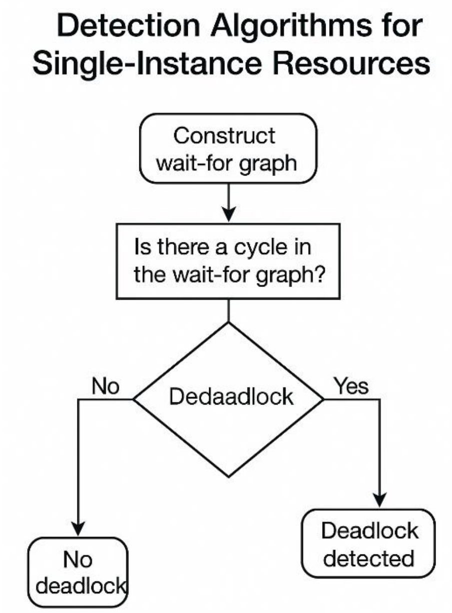

Figure.12.8: Diagram illustrating Detection Algorithms for Single-Instance Resources

Detecting deadlocks can be done with graph-based algorithms, comparatively easy, in systems in which every resource has a single instance. The typical method is to create and examine what is called a wait-for graph which is a simplified version of the resource allocation graph where the process nodes are connected directly by edges denoting wait relationships. This graph edge from process P1 to process P2 means that process P1 is waiting for a resource that is currently held by process P2. Detecting deadlocks subsequently boils down to cycle detection in this directed graph which can be done using classical graph algorithms e.g., depth-first search (DFS) or breadth-first search (BFS).The detection algorithm usually works in three steps: First, build the wait-for graph from the current resource allocation and request; Second, check if the graph has cycle either with DFS or BFS; And thirdly, if any cycle has found, Then declare a dead-lock involving the processes in the cycle. The time complexity is generally $O(n^2)$ (where n is the number of processes that need to be executed) making this approach computational efficient and suitable for normal execution, for instance in a small system with no more than 60 processes. A single-instance detection algorithm can also be applied at the resource type level instead of the add instance level, grouping similar resources together. Further optimization: we can do so less frequently, based on system activity patterns (e.g. detecting when two processes cycle back on holding resources), and further focus detection on when deadlocks are more likely, e.g. within the periods after sequences of resource requests and when processes claim to be waiting past a threshold period of time.

Detection Algorithms for Multiple-Instance Resources

We have already said that deadlocks in systems where multiple instances of resources exist are more complex than with systems with a single instance. So from what it follows: Not having cycles in a resource allocation graph no longer implies that there are no deadlocks, because it may be the case that there are other types of instances of a resource that have not yet been allocated, making at least one process to finish and freeing some resources. There are multiple algorithms developed for this purpose, of which, the most notable ones are - the banker's algorithm and deadlock detection algorithm. The multiple-instance resources deadlock detection algorithm generally checks for the possibility of some sequence of resource acquisitions enabling all the processes to execute. This means keeping data structures that keep track of: available resources (ones that are not currently allocated to any process), allocated resources (ones that are currently held by each the process), and requested resources (ones that each process is waiting to get). The algorithm then tries to find a hypothetical execution sequence, repeatedly finding processes whose resource requests can be satisfied with the current available resources. If such processes are discovered, the algorithm simulates such processes starting and releasing the resources they had, putting their allocated resources back into the pool. This is repeated until we are either out of processes (no deadlock) or we run out of eligible processes (indicating a deadlock involving the remaining processes). Since m is the number of resource types and n is the number of processes, this algorithm has $O(m \times n^2)$

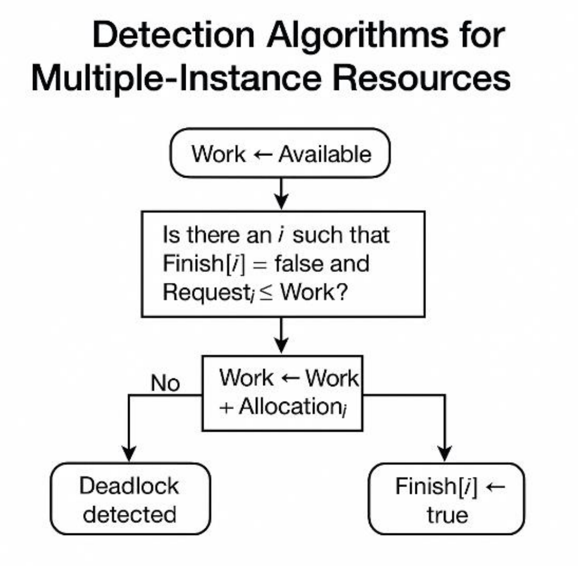

Figure.12.9: Diagram illustrating Detection Algorithms for Multiple Instance Resources

time complexity which is usually more computationally intensive than single-instance detection. Such overhead can be minimized using several optimizations like incremental detection with processes and resources their state has changed since last detection cycle or priority-based approaches that considers process that are more likely to cause deadlocks before others.

Distributed Deadlock Detection

The problem of deadlock detection in distributed systems has its own complexity that doesn't appear in centralized systems. A distributed system is one in which resources and processes are spread across multiple nodes or sites, and no single entity has complete knowledge of the global state of the system. This distributed nature makes it challenging to build an overall resource allocation graph and requires specific algorithms for effective deadlock detection. Three general types of methods have been proposed for the distributed detection of deadlocks: path-pushing, edge-chasing, and global state detection methods. Similarly, path-pushing algorithms propagate the dependencies between the processes along the paths in the wait-for graph, with the eventual goal of being able to tolerate cycles that span multiple nodes. Edge chasing algorithms employ special "probe" messages that move along the edges of the wait-for graph, which return to their originators to signal a cycle. Global state detection methods try to make a global view of the system state at all nodes and analyze the global state for deadlocks with the help of centralized algorithms. Such distributed detection algorithms face additional complexities like message delays, partial failures, false positives or false negatives owing to dynamic nature of system. Additionally, they should incur little communication overhead; even small amounts of message passing to perform deadlock detection can be detrimental to system performance. Indeed, many distributed systems operate with hierarchical strategies that integrate local detection among nodes with global coordination across nodes, thereby balancing detection accuracy and communication efficiency.

Deadlock Prevention Strategies

Deadlock prevention involves designing a system with resource allocation policies that prevent at least one of the four necessary conditions for deadlock. These strategies ensure that deadlocks are structurally impossible in the system by guaranteeing at least one condition cannot occur. Prevention strategies are conservative by nature and involve placing restrictions on the ways processes are allowed to request and hold onto resources. In most resources question of mutual exclusion prevention is very rare, however minimum number of resources should be made non-shareable by the system designers. Preventing hold and wait generally leads processes to either have to request for all resources required by them before they can proceed, or to release all of the resources they hold before they can request more. This means that these systems may need to take away resources from a process when it runs out of other options, in what is called forced reclaiming. Circular wait can typically be prevented by defining a total ordering for resource types and forcing processes to request resources in that order. Although prevention strategies offer the strongest guarantee against deadlocks, they often incur a significant cost in terms of resource utilization, system performance, and programming complexity. This composite of trade-offs is what makes prevention strategies well-suited for critical systems in which deadlocks are simply unacceptable under any conditions, but less so for the general-purpose computing environments where more well-rounded approaches may be preferred.

Eliminating Mutual Exclusion

One of the base yet difficult methods for deadlock prevention is to remove the mutual exclusion condition. This strategy is designed from the perspective of systems approach, and aims to design systems where resources can be simultaneously shared among two or more processes, hence breaking the contention that creates the basis of deadlocks. In practice, completely avoiding mutual exclusion is not possible for many types of resources that are inherently non-shareable (e.g. printers, tape drives, database locks). Yet, some strategies can mitigate this impact by designing systems such as spooling where resource-executing processes interact with processes running on virtual resources instead of the actual resources themselves. Print spooling, for instance, enables multiple processes to send data to a print job queue as opposed to needing direct access to the printer hardware. In much the same way, virtualization technologies allow multiple virtual machines to share the same physical hardware, de-stabilizing exclusive resources for shared ones at a higher level of abstraction. Another response is to redesign resources or the patterns in which they are accessed to allow concurrent usage, for instance through reader-writer locks in which multiple processes can read the data concurrently whilst still allowing exclusive access for writing. Asynchronous data structure, lock-free and wait-free -- Among the ways to decrease mutual exclusion is the development of lock-free and wait-free data structures. Although it is impossible to eliminate mutual exclusion for every type of resource, it is possible to look at some resources and determine if they can be made into shareable resources and reduce the potential deadlocks in a system.

Preventing Hold and Wait

Hold and Wait − In this condition, a process holds a resource while waiting to acquire additional ones. To prevent this condition, we need to design resource allocation policies that guarantee that processes will never concurrently possess some resources while it is waiting for others. There are two common methodologies in pursuing this end. This way all resources required by each process should be requested at the beginning of execution. This means that when a process requests resources, the system will give either all resources or nothing, in this way, it does not allow a process to hold some resources while in wait for others. Although conceptually simple, this strategy requires processes to specify all of their resource needs ahead of time, something that may not always be realistic for practical applications that can develop dynamic resource requirements. It can further cause the waste of resources since resources that are reserved at a very early point in a process lifecycle can go unused for a long time. The second approach allows processes to request resources incrementally but forces them to relinquish all currently held resources upon a denied request. Then, the process tries to grab all necessary resources at once in a next request. This is more flexible, but leads to complexities including the potential for starvation (if a process repeatedly fails to acquire all the resources it needs) and the extra cost of repeatedly releasing & reacquiring resources. Both strategies can be improved upon, such as using resource reservation in which processes inform the system in advance of their expected future demands for resources without actually requesting the resources, allowing the system to plan allocations and reduce waiting whenever possible. Furthermore, pooling of resources together can be used where similar resources are grouped, and operations are less, again reducing the chances of hold and wait condition occurring.

Allowing Resource Preemption

No-preemption: The system should be designed in such a way that resources cannot be forcibly taken away from the processes holding them, which is one of the necessary conditions for deadlocks. In the context of preemption, if a process requests a resource that it cannot yet access, the system checks whether preempting resources from other processes might help to avoid a potential deadlock. If we identify any of our resources on which a suitable candidate for preemption would be found, we can release it and grant the requesting process the resource, breaking the formation of a deadlock before it can even materialize. There are a number of approaches to making preemption work. These include process priority schemes, where higher-priority processes can preempt one or more resources from lower-priority processes. A second approach uses resource age or holding time as its criteria and preempts resources holding for a while. This is not a bad description of a one-shot preemption/cancel paradigm, checkpoint-based preemption is a better fit because a process periodically saves its execution check pointed state, allowing it to be rolled back to a sort of consistent state after preemption of its resources. This makes preemption a complex topic that requires careful design of the system implementing it. Such a system must ensure the safe aspects of the process context saving, the performance costs of saving process state when preempting processes, and starvation policies to stop processes from being repeatedly preempted in to sensibly afford a system which ensures progress in userspace. The system also needs some policies on how to choose which resources to preempt, where there are multiple candidates, e.g. the system should try to minimize the disruption to the processes, should provide fairness and let the processes make progress.) Indeed, many modern operating systems employ some forms of resource preemption, and for some resource types—especially memory, CPU time, and some I/O resources—there are practical ways to implement this process, even if it is not easy.

Avoiding Circular Wait

Out of the deadlock prevention strategies, preventing circular wait is one of the most commonly used strategies since it is easier to implement than removing other necessary conditions. The basic method is to develop a total ordering of all classes of resources and require processes to request resources in this order. This phone work eliminating circular dependencies between processes at the level structure. By never requesting resources except in order. To implement this strategy, however, the following steps are required: (1) assign a unique numerical identifier to each resource type; (2) require that processes request resources strictly in increasing (or decreasing) order of identifiers; (3) enforce this ordering in system calls or middleware that validates resource request sequences. For instance, if there are resource R1, R2, and R3 with identifiers 1, 2, and 3, a process must ask for them in the order R1, R2, R3 This avoids creating cycles within the resource allocation graph ensuring that processes can only wait on resources than have identifiers greater than those that they currently possess. While simple in principle, this technique can be difficult in practice. The processes in the system must be designed or modified to acquire the resources in the specified sequence, which may conflict with their actual operational order. It also needs to find a logical ordering of resources such that processes do not have to request resources out-of-order. By deciding hierarchies of related resources to work with can ease some of these problems, where stabilization can be at a classification level rather than a direct resource. There are dynamic resource hierarchies of resources that dynamically adjust the ordering of resource accesses based on observed usage patterns, hopefully matching the application requirements better while still avoiding circular wait conditions.

Deadlock Avoidance Algorithms

Deadlock avoidance is a halfway house between the very restrictive prevention and the more reactive detection and recovery. These algorithms adopt an approach where processes can make incremental resource requests, without taking the system to an unsafe state that might independently bring about deadlock. They work based on extra information about the resources needed for processes, usually expressed as predetermined maximum resource demands. Based on this information, the system can decide on each resource request whether it can grant it or might place the system in a potential deadlock state in the future. The banker's algorithm, one of the most popular deadlock avoidance algorithms, designed by Edsger Dijkstra, simulates a tentative allocation of resources to find out if, there is a sequence of processes that can be executed without deadlock. This solution is safe because all processes can finish even if they request their maximum remaining resources right away. A request is denied if after the allocation there is no safe sequence and the requesting process blocks until resources can be granted. The main contribution of this paper is an alternative model to the banker's algorithm, called the resource-trajectory approach, in which the sequence of resource allocations and deallocations is modeled as a trajectory through a multidimensional space, that is, the resource space, and a safe resource allocation is one that never allows the trajectory to enter unsafe regions. Avoidance algorithms offer stronger correctness guarantees than detection and recovery with much less severe restrictions than prevention strategies, but present their own challenges such as the overhead of safety checking to ensure avoidance, the need to know beforehand how much of a resource is needed, and, potentially, less than optimal use of resources due to conservative allocation policies.

The Banker's Algorithm and its Variants

Deadlock avoidance: The single most important approach to deadlock avoidance is the banker’s algorithm that was originally formulated by Edsger Dijkstra and is so named because of its analogy to banking systems. This algorithm uses a few data structures to maintain the state

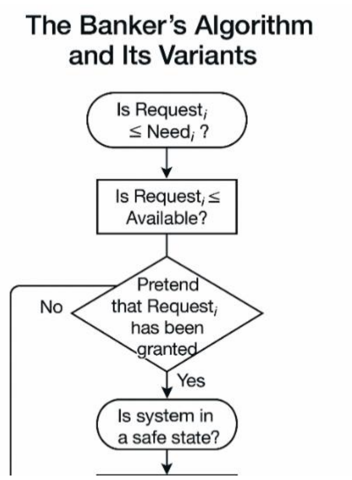

Figure:12.10: Diagram The Banker’s Algorithm and its Variants.

of resources that are allocated: the maximum resources still needed by each process — the resources currently allocated to each process — and the missing resources needed by each process. The algorithm simulates the allocation of processes' requests and looks for a "safe sequence" of process executions that would allow all processes to finish running without a deadlock. If such a sequence, does exist the state is said to be safe and the request is granted; otherwise the request is denied and the process that made the request has to wait. The banker's algorithm forms the basis of several extended and optimized variants tailored to specific system needs. We can simplify the original Handle Algorithm into its single-resource version for the case of only one resource type, which reduces the computational complexity. Instead posing the bankers algorithm to hierarchal resources system, as in tree (parent-child) hierarchy. The distributed banker algorithm works similar in nature as the banker's algorithm but it does not work with the centralized method, instead it uses a distributed matter to prevent deadlock, The process of allocation takes place in a distributed manner. There are also variants of the banker's algorithm that are dynamic, meaning they take resource requests that come up during execution of a process into consideration, thus avoiding one of the main problems of the original algorithm. Therefore, with a theoretically sound approach proposed with the banker's algorithm, the practical implementation can be difficult due to the safety need to be checked for every resource request, processes should declare their maximum needs in advance (which in many cases may be difficult to evaluate), resources may remain underutilized due to conservative allocation policies. These constraints have driven many general-purpose operating systems to prefer different strategies for deadlock management, but it provides a useful way in very specific contexts, where resource requests are measurable ahead of time and a high reliability is fundamental.

Resource Trajectory Methods

An alternative to deadlock avoidance are resource trajectory methods, which model resource allocation as a path through a multi-dimensional resource space. The axes in this model represent different types of resources, while a point in space reflects how many of those resources are currently dedicated to a process. The system moves through this space along a trajectory as processes request and release resources. Some areas of the space correspond to unsafe allocations that can cause deadlocks, and others represent safe allocations. For resource trajectory methods, the key ideas are to keep the state trajectory in a safe region.

A key aspect of this approach is identifying the critical boundaries that delineate the safe from the unsafe regions in resource space. This is when a process requests resources and the system assesses whether granting the request would cross a dangerous threshold into an unsafe space. If so the request is denied, if not the request is granted. There have been various mathematical formulations proposed for defining such critical boundaries, as well as for more efficient identifications of these boundaries. The first-run single-resource trajectory approach streamlines the analysis, applying to systems with a single resource type. The claim-and-release trajectory method utilizes knowledge of future resource releases to model a tighter safe region. The process-interaction trajectory approach focuses directly on interactions between specific processes, as opposed to the global system state, which could enable more concurrency for resource allocation. In some cases, resource trajectory approaches can be more advantageous for certain situations than the banker's algorithm, particularly potentially displaying a more accurate representation of safe and unsafe conditions, lower computational complexity for specific configurations of the system, and more intuitive visualization of safety of the system. Nonetheless, they suffer from similar limitations to other avoidance strategies (Table 2): They require prior knowledge of resource needs, and conservative allocation will underutilize resources (catch recovery too late).

Deadlock Recovery Techniques

In cases where deadlock prevention, avoidance, and detection mechanisms fail or are not applied, systems must instead rely on recovery strategies to address deadlocks once they have manifested.

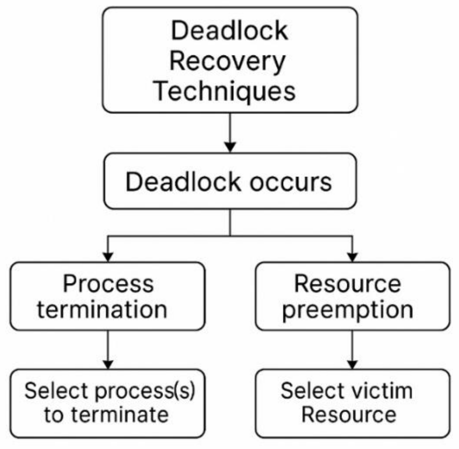

Figure.12.11: Deadlock Recovery Techniques

Deadlock recovery is where the system detects the deadlock and takes action to break it, e.g. by killing a process. Process termination techniques choose one or more competing processes in a deadlock to abort, freeing them of their held resources, and allowing potential continuation with other processes. For mustering process termination candidates, priority (w/o) process execution time, resources held and remaining work, may provide selection criteria. Resource preemption typically requires saving the state of the processes being preempted, identifying which resources to preempt, and dealing with the possible cascading effect. Since partial execution is an issue, both approaches need to cater for recovery, since aborted or preempted processes might have been state-changing and therefore need to be undone or compensated. Transaction rollback mechanisms in database systems offer a systematic way to reverse the effects of partially completed operations in the event of deadlock recovery. Current systems use hybrid recovery strategies that involve a combination of process termination and resource preemption, which chooses the appropriate strategy given a certain deadlock condition. Recovering from the deadlock allows systems to continue, but these techniques tend to have high penalties of lost work, degraded performance, and the prospect of data inconsistency, making recovery techniques sometimes a preferred strategy, but more often a strategy of last resort.

Process Termination Strategies

One of the most straightforward strategies for deadlock recovery is the termination of processes, in which one or more processes in the deadlock is/are chosen and aborted. These processes once terminated release all the resources they have if they were not previously finished, thus eliminating the circular wait condition and enabling other processes to make progress. There are several strategies for deciding which processes to kill when a deadlock is detected. Selection of victims tends to balance several elements so as to minimize the impact on the overall system. This strategy -known as the minimum disruption strategy- consists of terminate the minimum number of processes that is necessary to break the deadlock and generally reports a set of processes that, by terminating them, will release the resources needed to satisfy the needs of the remaining processes involved in the deadlock. In a cost-based approach, processes are assigned a termination cost based on dispatching priority, the amount of computation they have performed to date, resources they hold, and even the amount of work left to do. It then picks the processes that are cheapest to terminate. The resource utilization method aims at processes that hold many resources, specifically the ones that are used by several other processes, because killing them anyway unblocks more processes. Clearly, this kind of framework is quite conservative, as it terminates only the victims one-by-one and checks if the deadlock has been cleared before going after more victims: Incremental termination. For a system to achieve clean termination, things can get complex, since it needs to make sure to free up all previously allocated resources, correctly handle any shared data structures between the affected processes, inform dependent software, and potentially even hang onto some data to support restart. In systems that have transactional semantics, like databases, the termination of processes relies on the transaction undo mechanism to recover the system from operations that only partially execute, preventing the system from becoming inconsistent. Though terminating processes will resolve deadlock, this results in substantial loss of computation and the potential of user created frustration especially if the process in question is interactive. As such, its cost means it is most appropriate as a last resort in systems where other deadlock management mechanisms have been unsuccessful or are not feasible.

Resource Preemption Methods

One technique of deadlock recovery that would fall under this method is resource preemption. This provides a more fine-grained way to intervene than terminating processes (which may lose more work and be more disruptive). There are several key challenges that need to be addressed for effective resource preemption. First, it must decide which resources to preempt, most often choosing those that will end the deadlock without a significant cost. The importance of the resource, the length of time it has been retained, progress in the holding process, and how many processes could be enabled by releasing it are among possible criteria. Second, the system must have means of saving the state of processes that have their resources preempted, so that they can resume execution later when the resources are available again. Third, the system has to deal with the complications of rolling back any partially executed operations that relied on the preempted resources in order to keep the data consistent. Different types of resource preemption strategies have been developed for different computing environment. Checkpoint-based preemption utilizes process checkpointing protocols to capture the execution state before preempting resources, enabling a clean restoration when the resources are reallocated. In priority-based preemption, processes with higher importance are preferred, and each resource is preempted from lower-priority processes to meet the demands of higher-priority ones. Costminimization preemption aims to characterize the cost of preempting various resources and chooses those with the lowest aggregate system cost. Preemption is a practical construct that can be applied for some types of resources such as memory pages, CPU time slices and some locks which are eligible for a clean preemption and being less applicable for resources that cannot be restored easily like open network connections or exclusive device controls. Resource preemption is effective mainly in systems capable of adequately capturing, and restoring process state, which makes it a more feasible solution in systems that provide rich facilities for checkpoint-restore.

Handling Partial Execution and Rollback

In many systems in which deadlocks are solved by killing processes, or by preempting their resources, the system faces the problem of partially executed operations. In scenarios of deadlocks, processes involved might have previously finished some parts of their work, modifying state of system, data structures or external systems in ways which need to be handled in the process of recovery. Transaction Rollback Mechanisms Some systems, especially database systems and

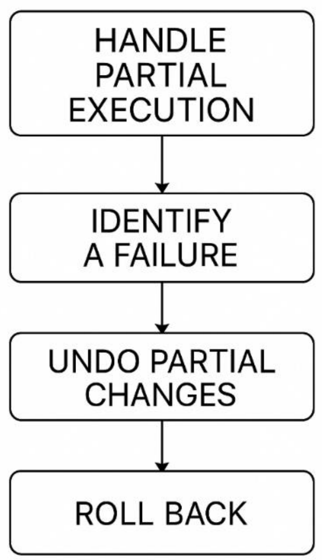

Figure.12.12: Diagram for Handling Partial Execution and Rollback

systems with transactional semantics, implement transaction rollback mechanisms that logically associate a series of operations with a transaction. These systems log enough information about commands to be able to cancel them, usually using write-ahead logging, shadow paging or journaling techniques. Since, during a deadlock recovery action, when a process is terminated or preempted, the transaction associated with it would need to be rolled back and thus restoring the system to a consistent state, as if the transaction had not started at all. In systems not supporting a fully-fledged transaction interface, compensating actions can be needed to cancel the effects of partial operations. These might be application-specific cleanup procedures, restoring modified data to their original values, releasing resources consumed and notifying dependent services of the failed operation. Checkpointing is another solution to the partial-execution problem, and involves processes periodically storing their state such that it is possible to return to that point (but note that checkpointing solutions often only deal with the data space of the processes). It can shrink the amount of work lost during deadlock recovery as well as improve deadlock recovery cleanliness relative to a crash/restart of the whole process. Some systems use speculative execution in systems where they allow an operation to proceed on an optimistic basis, but preserve enough information so they can back out the changes if there is contention for those changes, or a dead-lock occurs. The overall cost of deadlock recovery is heavily affected by the handling of partially executed actions. Rollback capabilities: Well extolled systems can roll back more gracefully from deadlocks, and poorly–designed ones may create data inconsistency, resource leakes or other side effects that must be cleaned up manually — which itself may lead to cascading failures.

Practical Implementations in Operating Systems

Deadlocks can be handled in many different ways by many different operating systems (OS), and some OS don't even bother trying to prevent a deadlock. Unix-like systems like Linux tend to be minimalist in nature, relying on timeouts and human rescues instead of robust deadlock prevention or detection tools. Most of these systems use closet timeout-based resolution for some resource types and provide administrative tooling to help identify and resolve deadlocks manually. In Windows operating systems, deadlocks are handled in a more structured approach, especially for synchronization objects like mutexes or semaphores, including wait chains traversal to detect cycles of dependency. Strict deadlock prevention is commonly introduced in real-time operating systems as they are time critical and this is typically done using priority inheritance protocols and resource reservation to eliminate priority inversion/deadlock conditions. Custom deadlock handling strategies tailored to specific hardware and application domains may be implemented by specialized embedded operating systems. Internal deadlock prevention mechanisms for services that are critical to operating system kernels typically use hierarchical locking, lock-free algorithms or careful ordering of resource acquisition. There are some specific problems for deadlock handling at the file system level, where contemporary designs employ things like delayed allocation, intent logging and non-blocking algorithms to limit deadlock risk. It is the responsibility of low-level memory management subsystems — via paging, virtual memory and the like — to prevent deadlock by treating physical memory as a preemptable resource. As you learn the approaches to implement this rather theoretical topic, you get to understand what happens at system design perspective when you are forced to go with a solution that constructs a trade-off between theory and logistics.

Unix and Linux Approaches

As for Linux and other Unix-like operating systems, historically they use a relatively minimalist policy when it comes to deadlock detection and resolution compared to more elaborate policies explored in theory. They provide a time out and other features from careful system design and user level intervention, rather than avoiding, detecting or recovering from deadlock through complex system level mechanisms. Another way of saying this, and one that is very Unix systems-like, is to "give me a lever and a place to stand" — offer hooks and ways to do things instead of trying to synthesize the thing you want right out of the core of the OS; this also fits with other Unix design principles: offer mechanisms, not policies, minimize overhead for common we-do-an-operating-and-a-some users operations, and — where it is at all possible — push complexity out into user space. Unix systems generally implement prevention strategies for specific classes of internal resources at the kernel level through careful lock ordering and acquisition protocols. Kernel synchronization primitives such as mutexes, semaphores and reader-writer locks are generally designed for deadlock prevention, using hierarchical locking schemas or lock dependency checkers to ensure consistent acquisition order. Unix systems also provide timeouts on many resource acquisition operations for user-level processes, allowing processes to detect when they are waiting too long for resources and to initiate appropriate recovery. Signal mechanisms allow blocked system calls to be interrupted, enabling applications to perform their own timeout-based recovery strategies. Resource limits and quotas ensure no single process can monopolise the system resources in such a way as to create a widespread deadlock. Linux itself has also built on top of this, introducing additional deadlock features including pthread mutexes that detect deadlocks, a kernel lock validator called "lockdep" that seeks to guarantee that no deadlocks can occur, and process monitors to determine which processes might be competing for shared resources. The watchdog facility in systemd is a promising feature since modern Linux distributions ship with it, and if it detects that an application is hung, it tries to restart it which can also bring the system out of deadlock by terminating and restarting affected processes. The practical concurrency model of Unix, with its ad-hoc approach to deadlock, captures both the challenge of implementing full deadlock detection and recovery in a general-purpose operating system and the Unix ethos to give application developers freedom — and responsibility — to design suitable deadlock strategies for their own use cases

Windows Operating System Deadlock Management

Differences in deadlock handling – The Windows OS uses a slightly more structured approach to deadlock management than any of the Unix-like OS systems, particularly with regards to synchronization objects and system resources. Similar to e.g. POSIX, Windows has a rich set of synchronization primitives (all with built-in timeout-based acquisition support), so that applications do not have to block indefinitely waiting for resources. Operating systems include timeout parameters as part of their wait functions, allowing processes to specify how long they will wait for a resource and thus provide a mechanism to detect and recover from potential deadlock situations.

Fig.4.3.11: Diagram for Windows Operating System Deadlock Management

Windows has a sophisticated wait chain transverser, capable of detecting circular dependencies in threads waiting on synchronization objects. This functionality is also exposed through programmatic interfaces and administrative tool such as Resource Monitor, allowing developers and system administrators to identify deadlocks involving Windows synchronization primitives. Windows applies internal deadlock prevention mechanisms for critical system resources: Kernel code is written to acquire locks in a consistent order and to follow hierarchical access patterns. The Windows memory management and process scheduling subsystems use resource reservation and preemption techniques to decisively limit the potential for system-wide resource deadlocks and to ensure that deadlocks can never take the entire system down. They offer a complementary technique to the timeout-based resource acquisition within an application, making the application capable of achieving graceful degradation thanks to structured exception handling implemented by the OS. Deadlock detection and resolution capabilities for distributed transactions across multiple resource managers are built into Windows via the Microsoft Distributed Transaction Coordinator (MS DTC), which integrates with database applications. With features like fair share CPU scheduling and resource metering, Windows Server editions provide additional resource governance to prevent resource monopolization that could cause deadlocks. Windows does not enforce global deadlock avoidance algorithms like banker's algorithm but its strategy of limiting resource access to short time frames, traversing wait chains, and providing administrative tools offer the system a practical approach to deadlock management that balances performance overhead with system reliability requirements.

Real-Time Operating Systems (RTOS)

The subject of this article is deadlock handling in real-time operating systems. Contrarily, in an RTOS environment, deadlocks can severely disrupt system functionality and directly encroach upon time constraints, causing disaster scenarios in critical applications (aerospace systems, medical devices, automotive control units, etc.). Thus, RTOS implementations tend to use stricter deadlock prevention techniques compared to general-purpose operating systems. One of the fundamental deadlock prevention mechanisms implemented in many RTOS is priority inheritance protocols that prevent priority inversion problems. The dynamic priority of a process holding a resource, therefore it would be adjusted to be equal to the highest priority of any process waiting for a resource. The priority ceiling protocol generalizes this idea by associating with every resource its priority ceiling (the highest priority of any process that may request the resource), and temporarily raising the priority of the process that successfully acquires the resource to its ceiling. Another common feature in RTOS environments is deterministic resource allocation policies where resources are allocated in fixed predictable patterns as opposed to dynamic decisions that could potentially lead to deadlock. No dynamic resource allocation means that the system is more rigid, with all resources being assigned to processes when the system is created, and thus many forms of deadlock are avoided at the expense of flexibility. Another RTOS paradigm is time-bounded resource acquisition; that is, every resource acquisition must finish within a fixed time limit, and timeout-based failure recovery mechanisms ensure that processes do not stall indefinitely. Commercial implementations such as VxWorks, QNX and FreeRTOS have these mechanisms as well as other specialized features such as deterministic scheduling, memory protection, and fault isolation to preserve system integrity in the event that part of the system fails. Due to the influence of time constraints on real-time systems, the potential consequence of uncontrolled deadlock may justify the overhead and increased complexity incurred by more extensive deadlock prevention, making them an interesting avenue for practical deadlock prevention and handling evaluation.

Check Your Progress:

1. Explain the need for concurrency control in databases and describe common control techniques.

----

2. Define deadlock and explain methods for its detection, prevention, and recovery.

12.8: Summary

Concurrency control allows multiple transactions to execute simultaneously without causing data conflicts or inconsistencies. Common issues such as lost updates, dirty reads, and unrepeatable reads arise when transactions interfere. Lock-based protocols, timestamp ordering, and optimistic control methods manage concurrent access effectively. However, concurrency may lead to deadlocks, where two or more transactions wait indefinitely for each other’s resources. Deadlock handling involves prevention (by resource ordering), detection (using wait-for graphs), and recovery (by aborting or rolling back transactions). Efficient concurrency control balances data consistency and system performance. It ensures that databases remain reliable and responsive even under heavy multi-user workloads.

12.9: Exercises

Multiple Choice Questions:

1. **Which of the following problems does concurrency control aim to prevent?**

a) Data redundancy

b) Lost updates and inconsistent reads

c) Transaction rollbacks

d) Index corruption

Answer: b) Lost updates and inconsistent reads

2. **In a lock-based protocol, which type of lock allows both reading and writing?**

a) Shared lock

b) Exclusive lock

c) Read-only lock

d) Binary lock

Answer: b) Exclusive lock

3. **A deadlock occurs when:**

a) A transaction terminates unexpectedly

b) Two or more transactions wait indefinitely for each other's resources

c) The database crashes during execution

d) A transaction violates ACID properties

Answer: b) Two or more transactions wait indefinitely for each other's resources

4. **Which of the following techniques assigns a unique time value to each transaction?**

a) Lock-based protocol

b) Timestamp ordering

c) Two-phase locking

d) Deadlock prevention

Answer: b) Timestamp ordering

5. **One common method for deadlock prevention is:**

a) Ignoring all deadlocks

b) Using the "wait-die" or "wound-wait" scheme

c) Disabling concurrency

d) Restarting all transactions periodically

Answer: b) Using the "wait-die" or "wound-wait" scheme

Descriptive Questions:

1. Define concurrency control and explain its importance in transaction management.
2. Discuss the problems that can occur in concurrent transaction execution.
3. Explain different concurrency control techniques with examples.
4. Define deadlock and describe strategies for its detection, prevention, and recovery.

12.10: References and Suggested Reading

- Bernstein, P. A., Hadzilacos, V., & Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.
- Silberschatz, A., Korth, H. F., & Sudarshan, S. (2020). *Database System Concepts* (7th ed.). McGraw Hill.
- Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson.
- Ramakrishnan, R., & Gehrke, J. (2003). *Database Management Systems* (3rd ed.). McGraw Hill.

Block 5: Object Oriented Database

Unit 13: Limitations of RDBMS and Introduction to Advanced Databases

Structure

13.1: Introduction

Relational Database Management Systems (RDBMS) have been the foundation of data management for decades, offering structured storage, integrity constraints, and strong consistency. However, with the growth of complex applications, unstructured data, and big data processing, traditional RDBMSs face several limitations.

RDBMSs are not efficient in handling complex data types such as multimedia, spatial, and hierarchical data. They also struggle with scalability, schema rigidity, and performance issues when managing semi-structured or unstructured information. To overcome these challenges, new types of databases have emerged—collectively known as Advanced Databases—which include Object-Oriented Databases (OODBMS), Object-Relational Databases (ORDBMS), NoSQL Databases, and Distributed Databases.

These systems extend the capabilities of traditional relational models to support complex applications in fields such as multimedia systems, scientific research, and cloud-based environments.

13.2: Learning Outcomes

After completing this unit, students will be able to:

1. Identify and explain the major limitations of traditional RDBMS.
2. Describe the need for advanced database systems in modern applications.
3. Differentiate among object-oriented, object-relational, and NoSQL databases.

4. Understand how advanced databases handle complex and unstructured data.
5. Recognize real-world applications where advanced databases are preferred.

13.3: Limitations of RDBMS

For several decades, structured data storage and retrieval have relied on Relational Database Management Systems (RDBMS). They provide many advantages like ACID (Atomicity, Consistency, Isolation, Durability) compatibility, SQL Sizes, and data integrity properties. However, despite widespread adoption and capability, there are a few downsides to RDBMSS. With advances in technology, new problems have arisen that highlight the limitations of traditional relational databases. Some of these constraints affect the performance, scalability, flexibility, and usability especially in the modern applications where massive data. These limitations are important for database architects, developers, and organizations to consider when making data management strategies. Scalability is one of the major limitations of RDBMS. Traditional relational databases were built for vertical scaling, which refers to increasing the strength of a single server by provisioning additional CPU, memory, or storage. This technique works great for moderate workloads but becomes very expensive and illogical as data volume starts to increase exponentially. Since RDBMS relies on strict table structure and complex joins to access data organized into related tables, horizontal scaling—i.e. spreading data over many machines—is fundamentally difficult. As a result, distributed systems and NoSQL databases gained notoriety as an alternative, since they can scale out well across clusters of low-cost hardware. RDBMS solutions, on the other hand, need extensive architectural changes like sharding and partitioning to scale to this level, leading to added complexity and maintenance costs. Another critical problem of RDBMS that pinpoints its ineffectiveness especially in high-throughput environments is performance. Since the Regular databases query execution involves joins, indexes, transactions, etc. Although indexes can enhance read performance, they can also increase the time taken for write operations because of the overhead of maintaining multiple indexes. Query performance over a growing dataset can degrade considerably, giving longer response times and less effective operation. Furthermore, real-time data processing

Notes

requirements are challenging for RDBMSs, which are designed for transactional consistency instead of speed and live analytics. RDBMS usually are not able to provide the low-latency requirements of applications like recommendation engines, financial trading engines, and IoT applications. On the other hand, NoSQL is focused on optimizing performance for certain use cases like key-value stores for fast lookups, or columnar databases for analytical workloads. This is another place where RDBMS is lacking, flexibility. RDBMSs enforce strict schema, which means that you need to define the structure of tables (columns and their types) before you store any data. While this rigidity guarantees well-defined measures of data consistency and integrity, it can be a massive limitation in the use case of changing data requirements. Changing an existing schema may be a painful process and need downtime and long data migrations. Another very important aspect is being able to adapt to changing business requirements. Designed to cater to the needs of big data and cloud storage, NoSQL databases are schema-less, which provides developers with the ability to store unstructured or semi-structured content without needing to define a schema beforehand.

RDBMS have another biggest disadvantage that it cannot store unstructured data in enough wide scale. Common applications in modern systems generate a variety of data types: text, images, videos, logs, sensors. RDBMSs are optimized for structured data with well-defined relationships, making storing and processing such data inefficient in a relational database. Although some relational databases offer a special field type called Binary Large Objects (or BLOBs) to store unstructured data, querying and accessing them can be slow and resource-consuming. Key features: NoSQL databases, including document stores and graph databases, are structured to handle unstructured and semi-structured information more rapidly, ideal for applications such as content management systems (CMS), big data analytics, or a machine learning workload. One more major drawback of RDBMS is the difficulty in managing relationships, and keeping data consistent. Though the performance is good, maintaining data integrity which is very important in transactional applications is achieved with relational databases utilizing foreign keys and normalization techniques. As the database grows and the behind-the-scenes maintenance of these relationships can become complex, which can cause performance bottlenecks. Joins are a necessary part of why relational databases are so powerful, but they can be computationally heavy, especially at scale. As a result, queries that perform multiple joins can become slow and inefficient, affecting application performance. On the other hand, graph databases and NoSQL databses are able to accommodate highly connected data, making them useful in scenarios like social networks, recommendation engines, and fraud detection systems. One more major limitation of RDBMS is its high maintenance and admin cost. Maintaining a relational database effectively takes an understanding of database design, indexing techniques, query optimization, and performance mitigation — and, honestly, this is a full-time job in itself. This increases operational costs and SQL database administrators (DBAs) have a vital role in keeping the database performing smoothly. Database management systems, or RDBMS, require thoughtful planning and execution of backup and recovery, replication, and security management. Scaling an RDBMS solution across many nodes multiplies these administrative challenges and necessitates Replication / Sharding, challenges that have their own

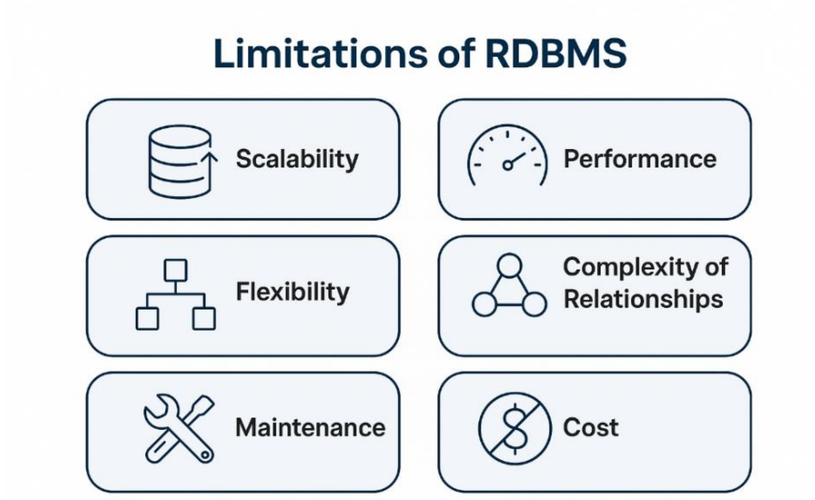

Figure: 13.1- Limitation of RDBMS

complexity. On the other hand, spherical NoSQL databases listen to availability and scalability by automating the process of scaling with auto-scaling, automated replication and replication recovery mechanisms.

RDBMS also has challenges with concurrency control and transaction management. Although the ACID properties guarantee the integrity of the data, they can also come with performance overhead, especially in situations involving high transaction concurrency. By using locking mechanisms to maintain consistency, contention issues can occur since several transactions try to grab the same resources, resulting in bottlenecks and reduced throughput. This is especially problematic in

Notes

distributed systems where one may have multiple nodes, maintaining strong consistency can lead to higher latency. NoSQL databases typically follow an eventual consistency model, trading off full ACID compliance for better performance and scalability. Not all applications are suited for this, but it certainly provides many benefits for anything with high availability and fault tolerance requirements. Another important aspect that needs to be taken care of by organizations is the cost of implementing and maintaining an RDBMS. Commercial relational database solutions like Oracle, Microsoft SQL Server, and IBM Db2 are costly due to hefty licensing fees; this proves to be too expensive for small and medium-sized businesses. Even open-source alternatives come with a hefty investment in infrastructure, expertise, and ongoing maintenance, with potential pitfalls similar to those of DynamoDB for scaling products. Moreover, scaling an RDBMS solution becomes more and more costly, as data volume increases, as it requires a high-performance hardware, storage, and networking resources in the most sense. On the other hand NoSQL databases are generally more economical because of their distributed architectures which let organizations take advantage of commodity hardware and cloud-based resources to scale effectively. RDBMSs also have limitations in terms of security and compliance. However, relational databases offer solid security features, such as authentication, authorization, and encryption, but configuring and managing them is not a trivial task. Ensuring compliance with industry regulations like GDPR, HIPAA, and PCI-DSS is necessitating stringent access controls, audit logging, and data encryption mechanisms. In an RDBMS environment, enforcing compliance can be difficult because distributed architectures are increasingly common. And for big data analysis, traditional relational databases are more prone to SQL injection, which is an attack when attackers can manipulate poorly designed queries. NoSQL databases are not without their own security risks, but can offer alternative security models to address specific threats. To sum up, even though RDBMSs are an initial basic part of big data management, their scopes have revealed themselves due to the new data paradigms. All the issues mentioned in terms of scalability, performance, flexibility, unstructured data handling, administrative complexity, concurrency control, cost, and security have created more need for alternative database solutions. As a result, NoSQL databases, cloud-based storage solutions, and distributed data architectures have developed as valid options and provide more scalability, performance, and flexibility for modern applications. So, you know you must consider their particular use cases and needs when deciding whether an RDBMS has the best fit, or some of alternative database technologies offers the best solution for the organization. Knowing these constraints allows businesses to make informed decisions to streamline their data management strategies, helping them cope better with their evolving data requirements.

Check Your Progress:

1.Discuss the major limitations of traditional Relational Database Management Systems (RDBMS) and explain how these limitations led to the development of advanced database systems.

----

2. Describe the role and features of advanced database models such as Object-Oriented Databases and NoSQL Databases in addressing modern data challenges.

13.4: Summary

Traditional Relational Database Management Systems (RDBMS) have been highly successful in managing structured data using well-defined schemas and tables. However, with the increasing demand for handling unstructured, semi-structured, and multimedia data, their limitations have become evident. RDBMSs struggle with complex data types, scalability, and flexibility in schema management. To address these challenges, advanced database systems such as Object-Oriented Databases (OODBMS), Object-Relational Databases (ORDBMS), and NoSQL Databases have evolved. These systems extend traditional models to accommodate diverse data formats and complex relationships. Object-Oriented Databases integrate programming concepts with data storage, while NoSQL databases support distributed and large-scale data management. Advanced databases also improve performance, enable horizontal scalability, and offer better support for big data and real-time applications. Their adaptability makes them suitable for emerging domains like social media analytics, IoT, and cloud computing. Thus, the evolution from RDBMS to advanced databases represents a paradigm shift towards more flexible, powerful, and application-oriented data management systems.

13.5: Exercises

Multiple Choice Questions:

1. Which of the following is a limitation of traditional RDBMS?

a) Support for ACID properties

b) Difficulty in handling unstructured data

c) Strong data consistency

d) Efficient data modeling

Answer: b) Difficulty in handling unstructured data

2. Which type of database extends relational systems by integrating object-oriented features?

a) Distributed Database

b) Object-Relational Database

c) Hierarchical Database

d) Network Database

Answer: b) Object-Relational Database

3. Which of the following is an example of an advanced database system?

a) MySQL

b) Oracle RDBMS

c) MongoDB

d) MS Access

Answer: c) MongoDB

4. Which of the following applications requires handling complex data types like images and videos?

a) Payroll system

b) Multimedia database

c) Library management

d) Billing system

Answer: b) Multimedia database

5. The main reason for the evolution of advanced databases is:

a) Simpler data models

b) Handling complex, large-scale, and unstructured data

c) Reducing data redundancy only

d) Decreasing hardware cost

Answer: b) Handling complex, large-scale, and unstructured data

Descriptive Questions:

1. What are the major limitations of traditional RDBMS? Explain with examples.
2. Discuss the need for advanced database systems in modern data applications.
3. Differentiate between RDBMS, OODBMS, and ORDBMS.
4. Explain how advanced databases manage complex and unstructured data efficiently.

13.6: References and Suggested Reading

- Özsu, M. T., & Valduriez, P. (2020). *Principles of Distributed Database Systems* (4th ed.). Springer.
- Coronel, C., & Morris, S. (2019). *Database Systems: Design, Implementation, and Management* (13th ed.). Cengage Learning.
- Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson Education.
- Hellerstein, J. M., & Stonebraker, M. (2005). *Readings in Database Systems* (4th ed.). MIT Press.

Block 5: Object Oriented Database

Unit 14: Limitations of RDBMS and Introduction to Advanced Databases

Structure

14.1: Introduction

Relational Database Management Systems (RDBMS) have been the foundation of data management for decades, offering structured storage, integrity constraints, and strong consistency. However, with the growth of complex applications, unstructured data, and big data processing, traditional RDBMSs face several limitations.

RDBMSs are not efficient in handling complex data types such as multimedia, spatial, and hierarchical data. They also struggle with scalability, schema rigidity, and performance issues when managing semi-structured or unstructured information. To overcome these challenges, new types of databases have emerged—collectively known as Advanced Databases—which include Object-Oriented Databases (OODBMS), Object-Relational Databases (ORDBMS), NoSQL Databases, and Distributed Databases.

These systems extend the capabilities of traditional relational models to support complex applications in fields such as multimedia systems, scientific research, and cloud-based environments.

14.2: Learning Outcomes

After completing this unit, students will be able to:

1. Explain the need for integrating object-oriented concepts into relational databases.
2. Define and describe key object-oriented features such as encapsulation, inheritance, and polymorphism.

3. Understand how ORDBMSs support complex data types and user-defined objects.
4. Illustrate how object-oriented features improve database modeling and reusability.
5. Compare traditional RDBMS and ORDBMS with suitable examples.

14.3: Introduction: OODBMS and ORDBMS

OODBMS and ORDBMS

Database management systems have made great strides in recent years and two of the most notable evolutions beyond RDBMS are Object-Oriented Database Management Systems and Object-relational Database Management Systems. The RDBMS was not without its limitations, particularly when it came to dealing with complex data structures, multimedia applications, and systems that required a tight coupling between the object-oriented programming language used for application development and the underlying database technology.

Object-Oriented Database Management System (OODBMS)

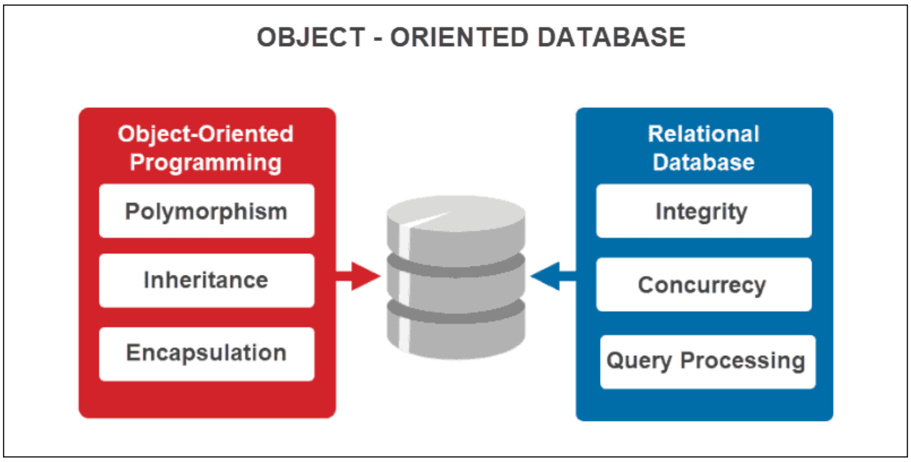

Figure: 14.1: Object Oriented Database

An Object-Oriented Database Management System (OODBMS) is a database management system that supports the modeling and creation of data as objects. This is in contrast to RDBMS architecture where data is structured in the form of rows and columns in tables, OODBMS stores data in the form of objects in a similar way just like data is represented in object-oriented programming languages like Java, C++, and Python. It provides a better way of dealing with complex data like images, audio-visual data, and nested structures, as they support object-oriented features. Suitable for CAD, multimedia database, real-time system, and AI applications. OODBMS supports inheritance,

Notes

encapsulation and polymorphism, allowing developers to directly work with objects without being forced to convert them to relational tables. This eliminates the O/R Mapping (which is needed when using RDBMS over any OO language). Yet OODBMS is not as widely used as RDBMS because of compatibility issues, a lack of standards, and a steep learning curve for many developers who are used to working in traditional relational models.

Object-Relational Database Management System (ORDBMS)

ORDBMS (Object Relational Database Management System): It is a combination of RDBMS and OODBMS. It preserves the traditional SQL and ACID (Atomicity, Consistency, Isolation, Durability) aspects of relational databases, while also allowing for the use of object-oriented techniques, such as user-defined types (UDTs), inheritance, and complex data types.

ORDBMS provides the ability to store and manipulate complex objects such as array, multimedia, geographical data, and application-defined data types without having to convert these into normal relational types. This makes well suited for applications such as geographic information systems (GIS), data warehousing, and scientific computing. Widely used ORDBMS systems include PostgreSQL, Oracle and IBM Db2 that provide object-oriented features while maintaining the performance and familiarity of SQL-like relational databases.

While both OODBMS and which are O/R DBMS exist to handle for all those complex data that the regular RDBMS simply do not work for. OODBMS would be best suited where there is a need to integrate a lot with an object-oriented programming environment, while ORDBMS can be used as a middle ground between the relational and object-oriented paradigms, which is useful for enterprises looking to extend their existing relational databases. Organizations can determine the best database system for their needs by examining the complexity and scalability of their data and the purpose of their application through these database models.

14.4 Storing and Accessing Objects in a Relational Database

Relational database management systems (RDBMS) follow the table structure, making it look difficult to store and access objects since generally the objects are directly used in object-oriented programming.

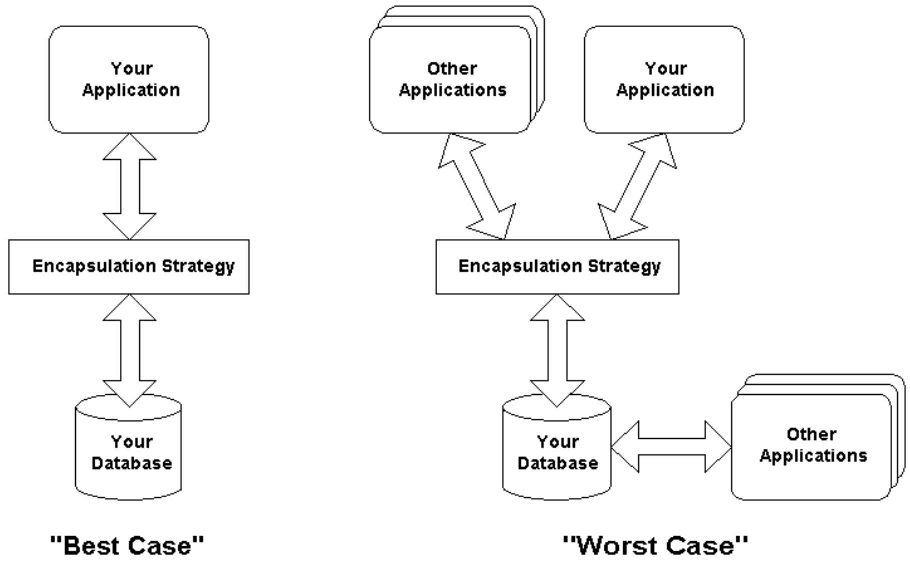

Figure: 14.2: Storing and Accessing Objects in a Relational Database

Nonetheless, as object-oriented programming languages like Java, Python and C++ became more widely used, the definition of an RDBMS changed, as most modern RDBMSs now support object storage and retrieval in one way or the other. Object-Relational Mapping (ORM), serialization, and structured storage techniques are commonly used for storing and accessing objects in a relational database.

1. Object-Relational Mapping (ORM)

Object-Relational Mapping (ORM) is a common technique that maps objects to their corresponding records in relational databases. ORM tools (Object Relational Mapping tools) are used to create mapping between objects in programming languages and relational database tables.

• In ORM-based approaches, each class of the object-oriented language corresponds to a table in the database, and each instance of that class corresponds to a row in that table.

• ORM Libraries: Hibernate (java), SQLAlchemy (python), Entity Framework (. In.NET, Hibernate (Java), and Django ORM (Python), the conversion between objects and database records is handled automatically.

• By avoiding manual SQL query writing, this method increases productivity and lowers the risk of SQL injection attacks.

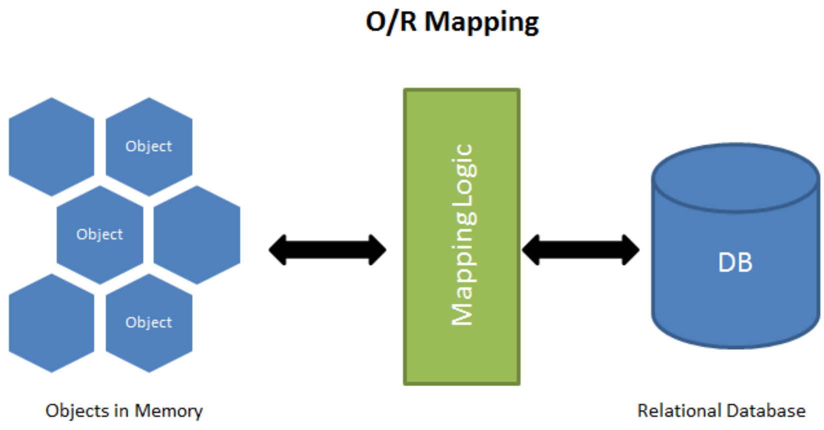

Fig. 14.3 O/R Mapping

• While ORMs provide significant development benefits, they also come with a performance trade-off related to the necessity for query translation, affecting the efficiency of complex queries and large scale data operations.

2. Storing Objects as Serialized Data

Alternative ways of persisting objects in a relational database can be achieved through serialization, which involves transforming the objects into a format that can be persisted in a database column and reconstructed when pulled from the database.

- Serialization format: JSON, XML, YAML, or binary formats (e.g., Protocol Buffers, Avro)
- Usually the serialized object is stored in a BLOB (Binary Large Object) or TEXT field in the database.
- JSON and XML formats enable semi-structured storage and simplify retrieval using built-in database functions like PostgreSQL's JSONB type or MySQL's JSON functions.
- Even though serialization provides flexible storage, the serialized data is not efficient to query since relational databases are optimized for structured tabular data not embedded hierarchical structures.

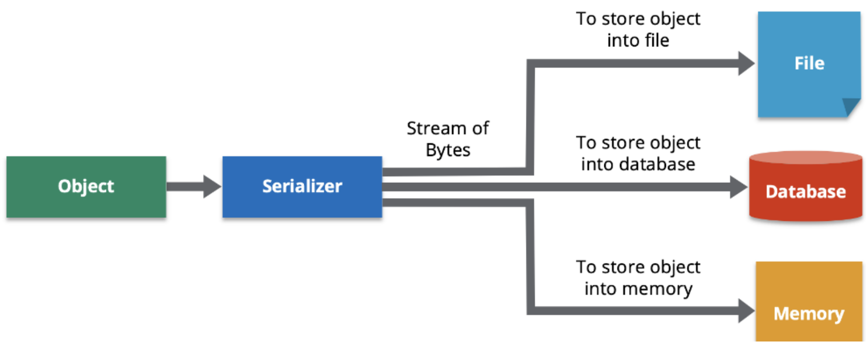

Figure: 14.4: Storing Objects as Serialized Data

3. Storing Objects in Relational Tables

Objects can be persisted through a normalized relational structure for performance, storage or data integrity. In this method:

- Normalized — these are complex objects that are broken into 2+ relational tables and result in foreign key relationships.
- Objects are kept associated through primary or foreign keys, ensuring referential integrity.
- For example, an embedded Address object in an object Person will have a separate Address table with a foreign key reference from Address to Person.
- Using this approach allows for fast querying and consistency, though retrieving the object will require JOIN operations to restore it.

4. Using Object-Relational Features in ORDBMS

Some Object-Relational Database Management Systems (ORDBMS), such as PostgreSQL, Oracle, and IBM Db2, offer built-in support for storing objects with object-oriented features like:

- User-Defined Data Types (UDTs): Allow defining custom data structures in the database.
- Nested Tables and Arrays: Support for multi-valued attributes within relational tables.
- Inheritance: Enables table hierarchies similar to object-oriented class inheritance.
- Table Functions: Allow querying objects as structured entities instead of flat tables.

These features allow for more natural object storage while maintaining the advantages of relational databases, such as data consistency and ACID compliance.

Accessing Stored Objects in a Relational Database

Once objects are stored, they must be accessed efficiently for retrieval and manipulation. Common methods include:

1. **Using SQL Queries:**
   - Standard SQL queries (SELECT, JOIN, WHERE) are used to retrieve object-related data from multiple tables.
   - Indexed queries improve performance when retrieving objects with complex relationships.
2. **ORM Query Methods:**
   - ORM frameworks provide high-level query abstractions such as find(), filter(), or get() methods to fetch objects without writing SQL manually.

- Example using SQLAlchemy in Python:

```
person = session.query(Person).filter_by(id=1).first()
print(person.name)
```

3. **Deserialization for Stored Objects:**
   - Serialized objects stored as JSON/XML/BLOB need to be deserialized before being used in the application.
   - Example of JSON deserialization in Python:

```
import json
data = json.loads(json_string)
print(data["name"])
```

4. **Querying JSON/XML Fields in Modern RDBMS:**
   - Databases like PostgreSQL and MySQL allow direct querying within JSON fields using SQL functions:

```
SELECT data->>'name' FROM person_table WHERE id = 1;
```

Storing and accessing objects in relational databases requires a combination of ORM techniques, serialization, relational structuring, or object-relational extensions. While relational databases are optimized for structured data, modern enhancements like JSON support and ORM frameworks have made it easier to handle objects efficiently. The choice of method depends on application requirements, performance considerations, and scalability needs.

14.5: Object-Oriented Database Design

Object-Oriented Database Design (OODD) is a methodology for designing databases that align with the principles of Object-Oriented Programming (OOP). Unlike traditional relational database design, which relies on tables, rows, and columns, object-oriented database design structures data as objects, encapsulating both attributes (data) and behaviors (methods). This approach is particularly beneficial for applications that handle complex data types, multimedia content, real-time processing, and hierarchical relationships.

Object-Oriented Database Management Systems (OODBMS) such as ObjectDB, db, Versant, and GemStone/S support this design paradigm, enabling direct storage and retrieval of objects without the need for Object-Relational Mapping (ORM). Additionally, Object-Relational Database Management Systems (ORDBMS) like PostgreSQL and Oracle provide hybrid solutions that integrate object-oriented features into relational models.

Key Concepts of Object-Oriented Database Design

1. Objects and Classes

In OODD, data is modeled as objects, which are instances of classes.

- Objects store both data (attributes) and methods (behavior) in a single entity.
- Classes define a blueprint for objects, specifying attributes and behaviors.
- Objects persist in the database in the same way they exist in object-oriented programming, reducing the need for transformation.

Example of an Object in OODBMS

```
class Employee {
   String name;
   int employeeID;
   Address address; // Reference to another object
   void calculateSalary() {
      // Method logic
   }
}
```

Here, the Employee object contains attributes (name, employeeID) and a method (calculateSalary). It also contains a reference to another object (Address), demonstrating object composition.

2. Encapsulation

Encapsulation ensures that data is bundled with methods that operate on it, preventing unauthorized access.

- In an OODBMS, objects maintain their own states and behaviors, allowing operations to be performed directly on them rather than using SQL queries.
- This reduces complexity by allowing direct object manipulation instead of translating objects into relational data structures.

3. Inheritance

Inheritance allows new classes to derive properties and behaviors from existing classes, promoting code reusability.

- OODD supports hierarchical data modeling, where subclasses inherit attributes and methods from a parent class.
- This eliminates data redundancy and enables efficient data organization in the database.

Example of Inheritance in OODD

```
class Person {
   String name;
```

Notes

```
    int age;
}
class Employee extends Person {
    int employeeID;
    double salary;
}
```

Here, Employee inherits properties (name, age) from Person, reducing redundancy.

4. Polymorphism

Polymorphism allows objects of different types to be treated uniformly through method overriding or overloading.

- In an OODBMS, polymorphism ensures that queries and operations can be applied to objects of different subclasses seamlessly.
- This makes applications more adaptable to changing requirements.

Example of Polymorphism in OODD

```
class Shape {
    void draw() {
System.out.println("Drawing a shape");
    }
}

class Circle extends Shape {
    void draw() {
System.out.println("Drawing a circle");
    }
}
```

A draw() method can be called on any Shape object, whether it is a Circle or another shape, demonstrating polymorphism.

5. Object Identity (OID) and Relationships

Each object in an OODBMS has a unique Object Identifier (OID), which is independent of the object's data.

- OID is used instead of primary keys (as in relational databases) to maintain object uniqueness.
- Objects can be related using one-to-one, one-to-many, or many-to-many relationships.

Example of Object Relationships

- An Order object may contain multiple Product objects, forming a one-to-many relationship.

- Unlike relational databases, these relationships are maintained via direct object references rather than foreign keys, improving retrieval efficiency.

Steps in Object-Oriented Database Design

Step 1: Requirement Analysis

- Identify the entities (objects) that need to be stored in the database.
- Define the behaviors associated with each entity.
- Understand data relationships and constraints.

Step 2: Identify Classes and Attributes

- Define classes corresponding to real-world objects.
- Identify attributes and categorize them as simple types (integers, strings) or complex types (nested objects).
- Specify methods that belong to each class.

Step 3: Define Inheritance Hierarchies

- Identify common properties among classes and define superclasses.
- Establish subclass relationships to minimize redundancy.

Step 4: Establish Associations and Aggregations

- Define relationships between objects.
- Use aggregation (whole-part relationships) and composition (strong association) where necessary.

Step 5: Assign Object Identifiers (OIDs)

- Ensure each object has a unique identifier.
- OIDs remain constant even if object attributes change, unlike primary keys in relational databases.

Step 6: Normalize the Object Schema

- Avoid redundant attributes by following object normalization techniques similar to database normalization.
- Convert redundant objects into reusable components.

Step 7: Implement Methods and Constraints

- Define object methods that enforce business logic.
- Implement constraints (e.g., salary cannot be negative) at the object level.

Step 8: Optimize for Performance

- Use indexing techniques for efficient retrieval.
- Apply caching to store frequently accessed objects in memory.
- Consider partitioning large object collections.

Advantages of Object-Oriented Database Design

Better Handling of Complex Data

OODBMS efficiently stores multimedia, CAD models, XML, and hierarchical data, which is difficult in relational databases.

No Impedance Mismatch

Since objects are stored directly, there is no need for Object-Relational Mapping (ORM), reducing overhead.

Encapsulation and Reusability

Encapsulation keeps data and behavior together, while inheritance promotes code reuse.

Efficient Query Performance

Objects are retrieved using direct references (OIDs) rather than expensive JOIN operations in relational databases.

Scalability and Flexibility

OODBMS allows schema evolution, making it easier to accommodate changes without restructuring entire tables.

Challenges of Object-Oriented Database Design

Lack of Standardization

Unlike SQL-based relational databases, OODBMS lacks a universally accepted query language.

Steep Learning Curve

OODD requires familiarity with object-oriented programming concepts, making it difficult for traditional database administrators.

Limited Adoption

Due to wide enterprise reliance on RDBMS, many applications still require Object-Relational Mapping (ORM) rather than a full switch to OODBMS.

Object-Oriented Database Design (OODD) provides an efficient, flexible, and scalable approach to managing complex data structures by aligning with object-oriented programming principles. It overcomes limitations of relational databases, such as impedance mismatch and rigid schema structures, making it ideal for applications involving multimedia, CAD, IoT, and real-time systems. However, challenges such as lack of standardization, steep learning curve, and limited industry adoption must be considered before choosing an OODBMS over traditional RDBMS or ORDBMS solutions. As software development continues to embrace object-oriented paradigms, the demand for integrated object-oriented database systems is expected to grow.

Check Your Progress:

1. Explain how object-oriented features such as encapsulation, inheritance, and polymorphism enhance the modeling capabilities of relational databases.

----

2. Compare traditional relational databases with Object-Relational Databases (ORDBMS) in terms of structure, flexibility, and data modeling power.

14.6: Summary

Object-Oriented Features in Relational Databases were introduced to overcome the structural rigidity of the traditional relational model. The integration of object-oriented concepts such as encapsulation, inheritance, polymorphism, and object identity allows the database to model complex real-world entities more effectively. Encapsulation ensures that an object's data and behavior are bound together, promoting data abstraction and security. Inheritance enables the creation of hierarchical class structures that reuse existing definitions, reducing redundancy. Polymorphism allows objects to respond differently to the same operation, enhancing flexibility in application logic. Together, these features form the foundation of Object-Relational Database Systems (ORDBMS), which combine the robustness of relational storage with the expressiveness of object modeling. ORDBMS supports complex data types, multimedia objects, and user-defined types, making it ideal for engineering, scientific, and multimedia applications. By bridging the gap between data and behavior, it provides a unified framework for managing complex and heterogeneous data efficiently.

14.7: Exercises

Multiple Choice Questions:

1. Which of the following features allows objects to inherit properties from other objects?

a) Polymorphism

b) Encapsulation

c) Inheritance

d) Association

Answer: c) Inheritance

2. Encapsulation in an object-oriented database refers to:

a) Hiding internal details and exposing only necessary functionality

b) Combining multiple tables into one

c) Converting data into binary form

d) Deleting redundant objects

Answer: a) Hiding internal details and exposing only necessary functionality

3. Which of the following is NOT an object-oriented feature supported by ORDBMS?

a) Inheritance

b) Polymorphism

c) Data Redundancy

d) Encapsulation

Answer: c) Data Redundancy

4. Which SQL extension allows defining and using object-oriented data types?

a) SQL++

b) PL/SQL

c) Object SQL

d) JSON SQL

Answer: c) Object SQL

5. Object identity in ORDBMS ensures that:

a) Each object can be uniquely identified, independent of its value

b) Two objects with same values are always identical

c) Objects cannot be deleted

d) Object references are shared across databases

Answer: a) Each object can be uniquely identified, independent of its value

Descriptive Questions:

1. Explain the object-oriented features incorporated into relational databases.

2. Discuss encapsulation, inheritance, and polymorphism in the context of ORDBMS.
3. How do object-oriented features improve database modeling and performance?
4. Compare traditional relational databases and object-relational databases with examples.

14.8: References and Suggested Reading

- Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (2004). *Object-Oriented Modeling and Design with UML* (2nd ed.). Pearson.
- Bertino, E., & Martino, L. (2001). *Object-Oriented Database Systems: Concepts and Architectures*. Addison-Wesley.
- Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson.
- Coronel, C., & Morris, S. (2019). *Database Systems: Design, Implementation, and Management* (13th ed.). Cengage Learning.

Block 5: Object Oriented Database

Unit 15: Object-Oriented Data Models

Structure

15.1: Introduction

The **Object-Oriented Data Model (OODM)** represents data as objects, similar to the way data and functions are handled in object-oriented programming. Each object in the database includes both **state** (data or attributes) and **behavior** (methods or operations). This integration of data and behavior makes OODM highly expressive and suitable for modeling complex systems.

Object-oriented databases (OODBMS) based on this model support features such as **class hierarchies**, **object identity**, **inheritance**, and **message passing**. They are designed to handle complex data types and inter-object relationships that are difficult to represent in relational systems.

OODBMSs are widely used in applications involving **computer-aided design (CAD)**, **engineering**, **scientific data analysis**, and **multimedia systems**, where complex data and relationships must be modeled naturally and efficiently.

15.2: Learning Outcomes

After completing this unit, students will be able to:

1. Define and explain the object-oriented data model.
2. Describe the components of OODM such as classes, objects, attributes, and methods.
3. Understand the principles of inheritance, object identity, and message passing in data modeling.
4. Distinguish between OODBMS and traditional RDBMS architectures.
5. Identify real-world applications of object-oriented databases.

15.3: Introduction to Object-Oriented Data Models

The Object-Oriented Data Model (OODM) represents a significant evolution in database management systems, integrating the principles of object-oriented programming (OOP) with data storage and retrieval. Unlike traditional Relational Database Management Systems (RDBMS), which organize data into structured tables of rows and columns, the object-oriented model structures data as objects, similar to those used in programming languages such as Java, C++, and Python. These objects encapsulate both data attributes and behavioral methods, facilitating a more natural representation of real-world entities. The Object-Oriented Database Management System (OODBMS) extends this model by enabling direct storage and retrieval of objects without requiring conversion into relational tables. This approach eliminates the need for Object-Relational Mapping (ORM), which is necessary when using an RDBMS with object-oriented programming. As a result, OODM offers a more seamless integration between applications and databases, making it particularly suitable for complex data structures, multimedia applications, hierarchical data, and real-time systems.

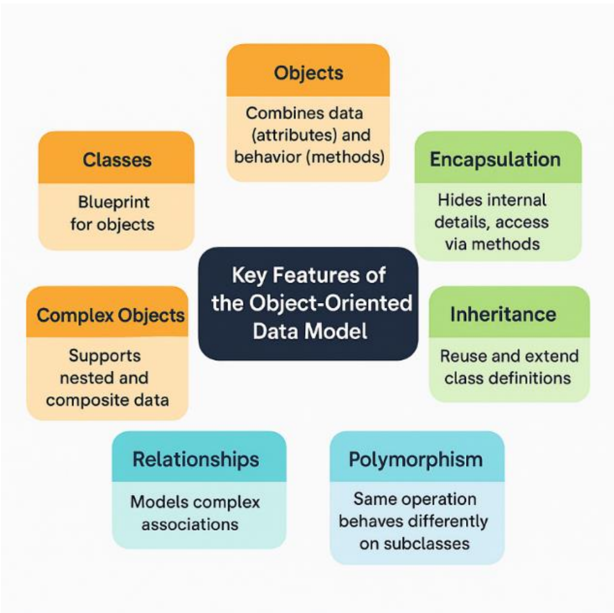

fig: 15.1 Key Features of the Object-Oriented Data Model

1. Objects as Fundamental Data Units

In an Object-Oriented Data Model, data is represented as objects, which are instances of classes. These objects store both attributes (data) and methods (functions), encapsulating behavior alongside data storage.

This design facilitates a more intuitive and flexible representation of entities within a system. For example, an Employee object may contain attributes such as name, employeeID, and salary, along with methods such as calculateBonus(). Unlike relational databases, where attributes and behavior are separated, OODM allows objects to self-manage their behavior and state, enhancing modularity and reusability.

2. Classes and Object Instances

The class serves as a blueprint for creating objects, defining their attributes and behaviors. Each instance of a class represents an individual object containing specific data values. For instance, a Car class may define attributes such as model, color, and speed. An instance of this class could be a Tesla Model S, characterized by a red color and a top speed of 200 km/h.

3. Encapsulation and Data Integrity

Encapsulation is a key principle of the object-oriented model, ensuring that data is bundled with its associated methods and protected from unauthorized access. Objects expose data through controlled interfaces, typically via getter and setter methods.

For example, in Java:

```
class Student {
   private String name;
   public String getName() { return name; }
   public void setName(String n) { name = n; }
}
```

Here, the name attribute is private, ensuring that it can only be accessed or modified through controlled methods. This design enhances data security, integrity, and modularity.

4. Inheritance and Code Reusability

The Object-Oriented Data Model supports inheritance, a mechanism that enables new classes to derive attributes and methods from existing classes. This feature promotes code reusability and hierarchical organization, reducing redundancy and improving maintainability. For example, a Manager class may inherit common properties from an Employee class, eliminating the need for redundant definitions.

```
class Employee {
   String name;
   int employeeID;
}
class Manager extends Employee {
   double bonus;
```

}

Here, the Manager class automatically inherits attributes from Employee, extending functionality without redefining common properties.

5. Polymorphism and Dynamic Behavior

Polymorphism allows different objects to respond to the same function call in multiple ways, enhancing flexibility and adaptability. This is particularly useful in object-oriented queries and dynamic data processing.

For instance, a draw() method can be applied to different shapes (Circle, Rectangle), each implementing its own version of the method:

```
class Shape {
   void draw() { System.out.println("Drawing a shape"); }
}

class Circle extends Shape {
   void draw() { System.out.println("Drawing a circle"); }
}
```

Here, invoking draw() on a Shape object may execute different behaviors based on the actual object type, demonstrating method overriding in polymorphism.

6. Object Identity (OID) and Unique Identification

Every object in an OODBMS is assigned a unique Object Identifier (OID), which remains constant throughout the object's lifecycle, even if attribute values change. Unlike primary keys in relational databases, OIDs provide efficient object retrieval and referencing without relying on external keys. For instance, a Customer object with OID C123 may reference an Order object with OID O456, creating a direct object relationship without foreign keys.

7. Relationships and Data Associations

Objects in an object-oriented database can be related through various associations:

- One-to-One: A Student object is linked to a LibraryCard object.
- One-to-Many: A Department contains multiple Employees.
- Many-to-Many: A Student can enroll in multiple Courses, and a Course can have multiple students.

Unlike relational databases, where relationships require foreign key constraints, OODBMS maintains direct references between objects, improving data retrieval efficiency.

15.4: Advantages of Object-Oriented Data Models

1. **Enhanced Representation of Complex Data:**
   - Supports multimedia, CAD models, hierarchical data, and real-world relationships.
2. **Seamless Integration with Object-Oriented Programming:**
   - Eliminates the need for Object-Relational Mapping (ORM), reducing conversion overhead.
3. **Reusability and Maintainability:**
   - Inheritance, encapsulation, and polymorphism facilitate efficient system design.
4. **Efficient Data Retrieval:**
   - Direct object references and OID-based indexing improve query performance compared to relational joins.
5. **Flexibility and Scalability:**
   - Objects can evolve dynamically, supporting schema evolution without requiring major restructuring.

Challenges and Limitations

1. **Lack of Standard Query Language:**
   - Unlike SQL, there is no universally accepted query language for OODBMS, making it less standardized.
2. **Steeper Learning Curve:**
   - Requires expertise in object-oriented programming and database management.
3. **Limited Enterprise Adoption:**
   - Many businesses rely on RDBMS solutions due to their mature ecosystem and widespread support.
4. **Complex Implementation:**
   - Object-oriented databases require efficient indexing and caching strategies to handle large datasets effectively.

The Object-Oriented Data Model (OODM) represents a paradigm shift in database management, offering a natural and intuitive approach to data storage by aligning with object-oriented programming principles. By encapsulating data and behavior within objects, it provides a flexible, scalable, and efficient solution for applications that require complex data modeling and hierarchical relationships. However, despite its advantages, the lack of standardization and the dominance of relational databases have limited its widespread adoption. Nonetheless, as modern applications increasingly demand dynamic and flexible data storage, OODBMS and hybrid Object-Relational Database Systems (ORDBMS) continue to gain traction, paving the way for next-generation data management solutions.

Check Your Progress:

1. Describe the fundamental components of the Object-Oriented Data Model (OODM) and explain how they differ from the elements of the relational model.

----

2. Discuss the advantages and challenges of implementing Object-Oriented Database Management Systems (OODBMS) in modern applications.

15.5: Summary

The Object-Oriented Data Model (OODM) represents a major advancement in database technology by combining the principles of object-oriented programming with data management. In this model, data is stored as objects that encapsulate both attributes (state) and methods (behavior). Objects are organized into classes, and relationships among them are established through inheritance and associations. The OODM supports key concepts such as object identity, class hierarchies, and message passing, which enable the modeling of complex real-world systems. Unlike relational databases, OODM allows for direct representation of multimedia, spatial, and scientific data without complex mapping. It provides high reusability and maintainability by enabling modular design. However, challenges such as lack of standardization, performance overhead, and integration with existing systems still exist. Despite these, Object-Oriented Databases have become essential in areas like computer-aided design (CAD), engineering, multimedia systems, and real-time simulations. The OODM thus offers a powerful and intuitive framework for managing complex and interrelated data structures.

15.6: Exercises

Multiple Choice Questions:

1. In an object-oriented data model, an object consists of:

a) Only data

b) Only methods

c) Data and methods together

d) Only attributes

Answer: c) Data and methods together

2. Which of the following best describes a class in OODM?

a) A specific instance of data

b) A collection of similar objects with shared structure and behavior

c) A physical storage file

d) A query language

Answer: b) A collection of similar objects with shared structure and behavior

3. Object identity in OODM ensures that:

a) Each object is identified by a unique identifier independent of its data value

b) Two objects with same values are considered identical

c) Objects can have duplicate keys

d) Objects are identified only by their class name

Answer: a) Each object is identified by a unique identifier independent of its data value

4. Which of the following is NOT a characteristic of object-oriented databases?

a) Inheritance

b) Message passing

c) Data redundancy

d) Object identity

Answer: c) Data redundancy

5. Object-oriented databases are particularly useful for:

a) Simple tabular data

b) Flat file processing

c) Complex and interrelated data structures

d) Text-based storage

Answer: c) Complex and interrelated data structures

Descriptive Questions:

1. Define the object-oriented data model and explain its key components.
2. Describe the role of inheritance and message passing in OODM.
3. Compare object-oriented databases with relational databases in terms of data representation and performance.
4. Explain the applications of object-oriented databases in modern computing systems.

15.7: References and Suggested Reading

- Atkinson, M. P., & Buneman, P. (1987). *Object-Oriented Database Systems*. Addison-Wesley.
- Bertino, E., & Martino, L. (2001). *Object-Oriented Database Systems: Concepts and Architectures*. Addison-Wesley.
- Elmasri, R., & Navathe, S. B. (2017). *Fundamentals of Database Systems* (7th ed.). Pearson Education.
- Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *The Unified Modeling Language Reference Manual* (2nd ed.). Addison-Wesley

Glossary

- ACID Properties: Set of principles (Atomicity, Consistency, Isolation, Durability) ensuring reliable transactions in a database.
- Application Layer: The middle tier in 3-tier architecture where application logic runs (e.g., Java, Python).
- Backup & Recovery: Mechanisms to restore data in case of failure or corruption.
- Big Data: Large and complex datasets characterized by Volume, Velocity, Variety, and Veracity.
- Clustered Storage: Storage method that groups related data physically close to improve performance.
- Concurrency Control: Ensures correct execution of transactions by multiple users at the same time.
- CREATE (DDL): SQL command to create a new object such as a table.
- Data Abstraction: Hiding complex storage details while presenting a user-friendly view of data.
- Data Definition Language (DDL): SQL language subset used to define or alter database schema (e.g., CREATE, ALTER).
- Data Independence: Ability to change storage structure without affecting applications.
- Data Manipulation Language (DML): SQL commands used to manage data in tables (e.g., SELECT, INSERT).
- Data Mining: Process of discovering patterns and knowledge from large datasets.
- Data Model: Framework for organizing data and defining relationships (e.g., Relational, ER, Object-Based).
- Data Warehouse: A central repository that stores historical data for analytical purposes.
- Database Administrator (DBA): Person responsible for database installation, security, tuning, and backups.
- Database Designer: Professional who defines the structure and schema of a database.
- Database Management System (DBMS): Software used to store, retrieve, and manage data in databases.
- Entity: An object or thing in the real world that is distinguishable from other objects (e.g., Student, Course).
- Entity-Relationship (ER) Model: Conceptual model used for database design showing entities, attributes, and relationships.

- Execution Plan: Optimized sequence of operations for executing a database query.
- Foreign Key: A key in one table that refers to the primary key in another table to establish relationships.
- FROM (SQL): Clause used to specify the table in a SQL query.
- GRANT (DCL): SQL command to give privileges to a user or role.
- GROUP BY: SQL clause used to group rows with the same values in specified columns.
- HAVING: Clause used to filter groups created by GROUP BY.
- Index: Database object that speeds up data retrieval operations.
- Instance: The current state or snapshot of the data in a database at a specific moment.
- INSERT (DML): SQL command used to add new records to a table.
- JOIN: SQL operation used to combine rows from two or more tables based on a related column.
- Logical Level: Middle level of data abstraction showing what data is stored and how it relates.
- LIKE: SQL operator used for pattern matching.
- MongoDB: A NoSQL database that stores data in JSON-like documents.
- MySQL: Popular open-source relational database management system.
- Normalization: Process of organizing data to reduce redundancy and improve integrity.
- NoSQL: Type of database that handles semi-structured or unstructured data (e.g., MongoDB, Couchbase).

- Object-Based Model: A data model that represents real-world entities as objects with attributes and methods.
- ORDER BY: SQL clause used to sort query results in ascending or descending order.
- Physical Level: The lowest level of data abstraction detailing how data is physically stored.
- Primary Key: A field (or combination of fields) that uniquely identifies each record in a table.
- PostgreSQL: Open-source object-relational database system.
- Query: A command used to retrieve or manipulate data from a database.

Notes

- Query Optimization: Process to enhance SQL query performance using indexes, execution plans, and rewriting.
- Relational Model: Data model that organizes data into tables with rows and columns.
- REVOKE (DCL): Removes access privileges from users or roles.
- ROLLBACK (TCL): Reverses changes made in a transaction since the last COMMIT.
- Schema: The overall design or blueprint of a database, including tables, attributes, and constraints.
- SELECT (DML): SQL command to retrieve data from a table.
- Semi-Structured Model: Data model used for loosely organized data like JSON or XML.
- System Analyst: User who connects system requirements with database implementation.
- TCL (Transaction Control Language): SQL subset to control transactions (e.g., COMMIT, ROLLBACK).
- TRUNCATE (DDL): Removes all records from a table while preserving its structure.
- Transaction: A sequence of database operations treated as a single logical unit.
- UPDATE (DML): SQL command to modify existing records in a table.
- User Roles: Defined set of permissions assigned to users to control database access.
- View Level: Highest level of abstraction, showing user-specific perspectives of data.
- WHERE Clause: SQL condition to filter query results.
- Weak Entity: An entity that cannot exist without a related strong entity and does not have a primary key.

MATS UNIVERSITY

MATS CENTRE FOR DISTANCE AND ONLINE EDUCATION

UNIVERSITY CAMPUS: Aarang Kharora Highway, Aarang, Raipur, CG, 493 441

RAIPUR CAMPUS: MATS Tower, Pandri, Raipur, CG, 492 002

T : 0771 4078994, 95, 96, 98 Toll Free ODL MODE : 81520 79999, 81520 29999

Website: www.matsodl.com