Aktu | Database Management System | BCS-501

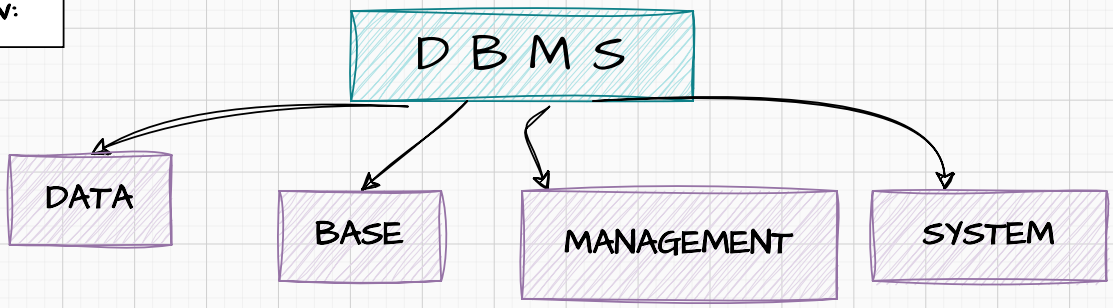
Syllabus

Introduction: Overview, Database System vs File System, Database System Concept and Architecture, Data Model Schema and Instances, Data Independence and Database Language and Interfaces, Data Definitions Language, DML, Overall Database Structure. Data Modeling Using the Entity Relationship Model: ER Model Concepts, Notation for ER Diagram, Mapping Constraints, Keys, Concepts of Super Key, Candidate Key, Primary Key, Generalization, Aggregation, Reduction of an ER Diagrams to Tables, Extended ER Model, Relationship of Higher Degree.

Topics

- Overview of DBMS
- DBMS vs. File System
- Database System Concept and Architecture (1-Tier, 2-Tier & 3-Tier)
- View of Data in DBMS
- Three-Level Architecture or Three Schema Architecture (2023-24 & 2022-23 AKTU (10Marks))
- Schema & Instances
- Data Models - (2021-22 , 2022-23 (Aktu-10Marks))
- Database Languages in DBMS
- Database Interfaces
- Overall Database Structure - (Aktu - 2021-22 & 2023-24)
- ER-Model
- Mapping Constraints
- Keys in DBMS
- DBA Roles
- Extended ER Model
- EER Features
- Reduction of an ER Diagrams to Table (2022-23 Aktu)

Overview:



DATA

It is a collection of raw, unprocessed facts, figures, or details that by themselves do not convey any specific meaning.

Example:
A list of numbers: "100, 200, 150."
Names, dates, and addresses: "John, 23rd Sept, 123 Main St."
These are just raw facts without context or interpretation.

Types of Data:

1. **Structured Data**: Organized in a defined format (e.g., tables in a database).
2. **Unstructured Data**: Unorganized and not easily interpretable (e.g., images, videos, plain text).

Information:

It is processed, organized, or interpreted data that conveys meaning or knowledge.

Examples:

- "John purchased items worth $100 on 23rd Sept at 123 Main St.

Key Idea: Information provides context and meaning to data, making it useful for decision-making or understanding.

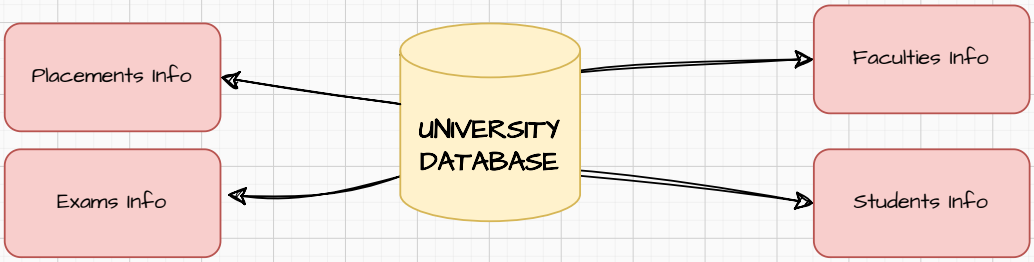
- Data is the raw input, and when processed or organized, it becomes information.

	DATA	INFORMATION
Definition	Raw, unprocessed facts and figures.	Processed or organized data that conveys meaning.
Nature	Unorganized and may not be useful on its own.	Organized, structured, and meaningful.
Example	"100, 200, 150," or "John, 23rd Sept, 123 Main St."	"John purchased items worth $100 on 23rd Sept at 123 Main St."
Representation	Simple observations, details, or statistics.	Contextualized facts that help with decision-making.
Processing	Data is the raw material that needs to be processed.	Information is the result of processing data.
Dependency	Data does not depend on information.	Information depends on data to exist.

What is Database?

A database is a **collection of interrelated data** that helps in the efficient retrieval, insertion, and deletion of data from the database and organizes the data in the form of **tables, views, schemas, reports**, etc.

For Example,
a university database organizes the data about students, faculty, admin staff, etc. which helps in the efficient retrieval, insertion, and deletion of data from it.



What is DBMS?

A Database Management System (DBMS) is a **software system** designed to create, manage, and manipulate databases. It allows users to efficiently **store, retrieve, update, and manage** data in a structured way.

- Ex:- MySQL, Oracle, etc

Key Functions of a DBMS:

1. Data Storage:

DBMS stores data in organized structures, typically in the form of tables with rows and columns. It ensures that data is stored efficiently and securely.

2. Data Retrieval:

DBMS allows users to query the database using languages like SQL (Structured Query Language) to fetch specific data quickly.

It uses indexes and optimized storage techniques to speed up the retrieval process, even when dealing with large amounts of data.

3. Data Manipulation:

With a DBMS, users can insert, update, and delete data using commands like INSERT, UPDATE, and DELETE in SQL.

It allows for batch processing and transactions, ensuring data integrity and consistency during operations.

4. Data Security and Integrity:

DBMS enforces access controls, ensuring only authorized users can view or manipulate certain data.

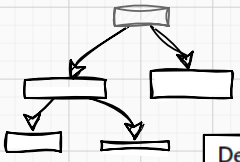
It ensures data integrity by enforcing constraints like primary keys, foreign keys, and unique constraints.

5. Backup and Recovery:

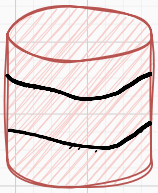
DBMS systems provide automatic backup and recovery features to protect data from accidental loss or corruption.

What is File System?

A File System is a method and structure used by an **operating system to organize, store, retrieve, and manage files** on a storage device such as a **hard drive, SSD, or USB**. It provides the foundation for storing data in files and directories, but **lacks the features** of a Database Management System (DBMS).



File System Vs DBMS



Definition	A system that manages files on a disk or storage device.	A software that manages databases, allowing storage, retrieval, and manipulation of data.
Data Storage	Stores data in files within directories.	Stores data in structured tables using rows and columns.
Data Organization	Unstructured; each file is independent.	Highly structured; data is related through tables, schemas, and relationships.
Redundancy	High redundancy due to lack of structure and relationships.	Minimizes redundancy by using relational models and enforcing constraints.
Data Integrity	No inherent mechanisms to ensure data accuracy or integrity.	Enforces data integrity through constraints like primary keys, foreign keys, and unique constraints.
Security	Basic file-level permissions (read, write, execute).	Fine-grained security controls; can restrict access based on user roles.
Concurrency Control	Minimal support for multiple users accessing files simultaneously.	Advanced concurrency control with mechanisms like locking, time-stamping, and transactions.
Data Querying	No querying capabilities; users access data by opening files.	Supports complex querying with SQL (Structured Query Language).

a **DBMS** is a more advanced system than a **File System**, especially when handling large and complex datasets that require consistency, security, and efficiency.

Data Relationships	No direct support for relationships between data files.	Supports relationships between tables via primary and foreign keys.
Data Independence	Low data independence; changes to file structure may affect applications.	High data independence; changes to database structure do not affect application code.
Backup and Recovery	Manual backup, with limited recovery options.	Automated backup and recovery mechanisms, including point-in-time recovery.
Transactions	Does not support atomic transactions.	Supports atomic transactions (ACID properties: Atomicity, Consistency, Isolation, Durability).
Performance	Slower when dealing with large amounts of data due to lack of indexing and optimization.	Optimized for performance with indexing, query optimization, and efficient storage mechanisms.
Example	File systems like FAT, NTFS, ext4 (e.g., in Windows or Linux).	DBMS like MySQL, Oracle, PostgreSQL, Microsoft SQL Server.

Database System Concept and Architecture

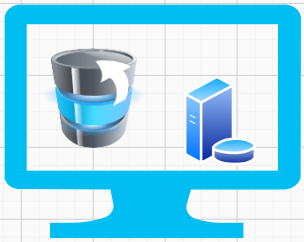
A database system refers to a **system that manages databases,** including the database itself and the Database Management System (DBMS). The architecture of database systems can be divided into **1-tier, 2-tier, and 3-tier architectures**, each offering different levels of abstraction and separation between users and the database.

1-Tier Architecture (Simple Architecture)

It is the **simplest architecture** of Database in which the **client, server, and Database all reside on the same machine**. A simple one tier architecture example would be anytime you install a Database in your system and access it to practice SQL queries. But such architecture is **rarely used in production.**

Advantages:

- Easy to manage.
- Suitable for local applications.



Disadvantages:

- Lacks scalability and security.
- Only suitable for small systems.

2-Tier Architecture (Client-Server Architecture)

In a 2-tier architecture, the system consists of **two layers**: the **client (frontend) and the server (backend)**. The client application **communicates directly with the database** server over a network.

Client: The user interface where users perform queries and operations (e.g., a desktop application).

Server: The database server, where the DBMS is installed and processes queries.

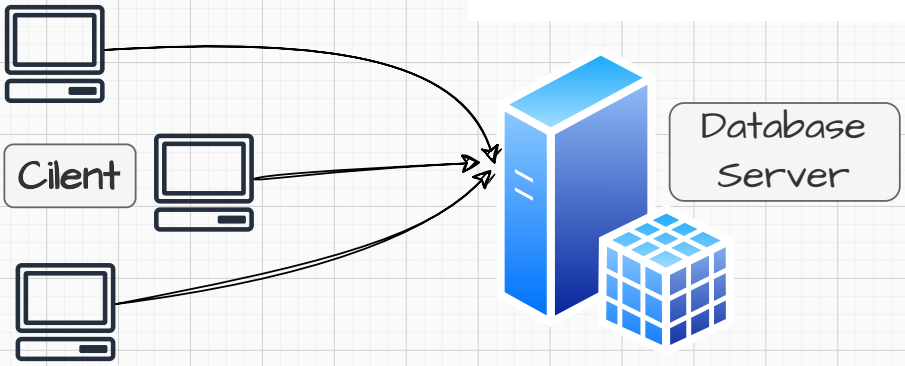
Use Case: This architecture is commonly used in desktop applications where multiple users connect to a central database server over a network.

Advantages:

-> Better performance for small to medium-sized systems.
-> Easier to manage security since the database is centrally located

Disadvantages:

-> Not very scalable for large systems.
-> Limited in handling multiple requests simultaneously, leading to performance issues.



3-Tier Architecture

Another layer exists between the client and server in the 3-Tier architecture. The **client cannot communicate directly** with the server with this design.
On the client side, **the program communicates with an application server**, which then communicates with the database system.
Beyond the application server, **the end-user has no knowledge of the database's existence**. Aside from the application, the database has no knowledge of any other users.

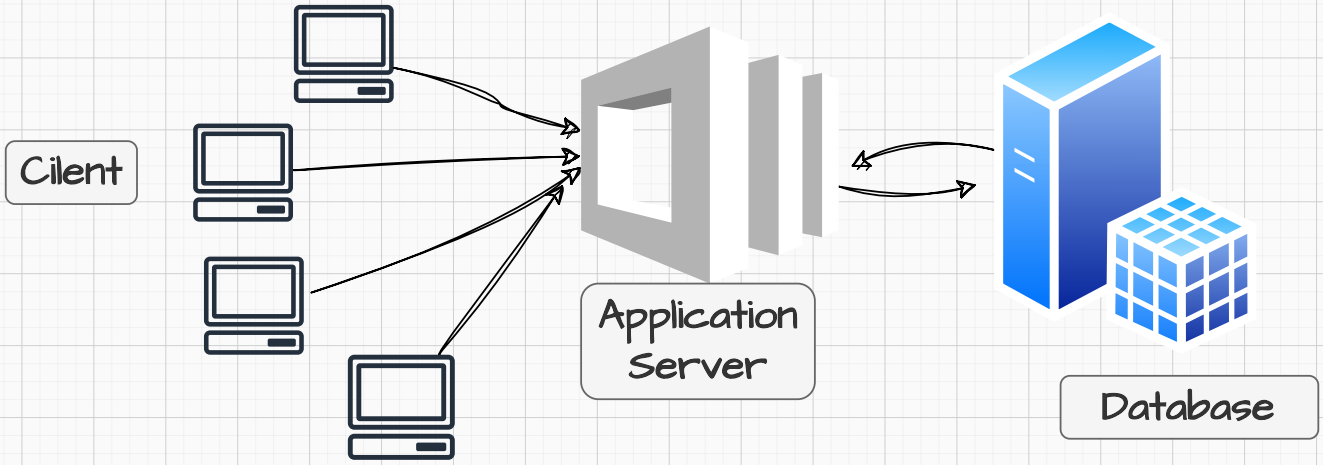
Client (UI): This is the user interface (e.g., web browser, mobile app) that users interact with.
Application Tier (Logic Layer): This is where the business logic resides. It processes user requests, communicates with the database, and returns data to the user interface.
Server: The database where data is stored and managed. The application tier communicates with the database using SQL queries.

Advantages

- **Scalable**: Easily handles large numbers of users and complex operations.
- **Security**: The application tier acts as a **middleware** that isolates the client from direct database access.
- **Maintainability**: Each layer can be updated or changed independently.

Disadvantages:

- More complex to set up and manage compared to 1-tier or 2-tier systems.
- Performance can be slower due to multiple layers of communication.



DBMS as Middleware:

In a 3-tier architecture, the DBMS functions as middleware between the application layer and the database. It provides the necessary services for querying, updating, and managing the database while abstracting the complexities of data storage from the application and the end user.

View of Data in DBMS

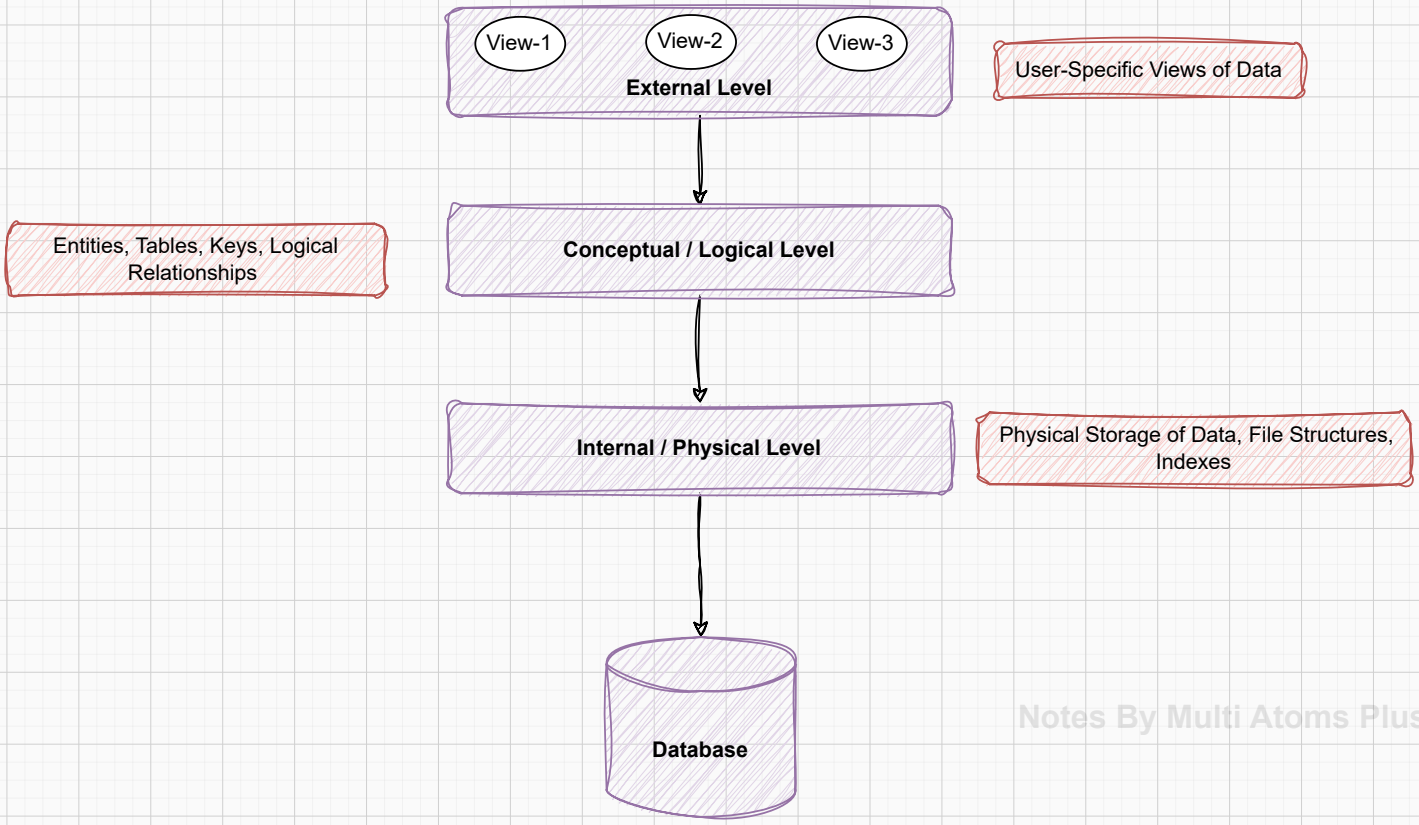
In a Database Management System (DBMS), the view of data refers to how data is presented and organized for users and applications. This view can be broken down into three levels of abstraction, which separates how data is stored from how it's viewed and accessed.

2023-24 & 2022-23 AKTU (10Marks)

Q. Discuss three level of abstractions or schemas architecture of DBMS.

Three-Level Architecture or Three Schema Architecture

The Three-Level Architecture (also known as Three-Schema Architecture) is a framework proposed by the **ANSI/SPARC** in the **1970s** for database management systems. It provides a way to separate user interactions with data from how the data is actually stored and maintained. The **three levels (schemas) of abstraction** help create flexibility, improve security, and ensure efficient data management.



1. Internal Level (Physical Schema)

Definition: The internal level represents the physical storage of data. It defines how data is stored on the storage medium, including file structures, indexes, and memory allocation strategies.
Purpose: This level focuses on the performance optimization and storage efficiency of data.
Responsibilities:

- Managing storage formats (binary, text, etc.).
- Handling the storage devices (e.g., disks, SSDs).
- Managing physical structures like indexes and block organization.

Example: A DBMS might store customer data in rows in a binary file and use indexing to make searching fast.

2. Conceptual Level (Logical Schema)

Definition: The conceptual level describes the overall structure of the database from a logical perspective. It defines the relationships, entities, attributes, and constraints.
Purpose: This level abstracts what data is stored in the database and the relationships among them.
Responsibilities:

-
- Defining entities (like Customer, Order) and relationships (e.g., each customer can place multiple orders).
- Creating tables, columns, data types, and keys (primary keys, foreign keys).
- Managing logical relationships between tables.

Example: A logical schema might define a Customers table with attributes like CustomerID, Name, and Email. It may also define the relationship between Customers and Orders.

3. External Level (View Schema)

Definition: The external level defines how the data is viewed by individual users or applications. It focuses on providing different views of the same data based on the needs of various user groups.

Purpose: This level provides security and simplicity by showing only the necessary parts of the database to different users while hiding irrelevant or sensitive data.

Responsibilities:

- Creating customized views for different user roles (e.g., a sales representative might only see customer names and sales data, while an HR person might see employee details).
- Restricting access to sensitive data by defining specific views.
- Simplifying complex database structures for end-users by hiding technical details.

Example: A SalesReport view might show only CustomerName and SalesAmount from the Customers and Orders tables, without exposing other sensitive information like customer addresses or credit card details.

Benefits of the Three-Level Architecture

1. **Data Abstraction:**
Separates the user's view from the physical structure of the database.
Provides users with different views of the same data without altering the underlying storage.

2. **Data Independence:**

- **Physical Data Independence:** Changes in the internal level (storage) don't affect the conceptual or external levels.
- **Logical Data Independence:** Changes in the conceptual level (e.g., modifying the schema) don't affect how users view the data at the external level.

3. **Improved Security:**
Users only access the part of the data they are authorized to see. This is managed through views at the external level.

4. **Flexibility:**
Enables multiple user views without impacting the logical structure of the database, allowing different departments or applications to interact with the database as needed.

Level	Description	Use Case
Physical Level	Describes how data is physically stored on storage devices.	Managed by DBAs to optimize performance and storage.
Logical Level	Describes what data is stored and the relationships between them.	Used by developers to design the schema and relationships.
View Level	Describes how data is viewed by users or specific applications.	Used to create customized views for different user groups.

Instances and Schema

In a database, instances and schemas are fundamental concepts that describe the structure and state of the database. They help define how data is stored, represented, and manipulated over time.

Schema:

Definition: A schema is the overall design or structure of the database. It defines the organization of data and the relationships between the different entities (tables) in the database.

Nature: The schema is static and usually does not change frequently. It is the blueprint of the database that outlines how the data is logically organized.

Components:

- **Tables**: Entities that store the data (e.g., Customers, Orders).
- **Columns**: Attributes of the tables (e.g., CustomerID, Name, OrderDate).
- **Relationships**: How tables relate to each other (e.g., foreign keys).
- **Constraints**: Rules applied to the data (e.g., primary keys, unique constraints, foreign keys)

Types of Schemas:

1. **Physical Schema**: Describes how data is stored physically (file systems, indexing).
2. **Logical Schema**: Describes the logical structure of the database (tables, columns, relationships).
3. **View Schema**: Describes the views that different users or applications have of the data.

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(100),
    Address VARCHAR(255)
);
```

Instances:

Definition: An instance refers to the actual data in the database at a specific point in time. It is the content stored in the database, based on the schema.

Nature: Instances are dynamic and constantly change as data is inserted, updated, or deleted. The schema remains the same, but the instance evolves with time.

```
INSERT INTO Customers (CustomerID, Name, Email, Address)
VALUES (1, 'Alice', 'alice@example.com', '123 Elm St');
```

Aspect	Schema	Instance
Definition	The blueprint or structure of the database.	The actual data stored in the database.
Nature	Static (changes infrequently).	Dynamic (changes frequently).
Role	Describes how data is organized and structured.	Refers to the actual values and content.
Examples	Table definitions, column types, constraints.	Rows of data in the tables.
Changes	Typically updated when the database is modified (e.g., adding a new table).	Changes occur every time data is inserted, updated, or deleted.

Q. What are the different types of Data Models in DBMS? Explain them.

2021-22 , 2022-23 (Aktu-10Marks)

Data Models in DBMS

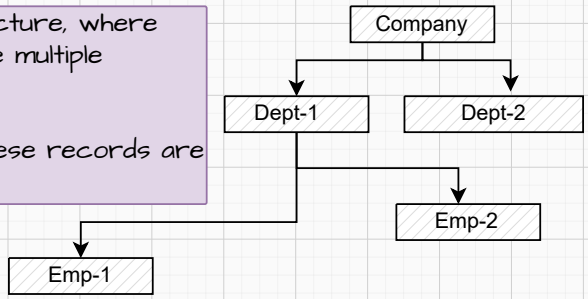
In a Database Management System (DBMS), data models define how data is stored, organized, and manipulated. They provide a structured framework for representing the relationships between data entities and offer rules for handling data operations.

There are several types of data models, each serving different purposes and providing various methods of organizing and accessing data. The most commonly used models in DBMS are:

1. Hierarchical Data Model

Definition: The hierarchical model organizes data in a tree-like structure, where each record (entity) has a single parent, and each parent can have multiple children.

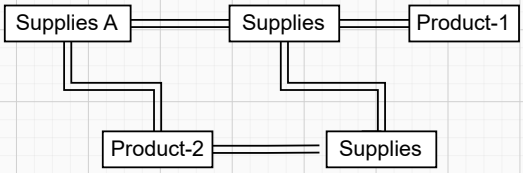
Structure: Data is represented as a collection of records, and these records are connected through parent-child relationships.



2. Network Data Model

Definition: The network model extends the hierarchical model by allowing more complex relationships, including many-to-many relationships. Data is organized as graphs, with records connected by pointers.

Structure: Data records are connected through sets, which define relationships between records.



3. Relational Data Model

Definition: The relational model organizes data into tables (relations), where each table consists of rows (records) and columns (attributes). Relationships between tables are defined through foreign keys.

Structure: Data is stored in relations (tables), and each table represents an entity or relationship. SQL is the standard query language used to interact with relational databases.

Customer Table

Customer ID	Name	Email
1	Sagar	--
2	King	--

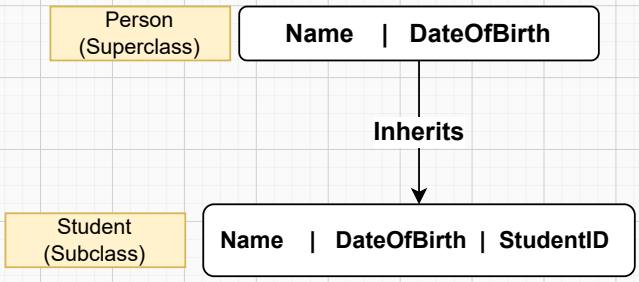
Order Table

Order ID	CustomerID	Date
101	1	--
102	2	--

4. Object-Oriented Data Model

Definition: The object-oriented model stores data as objects, similar to how object-oriented programming (OOP) languages like Java or C++ define objects. Each object contains attributes (data) and methods (behavior).

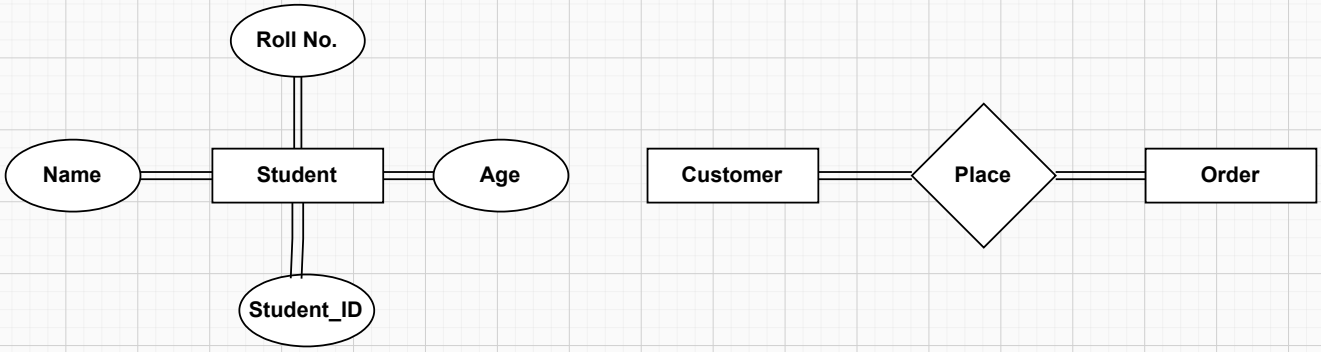
Structure: Objects are instances of classes, and classes define the properties and behaviors of those objects. Relationships are represented using inheritance and composition.



5. Entity-Relationship (ER) Model

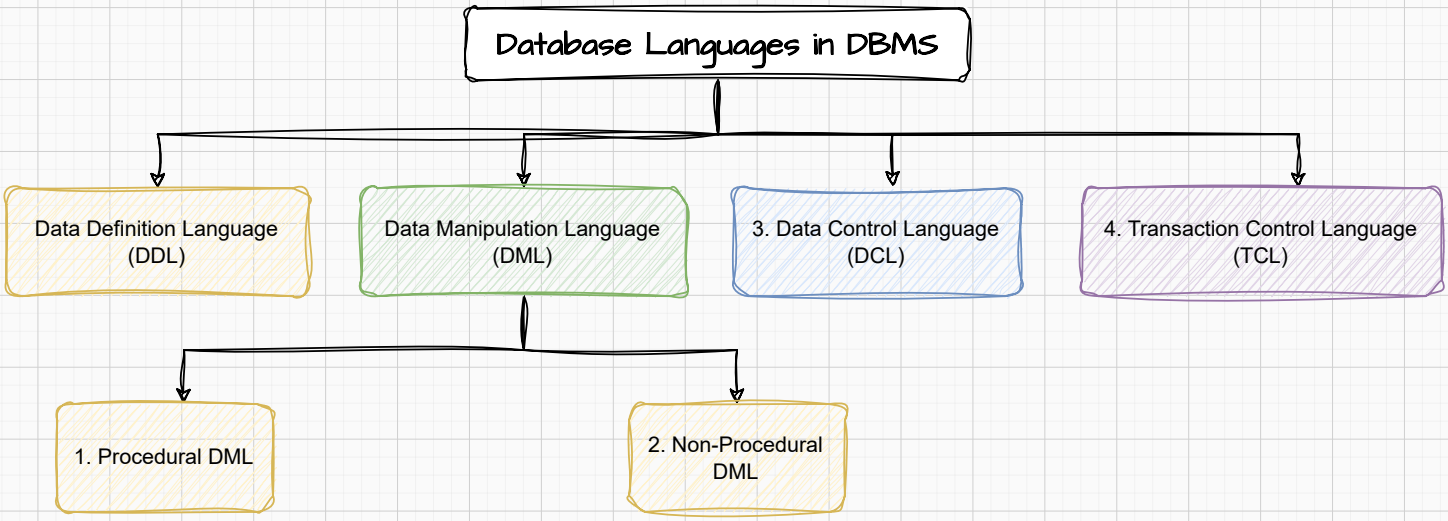
Definition: The ER model is used to visually represent the entities (objects) in the database and their relationships. It is often used during database design to model the logical structure.

Structure: Data is modeled using entities, attributes, and relationships. Entities are objects or concepts (e.g., Customer, Product), and relationships define how these entities interact with each other (e.g., a Customer places an Order).



Database Languages in DBMS

A Database Management System (DBMS) uses several languages to perform various operations like defining data structures and manipulating data. These languages form the backbone of interaction between users and the database. The most commonly used database languages are:



I. Data Definition Language (DDL)

DDL is used to define the structure of the database. It includes commands that allow users to create, modify, and delete database objects such as tables, indexes, views, and schemas.

Common DDL Commands:

- **CREATE**: Creates new database objects (e.g., tables, indexes).

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(100),
    Department VARCHAR(50)
);
```

- **ALTER**: Modifies existing database objects (e.g., adding a new column to a table)

```
ALTER TABLE Employees ADD Salary DECIMAL(10, 2);
```

- **DROP**: Deletes existing objects from the database (e.g., tables, indexes).

```
DROP TABLE Employees;
```

- **TRUNCATE**: Removes all records from a table without deleting the table itself.

```
TRUNCATE TABLE Employees;
```

- **RENAME**: The RENAME command only changes the name of the object; it does not affect the data or structure of the table.

```
RENAME Employees TO Staff;
```

Purpose: DDL commands help manage the structure of the database by creating, altering, and deleting database objects.

2. Data Manipulation Language

DML (Data Manipulation Language) is a subset of SQL used to manipulate and interact with the data stored within database objects. DML commands allow you to retrieve, insert, modify, and delete data in tables.

Common DML Commands:

- **SELECT**: Retrieves data from the database.

```
SELECT * FROM Employees WHERE Department = 'IT';
```

- **INSERT**: Adds new records to a table.

```
INSERT INTO Employees (EmployeeID, Name, Department) VALUES (1, 'John Doe', 'HR');
```

- **UPDATE**: Modifies existing data in the table.

```
UPDATE Employees SET Department = 'Finance' WHERE EmployeeID = 1;
```

- **DELETE**: Removes data from a table.

```
DELETE FROM Employees WHERE EmployeeID = 1;
```

- **DML commands focus on manipulating the data inside tables, as opposed to DDL, which deals with database structure.**
- **These commands are typically part of day-to-day database operations.**
- **SELECT is the most commonly used DML command for retrieving data.**

State the procedural DML and nonprocedural DML with their differences

Aktu-(2021-22 & 2023-24) 10marks

DML can be categorized into two types based on how the data is accessed and manipulated:

1. Procedural DML:

Procedural DML requires the user to specify how to get the data. In other words, it involves describing the exact steps the system should follow to retrieve or manipulate data.

Characteristics:

- The user must specify what data is needed and how to retrieve it.
- It requires knowledge of the database structure and flow of the query.
- The user has more control over the execution process, which often involves looping or conditional operations.

Examples: Relational Algebra & PL/SQL

```
DECLARE
  totalSalary NUMBER(10);
BEGIN
  SELECT SUM(Salary) INTO totalSalary FROM Employees WHERE Department = 'IT';
  DBMS_OUTPUT.PUT_LINE('Total Salary: ' || totalSalary);
END;
```


2. Non-Procedural DML:

Non-Procedural DML requires the user to specify what data is needed, but not how to get it. The system determines the best way to execute the query to retrieve or manipulate data.

Characteristics:

- The user only specifies what data is needed, leaving the query execution process to the DBMS.
- The focus is on the result, not the method of retrieval.
- Easier to use for end-users since it does not require an understanding of the underlying data structure or query optimization.

Examples: SQL

```
SELECT SUM(Salary) FROM Employees WHERE Department = 'IT';
```

Aspect	Procedural DML	Non-Procedural DML
Definition	Specifies **how** to retrieve or manipulate data.	Specifies **what** data to retrieve, but not how.
Control	The user controls the exact steps and methods.	The DBMS controls how the query is executed.
Complexity	More complex; requires knowledge of the database structure.	Easier to use; focuses on results rather than process.
Example Languages	Relational Algebra, PL/SQL, T-SQL	SQL, Tuple Relational Calculus, Domain Relational Calculus
Flexibility	Offers more flexibility in query execution.	Less control over execution but simpler to write.
Use Case	Suitable for complex operations where specific execution details are important.	Suitable for general queries where the result is more important than the method.

3. Data Control Language (DCL)

DCL is used to control access to the database. It includes commands that manage user permissions and access controls.

Common DCL Commands:

- **GRANT**: Gives user access privileges to database objects.

```
GRANT SELECT, INSERT ON Employees TO 'user1';
```

- **REVOKE**: Removes user access privileges.

```
REVOKE INSERT ON Employees FROM 'user1';
```

4. Transaction Control Language (TCL)

TCL commands manage transactions within the database, ensuring that groups of operations are either completely executed or not executed at all.

Common TCL Commands:

- **COMMIT**: Saves all changes made during the transaction.

```
COMMIT;
```

- **ROLLBACK**: Reverts the database to its previous state before the transaction began.

```
ROLLBACK;
```

- **SAVEPOINT**: Sets a point within a transaction to which you can roll back.

```
SAVEPOINT savepoint1;
```

The database languages (DDL, DML, DCL, and TCL) are essential tools for managing and interacting with a database. Each language serves a distinct purpose, from defining and manipulating data structures to controlling user access and managing transactions.

Category	Full Form	Purpose	Key Commands
DDL	Data Definition Language	Defines or modifies database structures and schema.	`CREATE`, `ALTER`, `DROP`, `TRUNCATE`, `RENAME`, `COMMENT`
DML	Data Manipulation Language	Manages and manipulates data within tables.	`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`
DCL	Data Control Language	Controls access to data and permissions.	`GRANT`, `REVOKE`
TCL	Transaction Control Language	Manages transactions in the database, ensuring data integrity.	`COMMIT`, `ROLLBACK`, `SAVEPOINT`, `SET TRANSACTION`

Interfaces in DBMS

A database management system (DBMS) interface is a user interface that allows for the ability to input queries to a database without using the query language itself. User-friendly interfaces provided by DBMS may include the following:

1. Command Line Interface (CLI):

- Uses command-line tools to execute SQL queries.
- Common for developers or administrators.
- Example: MySQL CLI, PostgreSQL psql.

2. Graphical User Interface (GUI):

- Provides a graphical interface to interact with the database.
- Often used by non-technical users for easier database operations.
- Example: phpMyAdmin, MySQL Workbench.
- Examples: Applications like Microsoft Access, Oracle SQL Developer, and MySQL Workbench.

3. Menu-Based Interfaces

- User-Friendly: Provides an intuitive point-and-click approach for easy database interaction.
- Predefined Commands: Offers a list of valid options to reduce errors during operations.
- Guided Workflow: Guides users through step-by-step processes for efficient task completion.

4. Application Programming Interface (API):

- A set of programming instructions that allow software applications to communicate with the database, enabling developers to perform database operations programmatically.
- RESTful APIs, GraphQL APIs, and database connectors/libraries in various programming languages.

5. Web-Based Interface:

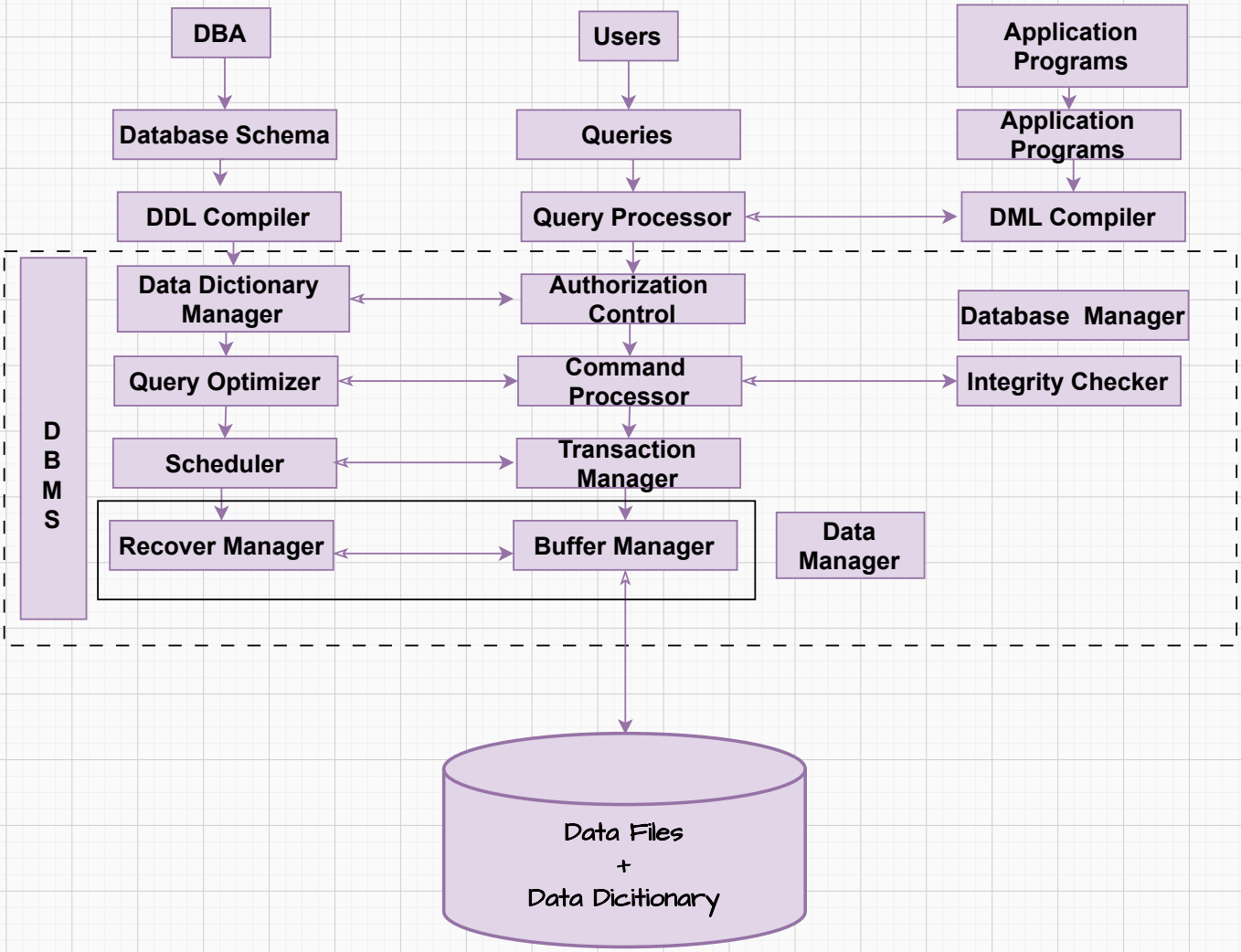
- An interface accessed through a web browser, allowing users to interact with databases via web forms and applications.
- Examples: phpMyAdmin, Adminer, and custom web applications using frameworks like Django or Flask.

A database interface plays a crucial role in enabling users and applications to efficiently interact with databases, ensuring that data can be managed and accessed effectively. The choice of interface often depends on user preferences, technical proficiency, and the specific requirements of the task at hand.

Structure of Database Management System

- Refers to the internal structure of a Database Management System (DBMS).
- It includes the components like the **Query Processor, Storage Manager, and Disk Storage**.
- Focuses on how the DBMS operates internally to manage and process data.

The Database Architecture focuses on internal DBMS components, while Tier Architecture emphasizes the structure of how a database is integrated into a broader application system.



Components of a Database System

1. Query Processor:

Role: Converts user queries into instructions that the system can execute.

Subcomponents:

- **DML Compiler:** Converts Data Manipulation Language (DML) statements into low-level machine instructions.
- **DDL Interpreter:** Converts Data Definition Language (DDL) statements into tables containing metadata.
- **Embedded DML Pre-compiler**: Transforms DML in application programs into procedural calls.
- **Query Optimizer:** Executes the instructions generated by the DML compiler, optimizing them for performance.

2. Storage Manager:

Role: Provides an interface between the queries and the physical data storage. It ensures data consistency, integrity, and proper data access control.

Subcomponents:

- **Authorization Manager:** Handles role-based access control to ensure only authorized users can access data.
- **Integrity Manager:** Ensures that integrity constraints (like primary keys and foreign keys) are enforced when data is modified.
- **Transaction Manager:** Manages transaction scheduling to maintain database consistency before and after transactions.
- **File Manager:** Handles file storage, managing file space, and structuring the data in the database.
- **Buffer Manager:** Manages cache memory and controls the transfer of data between secondary storage (e.g., hard drives) and primary memory.

3. Disk Storage:

Role: The physical layer where data is stored, organized, and managed.

Subcomponents:

- **Data Files:** Store the actual data in the database.
- **Data Dictionary:** Contains metadata, describing the structure and organization of database objects.
- **Indices:** Enable faster data retrieval by creating indexes on specific fields in the database tables.

Database Architecture deals with the internal workings of a DBMS, including query processing, storage management, and data integrity.

2021-22, 2023-24 Aktu

Role Of DBA (Database Administrator)

It is a professional responsible for managing, maintaining, and securing databases within an organization. They ensure that databases are properly designed, optimized, and safeguarded, while also managing user access, backups, and recovery processes to maintain the integrity and availability of data.

- **Database Design and Implementation**: Defining the structure of databases and implementing them to meet organizational needs.
- **Performance Monitoring**: Ensuring databases run efficiently and tuning them for optimal performance.
- **Security Management**: Implementing security measures to protect data from unauthorized access or breaches.
- **Backup and Recovery**: Ensuring regular backups and implementing recovery strategies in case of data loss.
- **Data Integrity**: Ensuring data accuracy, consistency, and preventing corruption.
- **User Management**: Managing database users, their access, and privileges.
- **Software and Hardware Management**: Installing, upgrading, and maintaining database software and hardware.

ER Model Concepts:

The **Entity-Relationship (ER) model** is a conceptual framework used to represent the data and relationships in a **database**. It helps in designing a database by defining **entities, their attributes, and the relationships** between them.

- ER diagrams represent database schema graphically and can be easily converted into relational tables.
- They model real-world objects, making it easy to visualize data structures.
- ER diagrams are easy to understand, even for non-technical users.
- They provide a standardized approach to visualizing data relationships and structure.

Components of an ER Diagram

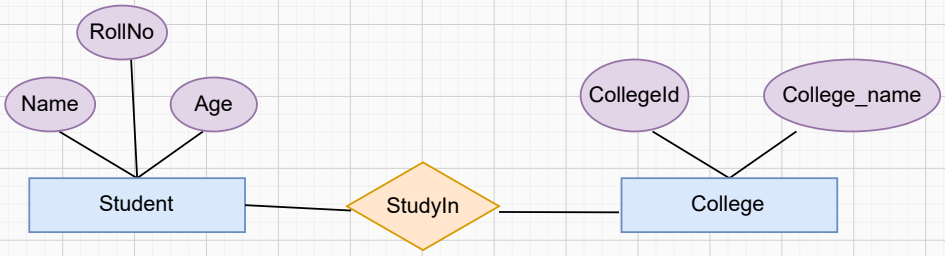
Entities: Objects or things in the real world that are distinguishable from other objects. Examples include "Student," "Course," and "Teacher."

Attributes: Characteristics or properties of an entity. For example, for a "Student" entity, attributes might include Student_ID, Name, and Age.

Relationships: Associations between entities. For example, a "Student" can enroll in a "Course," establishing a relationship between these entities.

Symbols Used in ER Model

1. **Rectangles**: Represent entities.
2. **Ellipses**: Represent attributes.
3. **Diamonds**: Represent relationships among entities.
4. **Lines**: Connect entities to attributes or relationships.



Entity Set: It is a collection of similar types of entities that share the same properties or attributes. In simpler terms, it's a group of entities that belong to the same type.

- If "Student" is an entity type, then each individual student, such as "John" or "Emily," is an entity.
- The collection of all students, such as John, Emily, and others, forms an Entity Set.

Types of Entities:

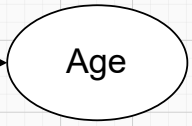
- **Strong Entity**: Can exist independently and has a primary key that uniquely identifies each instance.
- Represented by a rectangle.
- Example: Employee, Student.

- **Weak Entity**: Cannot exist independently and relies on a strong entity.
- Does not have its own primary key.
- Represented by a double rectangle.
- a ROOM can only exist in a BUILDING

Types of Attributes

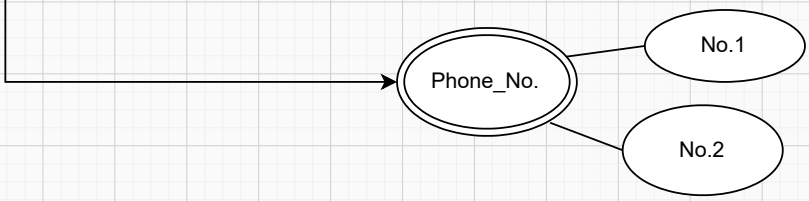
1. Single-Valued Attribute

Definition: An attribute that holds only a single value for a particular entity.
Example: Age of a person, Employee_ID of an employee.
Representation: In ER diagrams, it is represented by a single ellipse.



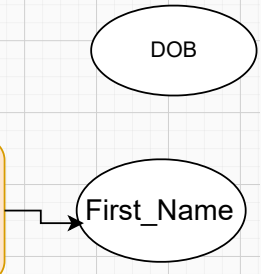
2. Multi-Valued Attribute

Definition: An attribute that can have multiple values for a particular entity.
Example: Phone_No of a person (since a person may have more than one phone no.).
Representation: In ER diagrams, it is represented by a double ellipse.



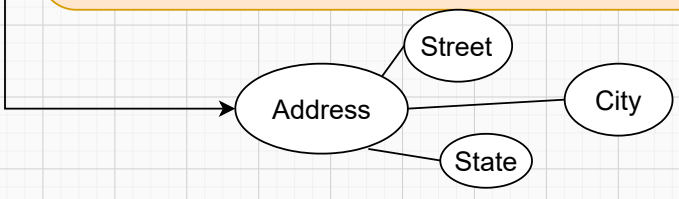
3. Simple (Atomic) Attribute

Definition: An attribute that cannot be divided further into smaller sub-attributes.
Example: First_Name, Last_Name, Date_of_Birth.
Representation: It is represented by a single oval in ER diagrams.



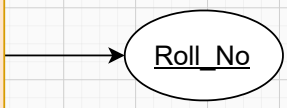
4. Composite Attribute

Definition: An attribute that can be divided into smaller sub-attributes, each representing a part of the whole attribute.
Example: Address which can be divided into Street, City, State, and Zip_Code.
Representation: In ER diagrams, it is represented by an oval with smaller ovals representing its sub-attributes.



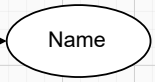
5. Key Attribute

Definition: An attribute that uniquely identifies an entity within an entity set. It serves as the primary key for an entity.
Example: Roll_No for a student, Employee_ID for an employee.
Representation: In ER diagrams, key attributes are represented by an oval with an underline.



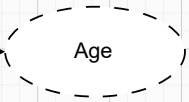
6. Non-Key Attribute

Definition: An attribute that does not uniquely identify an entity but provides additional information about the entity.
Example: Name, Address for a student (these do not uniquely identify a student but provide descriptive information).
Representation: Represented by a single oval without an underline.



7. Derived Attribute

Definition: An attribute whose value can be derived from other attributes within the entity.
Example: Age (can be derived from Date_of_Birth).
Representation: In ER diagrams, it is represented by a dashed oval.

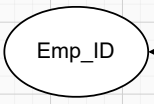


8. Stored Attribute

Definition: An attribute that holds a value stored directly in the database and is used to derive other attributes.
Example: Date_of_Birth (from which Age can be derived).
Representation: Represented by a single oval in the ER diagram (though there is no explicit differentiation from other attributes in representation).

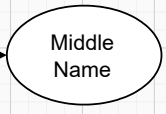
9. Required Attribute

Definition: An attribute that must have a value for each entity in the entity set.
Example: Employee_ID in the Employee entity (it must always be filled).
Representation: In ER diagrams, it is generally assumed that required attributes are present, but no specific notation differentiates them visually.



10. Optional Attribute

Definition: An attribute that may or may not have a value for a particular entity.
Example: Middle_Name for a person (not every person has a middle name).
Representation: No specific notation in ER diagrams for optional attributes, but often indicated in documentation.



Attribute Type	Description	Example	ER Diagram Representation
Single-Valued	Holds a single value for an entity.	`Age`	Single Ellipse
Multi-Valued	Holds multiple values for an entity.	`Phone_No`	Double Ellipse
Simple (Atomic)	Cannot be divided further.	`First_Name`	Single Ellipse
Composite	Can be divided into sub-attributes.	`Address`	Oval with sub-ovals
Key	Uniquely identifies an entity.	`Roll_No`	Oval with underline
Non-Key	Provides descriptive information, not unique.	`Name`, `Address`	Single Ellipse
Derived	Can be derived from other attributes.	`Age` (from DOB)	Dashed Oval
Stored	Attribute directly stored in the database.	`Date_of_Birth`	Single Ellipse
Required	Must have a value for every entity.	`Employee_ID`	Single Ellipse
Optional	May or may not have a value.	`Middle_Name`	No visual difference in ER diagram

Relationship Type and Relationship Set

Relationship Type:

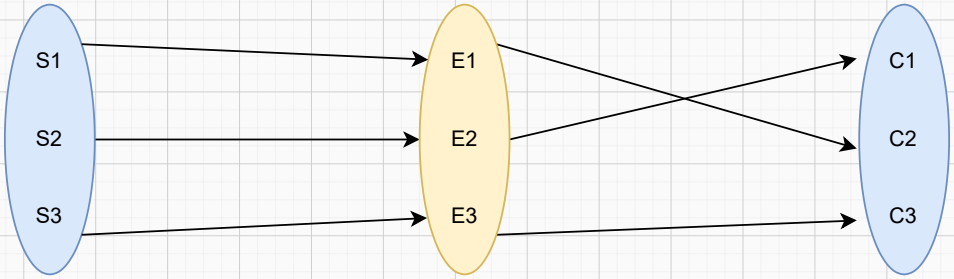
A Relationship Type represents the association or interaction between two or more entity types. It captures how entities are related to each other within a system or database. For example:

- In a university database, the relationship type "Enrolled in" can represent the association between the entity types Student and Course. This type shows that students enroll in courses.
- In an ER Diagram, a relationship type is represented by a diamond shape, with lines connecting it to the entities involved in the relationship.

Relationship Set:

A Relationship Set is a collection of the same type of relationships among entity instances. It is essentially a set of specific associations between instances of the related entity types.

- For example, in the "Enrolled in" relationship set, if S1 is enrolled in C2, S2 in C1, and S3 in C3, this forms the relationship set of the "Enrolled in" relationship type.



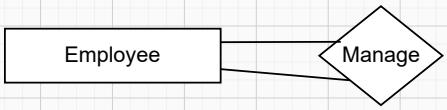
Degree of a Relationship Set

The degree of a relationship set refers to the number of entity sets involved in the relationship. It defines how many entities participate in a given relationship.

1. Unary Relationship (Degree = 1):

A unary relationship involves only one entity set. This is also known as a recursive relationship, where an entity is related to itself.

- **Example:** In an organization, an employee can supervise other employees. The relationship "Supervises" is a unary relationship, as it involves the Employee entity set relating to itself.
- **ER Diagram Representation:** A diamond labeled "Supervises" connects the Employee entity set to itself.



2. Binary Relationship (Degree = 2):

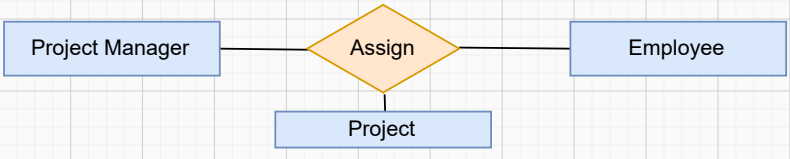
A binary relationship involves two entity sets. It is the most common type of relationship.

- **Example:** A Student enrolling in a Course. This is a binary relationship where the Student entity is associated with the Course entity.
- **ER Diagram Representation:** A diamond connecting the Student entity set with the Course entity set.

3. Ternary Relationship (Degree = 3):

A ternary relationship involves three entity sets.

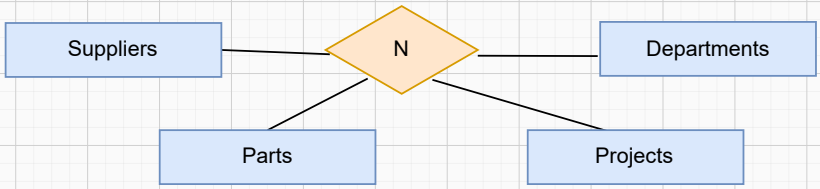
- **Example:** In a project management system, a Project Manager, an Employee, and a Project can be related through the relationship "Assigns." This connects a Project Manager to an Employee for a specific Project.
- **ER Diagram Representation:** A diamond labeled "Assigns" connects the Project Manager, Employee, and Project entity sets.



4. N-ary Relationship (Degree = n):

When there are n entity sets participating in a relationship, it is called an n-ary relationship.

- **Example:** A manufacturing process that involves Suppliers, Parts, Projects, and Departments can form a 4-ary relationship.
- **ER Diagram Representation:** A diamond connecting n entities



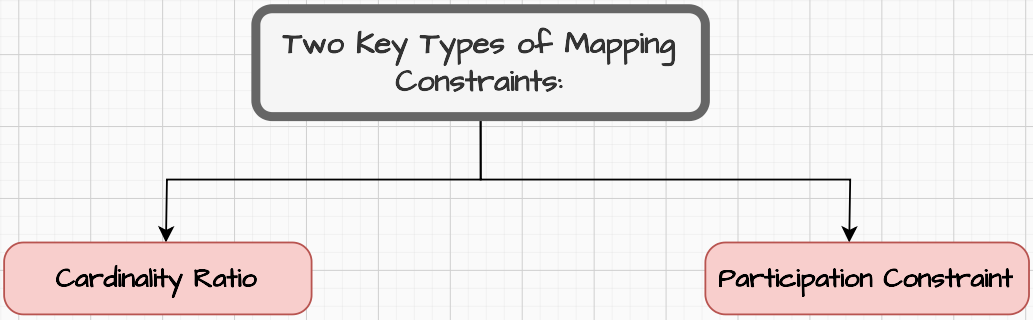
Mapping Constraints

Mapping constraints in DBMS define how many entities in one table (or entity set) can be associated with entities in another table (or entity set). They help us describe the relationship between two tables.

In real life, we have different kinds of relationships between objects or people. For example:

- One person can only have one passport.
- One teacher can teach many classes.
- Many students can enroll in many courses.

In DBMS, we describe these kinds of relationships using Mapping Constraints.



1. Cardinality Ratio (How many?)

This tells us how many entities from one set can be related to entities in another set.

Types of Cardinality Ratios:

1. One-to-One (1:1):

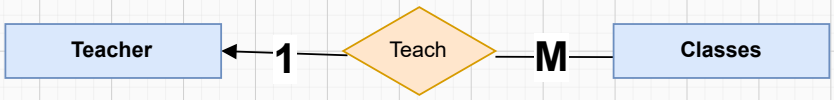
One entity in Set A can be related to only one entity in Set B, and vice versa.

- Example: A person has one passport, and a passport belongs to one person.

2. One-to-Many (1:M):

One entity in Set A can be related to many entities in Set B, but each entity in Set B is related to only one entity in Set A.

- Example: One teacher can teach many classes, but each class is taught by only one teacher.



3. Many-to-One (M:1):

Many entities in Set A are related to one entity in Set B.

- Example: Many students are assigned to one advisor, but each advisor manages many students.

4. Many-to-Many (M:N):

Many entities in Set A can be related to many entities in Set B, and vice versa.

- Example: Many students can enroll in many courses, and each course can have many students.

2. Participation Constraint (Who must participate?)

This tells us whether all entities or only some entities must participate in a relationship.

Types of Participation Constraints:

1. Total Participation:

Every entity in the entity set must participate in the relationship.

- **Example:** In a company, every employee must work in a department.
- **Diagram Representation:** Shown by a double line connecting the entity to the relationship.

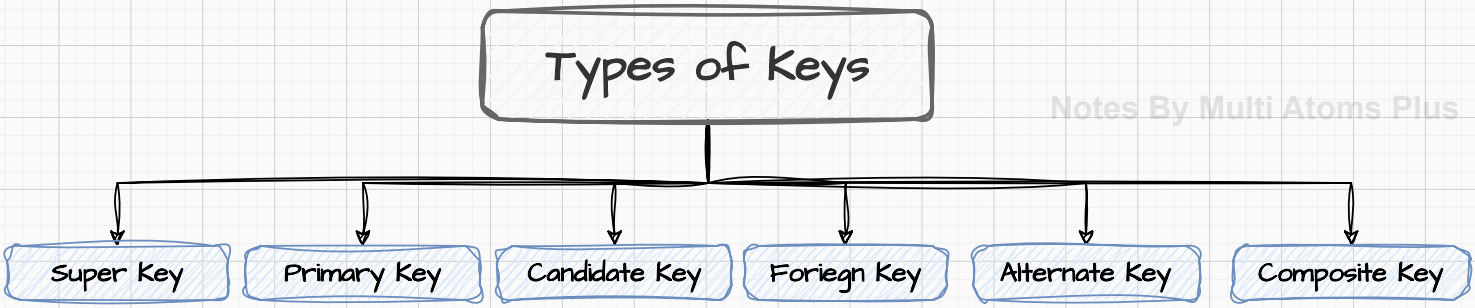
2. Partial Participation:

Some entities in the entity set participate in the relationship, while others do not.

- **Example:** Not every Employee Manages Department
- **Diagram Representation:** Shown by a single line connecting the entity to the relationship.

Keys in DBMS

Keys are **attributes or a combination of attributes** used to **uniquely identify** a tuple (row) in a relation (table). Let's explore the different types of keys:



Importance of Keys

Keys ensure data integrity, uniqueness, and efficient retrieval while establishing relationships between tables, preventing duplication, and maintaining database structure.

1. Super Key

A Super Key is a set of one or more attributes that can uniquely identify a row in a table. It can contain additional attributes that are not necessary for uniqueness.

Example: Let's say we have a Student table:

Roll_No	Name	Phone_Number	Email
101	Alice	9876543210	alice@gmail.com
102	Bob	8765432109	bob@gmail.com
103	Carol	7654321098	carol@gmail.com

Super Keys:

- {Roll_No}
- {Roll_No, Name}
- {Roll_No, Phone_Number}
- {Roll_No, Name, Email}

2. Candidate Key

A Candidate Key is a **minimal Super Key**. It's the smallest set of attributes that can uniquely identify a row.

Example: For the same Student table:

Roll_No	Name	Phone_Number	Email
101	Alice	9876543210	alice@gmail.com
102	Bob	8765432109	bob@gmail.com
103	Carol	7654321098	carol@gmail.com

Candidate Keys:

- **{Roll_No}** (since Roll_No alone uniquely identifies each student)
- **{Phone_Number}** (since every student has a unique phone number)
- {Roll__no, Phone__Number}

3. Primary Key

The Primary Key is a **special Candidate Key chosen by the database designer**. It uniquely identifies each row and cannot be null.

Example: Let's choose **Roll_No** as the Primary Key for the Student table:

Roll_No	Name	Phone_Number	Email
101	Alice	9876543210	alice@gmail.com
102	Bob	8765432109	bob@gmail.com
103	Carol	7654321098	carol@gmail.com

In this table, Roll_No is the Primary Key.

4. Alternate Key

An Alternate Key is any Candidate Key that is **not selected as the Primary Key**.

Example: In the Student table:

Roll_No	Name	Phone_Number	Email
101	Alice	9876543210	alice@gmail.com
102	Bob	8765432109	bob@gmail.com
103	Carol	7654321098	carol@gmail.com

Alternate Key:

- If Roll_No is the Primary Key, then Phone_Number could be an Alternate Key because it also uniquely identifies rows.

5. Composite Key

A Composite Key is made up of **two or more attributes that, together, uniquely identify** a row. Neither attribute alone can uniquely identify a row.

Example: Consider a **Course_Enrollment table** where students enroll in courses. A student can take multiple courses, and a course can have multiple students.

Student_ID	Course_ID	Enrollment_Date
101	CS101	2024-01-15
101	MA101	2024-02-10
102	CS101	2024-01-18

Composite Key:

- {Student_ID, Course_ID} because the combination of Student_ID and Course_ID uniquely identifies each enrollment.

6. Foreign Key

A Foreign Key is a field in one table that references the Primary Key in another table, establishing a relationship between the two.

Example: We have two tables: Students and Enrollments.

StudentID	Name	Age
1	John	21
2	Emma	22
3	Mike	20

EnrollmentID	Course	StudentID (Foreign Key)
101	Math	1
102	Science	2
103	History	1

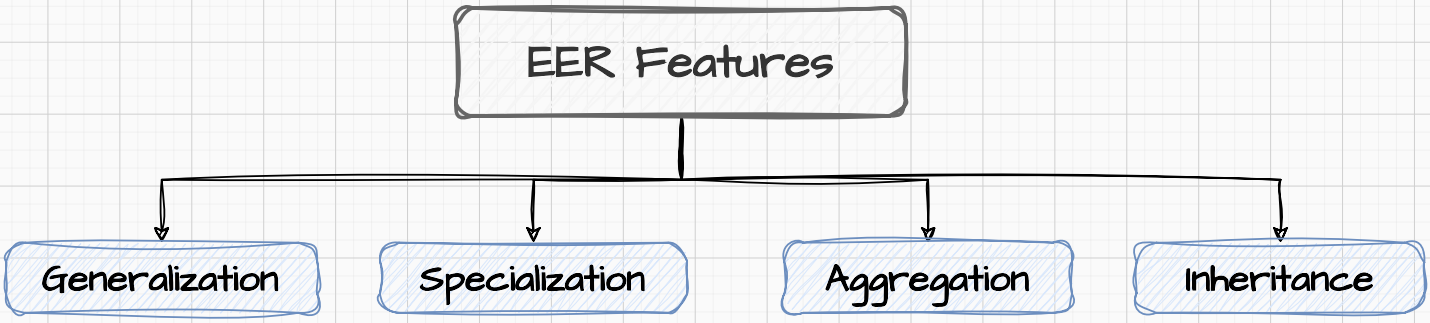
Foreign Key:

- StudentID in the Enrollments table is the Foreign Key.
- It references the StudentID in the Students table, meaning each entry in the Enrollments table must correspond to an existing StudentID in the Students table.

Extended ER Diagram (EER)

Today the complexity of the data is increasing so it becomes more and more difficult to use the traditional ER model for database modeling. To reduce this complexity of modeling we have to make improvements or enhancements to the existing ER model to make it able to handle the complex application in a better way.

The Extended ER Diagram (EER) is an enhancement of the basic Entity-Relationship (ER) model to better represent complex real-world scenarios. It introduces several new concepts and features, which are useful in modeling advanced database requirements. Here are the main features of the EER diagram:

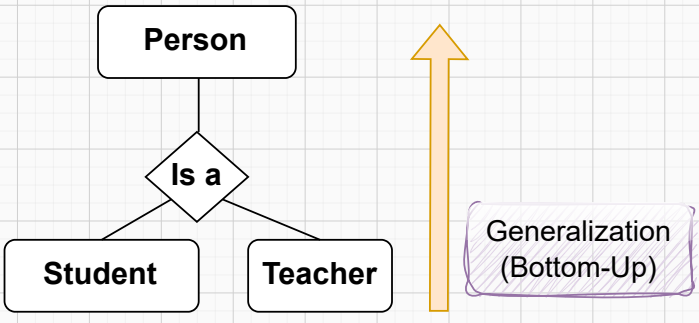


1. Generalization:

It is the process of **combining multiple specific entities into a more general**, higher-level entity. It identifies common features shared by multiple entities and groups them into a single generalized entity to avoid redundancy.

Example: Consider the entities Student and Teacher. Both can be generalized into a higher-level entity called Person because both share common attributes like Name, Address, and Date of Birth.

Why it matters: It simplifies the data model by reducing duplication of shared characteristics across multiple entities.

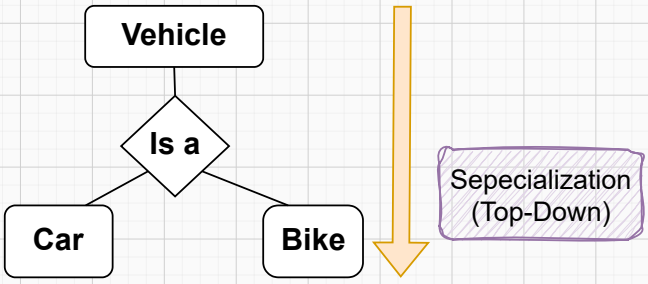


2. Specialization:

It is the **opposite of generalization**. It breaks down a general entity into more specific sub-entities. This allows us to represent entities with additional specific attributes and behaviors that differentiate them from the general entity.

Example: Starting with the general entity Vehicle, we can specialize it into Car and Bike. Both share general attributes like Brand and Model, but Car might have additional attributes like Number of Doors, and Bike might have Type of Handlebar.

Why it matters: It helps in representing more detailed and specific information about certain entities.

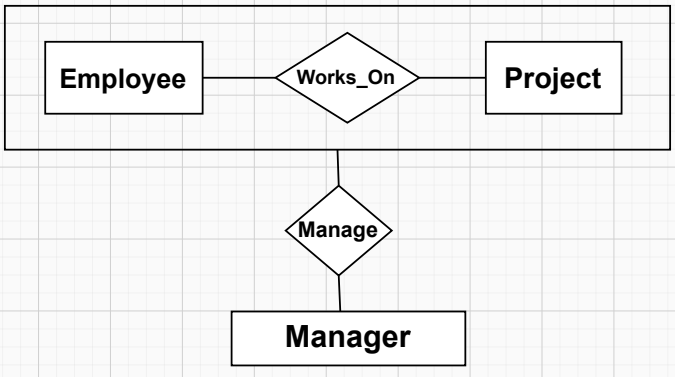


3. Aggregation:

Is used to **represent a relationship as an entity**. It allows you to treat a **relationship between entities as a higher-level entity**, which can then be related to other entities. This is useful when you have complex relationships that involve multiple entities, and you want to simplify or modularize the ER diagram.

Example: Suppose we have entities Employee and Project with a relationship Works_On. Now, if Manager is supervising the relationship between Employee and Project, we can aggregate the Works_On relationship and treat it as a single entity that Manager supervises.

Why it matters: It avoids repeating common information and ensures consistency in the model by allowing shared attributes to be reused across different entities.

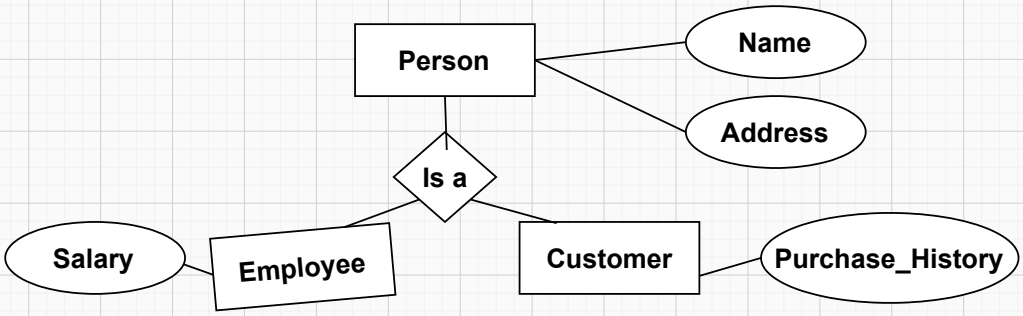


4. Inheritance:

It is a concept borrowed from object-oriented programming and refers to how a specialized entity inherits attributes and relationships from a generalized (parent) entity. The specialized entity can also have its own unique attributes.

Example: If we have a general entity Person, specialized entities like Employee and Customer can inherit common attributes like Name and Address from Person, while also having unique attributes like Salary (for Employee) and Purchase History (for Customer).

Why it matters: It simplifies complex relationships by grouping them and making them manageable.



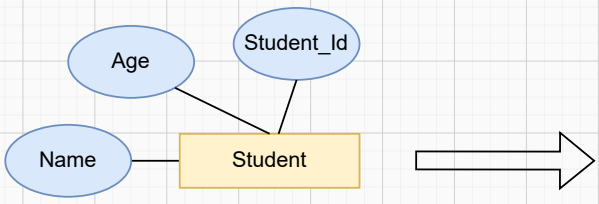
The **reduction of an ER diagram to tables** (also known as mapping an ER diagram to a relational schema) is a process that involves **converting the entities, relationships, and attributes from an ER diagram into tables** in a relational database.

This process ensures that all the information captured in the ER diagram is **properly stored in the database.**

I. Mapping Entities to Tables

- Each entity in the ER diagram becomes a table in the relational database.
- The attributes of the entity become the columns of the table.
- The primary key of the entity becomes the primary key of the table.

Example: If we have an entity **Student with attributes Student_ID, Name, and Age**, it is mapped to a table as follows:

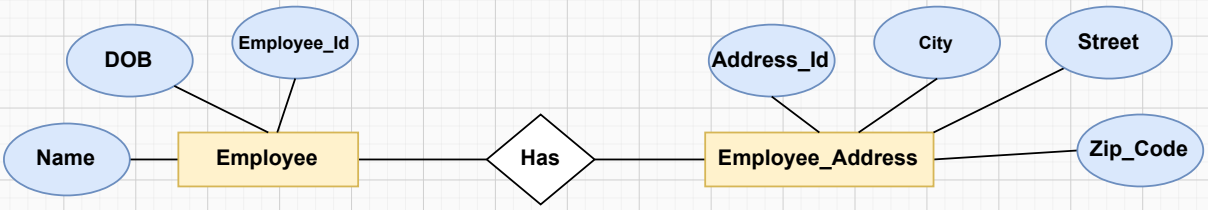


Student_ID (PK)	Name	Age
101	John	20
102	Alice	22

2. Mapping Relationships to Tables

- **One-to-One Relationship**: Can be merged into one table by combining the entities or by placing the foreign key in either of the tables.
- **One-to-Many Relationship**: The foreign key is added to the "many" side of the relationship.
- **Many-to-Many Relationship**: A new table is created with foreign keys from both entities to represent the relationship.

one-to-one relationship



Option 1: Merging the two entities into one table

Employee_ID (PK)	Name	DOB	Address_ID (PK)	Street	City	Zip_Code
101	John	1990-01-01	A101	Oak St	New York	10001
102	Alice	1992-05-12	A102	Pine St	Boston	02115

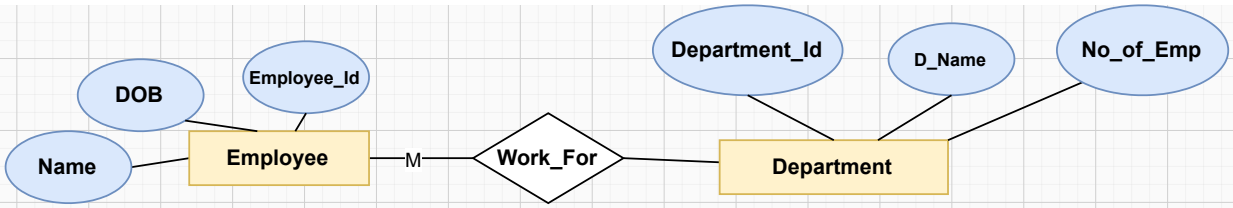
Option 2: Adding a foreign key in one of the tables

Employee_ID (PK)	Name	DOB
101	John	1990-01-01
102	Alice	1992-05-12

Address_ID (PK)	Employee_ID (FK)	Street	City	Zip_Code
A101	101	Oak St	New York	10001
A102	102	Pine St	Boston	02115

one-to-many relationship

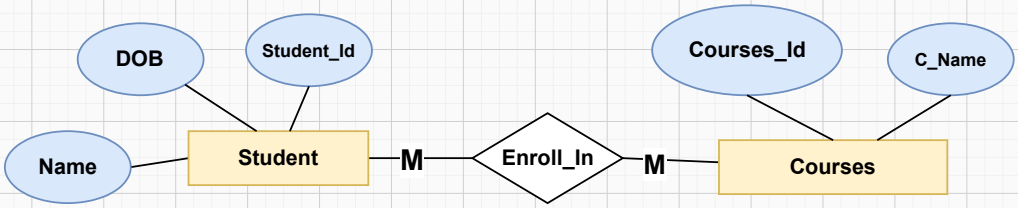
Example: A One-to-Many relationship between **Department** and **Employee** is mapped by adding a foreign key **Department_ID** in the Employee table.



Employee_ID (PK)	Name	Department_ID (FK)
1	John	101
2	Alice	102

Many-to-Many Relationship

Example: A new table **Student_Course** is created to represent the relationship, with **foreign keys from both Student and Course.**



Student_ID (FK)	Course_ID (FK)
101	CSE101
102	CSE101
101	MATH101

3. Mapping Attributes

- **Simple Attributes**: These are directly converted into columns of the corresponding table.
- **Composite Attributes**: Each sub-attribute of a composite attribute is treated as a column in the table.
- **Multi-valued Attributes**: A separate table is created with a foreign key from the original entity and the multi-valued attribute as columns.

Example: If a Student has multiple Phone Numbers, create a new table Student_Phone:

Student_ID (FK)	Phone_Number
101	1234567890
101	9876543210

4. Mapping Weak Entities

- A weak entity is represented as a table, but its primary key is a composite key consisting of its own attributes and the primary key of the related strong entity.

Aktu - 2022-23

(a) A database is being constructed to keep track of the teams and games of a sport league. A team has a number of players, not all of whom participate in each game. It is desired to keep track of players participating in each game for each team, the positions they play in that game and the result of the game.
(i) Design an E-R schema diagram for this application.
(ii) Map the E-R diagram into relational model.

For a database that tracks the **teams, players, games, and participation details in a sports league**, we need to design an ER diagram and map it to a relational model. Let's break this down step by step.

(i) Design an E-R Schema Diagram

Key Entities and Relationships:

1. **Team**: Represents the different teams in the league.
2. **Player**: Represents the players who are part of the teams.
3. **Game**: Represents the games played between teams.
4. **Participation**: Represents the participation of players in specific games, along with their positions.

Attributes

1. **Team**:

- Team_ID (Primary Key)
- Team_Name

2. **Player**:

- Player_ID (Primary Key)
- Player_Name
- Position
- Team_ID (Foreign Key from Team)

3. **Game**:

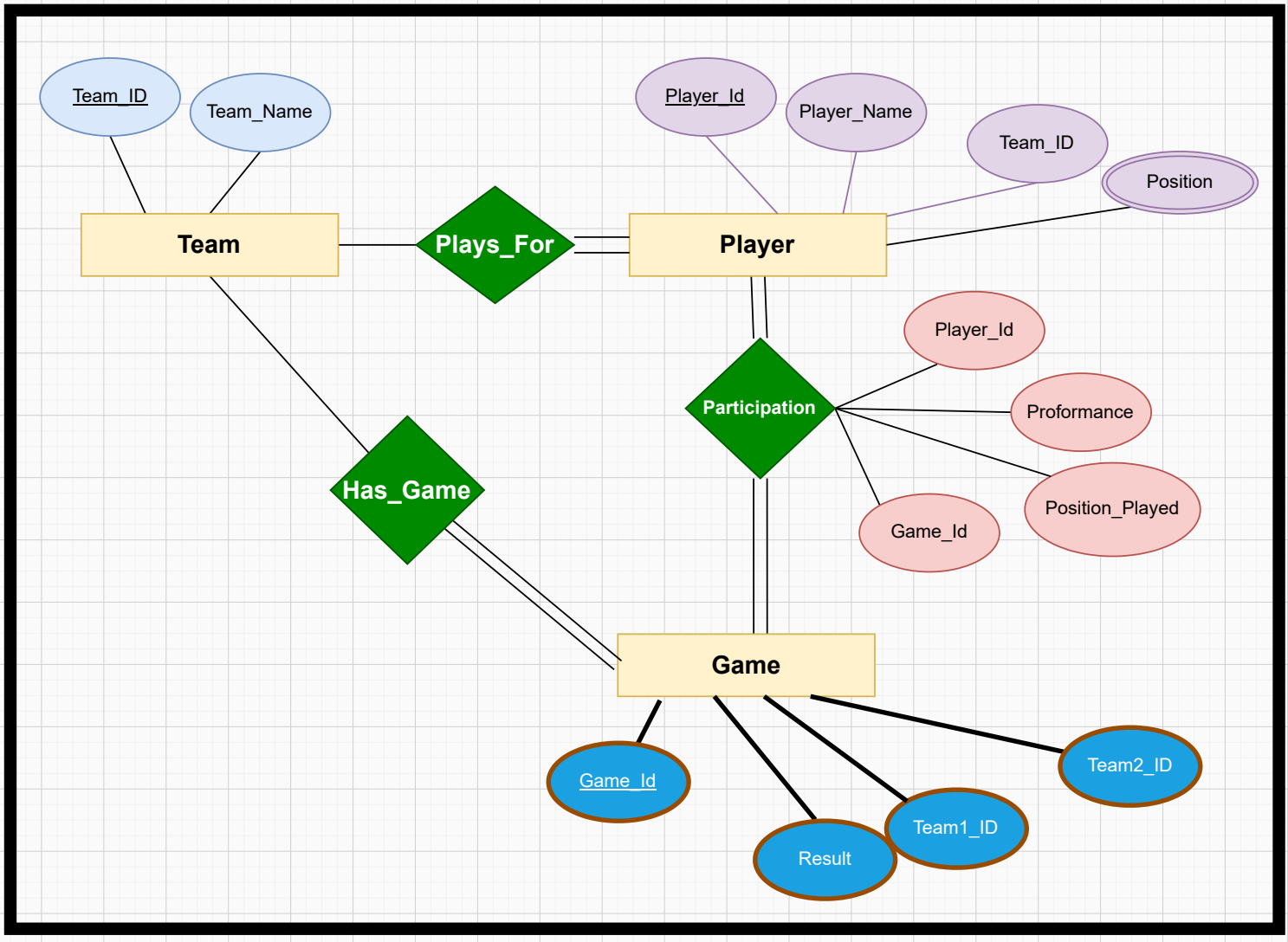
- Game_ID (Primary Key)
- Date
- Team1_ID (Foreign Key from Team)
- Team2_ID (Foreign Key from Team)
- Result (Stores the result of the game)

4. **Participation**:

- Game_ID (Foreign Key from Game)
- Player_ID (Foreign Key from Player)
- Position_Played (Position the player played in the game)
- Performance (Optional: Could store some stats or rating about their performance)

Relationships:

- **Plays_For**: Between Team and Player (one-to-many relationship).
- **Plays**: Between Game and Player (many-to-many relationship, since multiple players participate in a game).
- **Has_Game**: Between Team and Game (a team can participate in multiple games).



(ii) Map the E-R Diagram to a Relational Model

We will convert the entities and relationships from the ER diagram into tables.

1. Team Table:Team(Team_ID , Team_Name)

2. Player Table: (Player_ID , Player_Name ,Position ,Team_ID)

3. Game Table: (Game_ID, Date, Team1_ID, Team2_ID, Result)

4. Participation Table: (Game_ID, Player_ID, Position_Played, Performance)

1. Team Table:Team(Team_ID , Team_Name)

Team_ID (PK)	Team_Name
1	Team A
2	Team B

2. Player Table: (Player_ID , Player_Name ,Position ,Team_ID)

Player_ID (PK)	Player_Name	Position	Team_ID (FK)
101	John	Forward	1
102	Mike	Goalkeeper	1
103	David	Forward	2

3. Game Table: (Game_ID, Date, Team1_ID, Team2_ID, Result)

Game_ID (PK)	Date	Team1_ID (FK)	Team2_ID (FK)	Result
201	2024-10-12	1	2	2-1

4. Participation Table: (Game_ID, Player_ID, Position_Played, Performance)

Game_ID (FK)	Player_ID (FK)	Position_Played	Performance
201	101	Forward	9/10
201	102	Goalkeeper	8/10
201	103	Forward	7/10

Aktu | Database Management System | BCS-501

Unit-2

Syllabus

Relational data Model and Language: Relational Data Model Concepts, Integrity Constraints, Entity Integrity, Referential Integrity, Keys Constraints, Domain Constraints, Relational Algebra, Relational Calculus, Tuple and Domain Calculus. Introduction on SQL: Characteristics of SQL, Advantage of SQL. SQl Data Type and Literals. Types of SQL Commands. SQL Operators and Their Procedure. Tables, Views and Indexes. Queries and Sub Queries. Aggregate Functions. Insert, Update and Delete Operations, Joins, Unions, Intersection, Minus, Cursors, Triggers, Procedures in SQL/PL SQL

Topics Covered

- Relational Data Model Concepts
- Integrity Constraints:
- Relational Algebra
- Relational Calculus
- Tuple and Domain Calculus
- Introduction on SQL:
- Characteristics of SQL
- Advantage of SQL
- SQl Data Type and Literals.
- Types of SQL Commands. SQL Operators
- Tables, Views & Indexes
- Queries & SubQueries
- Aggregate Functions
- Insert, Update & Delete
- SQL Joins - 3PYQ
- Set Operations
- Cursors
- Triggers - 2PYQ
- Procedures in SQL/PL

Relational Data Model Overview

- Developed by E.F. Codd in 1970, the relational model organizes data into tables (relations) with rows and columns.
- Each table represents real-world entities and relationships, making it easier to manage and query than hierarchical or network databases.
- Examples of relational databases include MySQL, Oracle, DB2, and SQL Server.

Key Terminologies

Relation (Table): A table with rows and columns.
Example: A Student table with columns like **Stu_No, S_Name, and Gender**.

Tuple (Row): A single record in a table.
Example: A row in the Student table, like **(10112, 'Rama', 'F')**

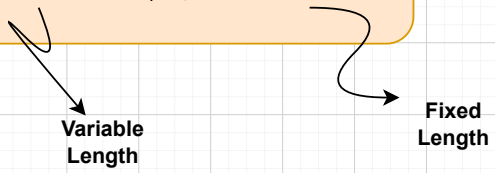
Attribute (Column): A single property of a relation.
Example: **Stu_No, S_Name, Gender**.

Domain: The set of valid values for an attribute.
Example: The domain for Gender is **{M, F}**.

Cardinality: The number of rows in a table.
Example: If Student has 5 rows, its cardinality is **5**.

Degree: The number of columns in a table.
Example: If Student has 5 attributes, its degree is **5**.

Relational Schema: Defines the table's structure, including names and data types for each column.
Example: Student schema could be **Student(Stu_No: INT, S_Name: VARCHAR, Gender: CHAR)**.



Relational Key: Unique identifiers for rows (e.g., **primary key**).
Example: Stu_No uniquely identifies each student.

Properties of Relations

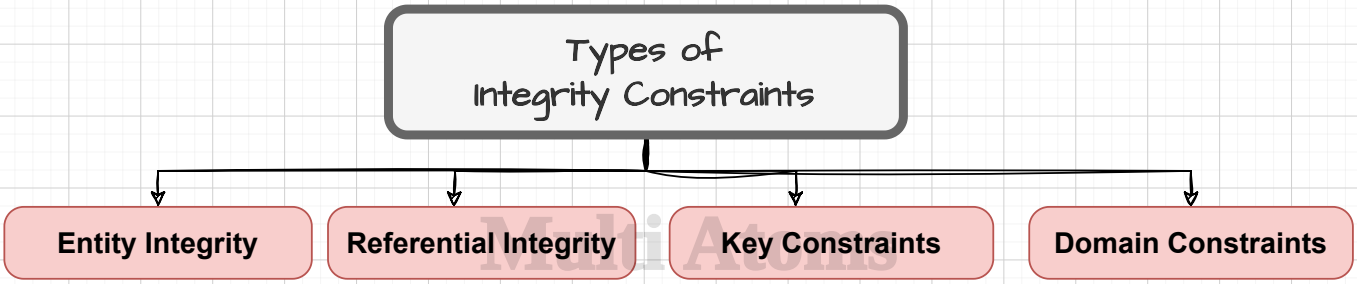
- Each table has a unique name, and each attribute also has a unique name.
- Tables do not have duplicate rows; each row is distinct.
- Each cell contains a single (atomic) value.
- Order of rows and columns is not significant. -> The order of rows and columns doesn't change the meaning of the data.

Basic Operations

1. **Insert**: Adds new rows to a table.
2. **Delete**: Removes rows from a table.
3. **Update**: Changes values in rows.
4. **Retrieve**: Fetches data based on queries.

Integrity Constraints

- Integrity constraints are rules that ensure data remains accurate, consistent, and reliable in a database. They prevent errors and keep the data meaningful.



I. Entity Integrity

It constraints enforce that every table must have a primary key that uniquely identifies each row, and that primary key cannot be NULL.

Product_ID	Product_Name	Price	Stock_Quantity
P1001	Laptop	70000	10
P1002	Phone	30000	50
P1003	Tablet	25000	30
NULL	Smartwatch	15000	20

- **Entity Integrity Constraint**: Product_ID (primary key) must not be NULL.
- **Explanation**: The row with NULL in Product_ID violates the entity integrity constraint, as the primary key must uniquely identify each row.
- **Violation Example**: Every product should have a non-null unique Product_ID.

2. Referential Integrity

It ensures that foreign keys in a table correctly reference primary keys in another related table.

StudentID	Name	Age
101	Alice	20
102	Bob	22
103	Carol	19

EnrollmentID	StudentID	CourseID
1	101	CS101
2	102	CS102
3	104	CS103

- **Referential Integrity Rule**: Each StudentID in Enrollment must exist in the Student table.
- **Explanation**: In the Enrollment table, StudentID acts as a foreign key referencing the primary key StudentID in the Student table.
- **Violation Example**: In the row with EnrollmentID 3, the StudentID 104 does not exist in the Student table, which violates referential integrity since it references a non-existent student.

3. Key Constraints

Key constraints ensure that the value in a column or set of columns must uniquely identify each row. A table can have multiple keys, but one will be designated as the primary key.

StudentID	Name	Age
101	Alice	20
102	Bob	22
103	Carol	19

- **Key Constraint**: Customer_ID must be unique.
- **Explanation**: The row with Customer_ID C001 violates the key constraint because it appears more than once in the table.
- **Violation Example**: The database should not allow two customers with the same Customer_ID.

4. Domain Constraints

Domain constraints define the set of valid values for a given column. They ensure that only permissible data types and values are allowed in the column.

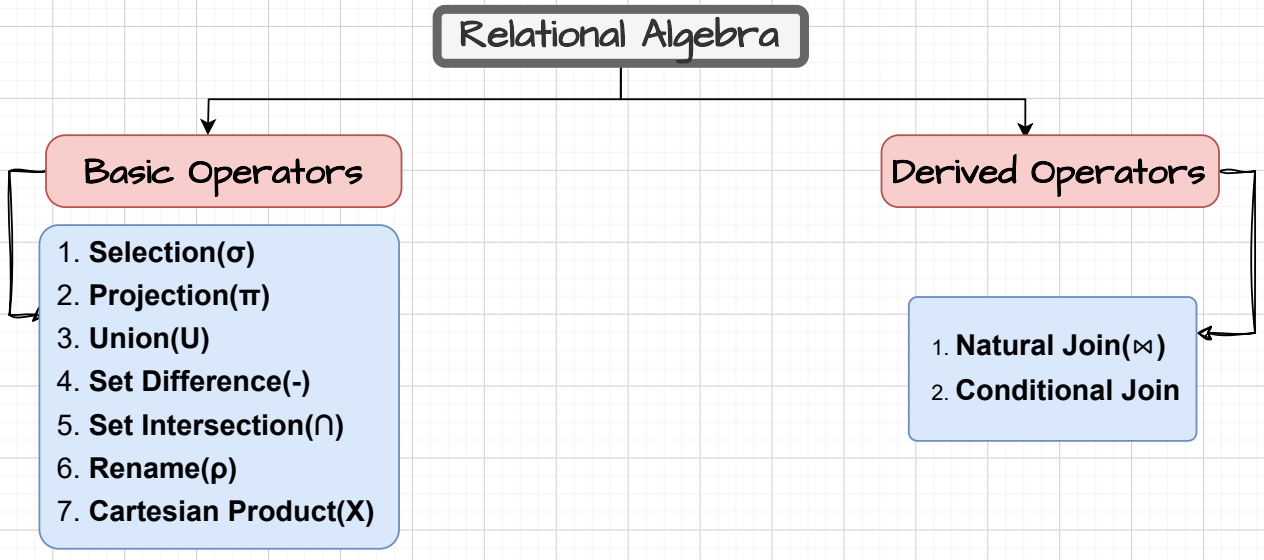
Employee_ID	Name	Department	Salary
E001	Alice	HR	50000
E002	Bob	IT	60000
E003	Charlie	Marketing	45000
E004	David	HR	55000

- **Domain Constraint:** Salary must be a positive integer.
- **Explanation:** The Salary column can only contain numerical values that are positive (no negative values or invalid data types).
- **Violation Example:** If the salary for an employee is entered as -5000, it would violate the domain constraint.

Relational Algebra

Relational Algebra is a **procedural query language** used to work with data in **relational databases**. It uses a **set of operators to retrieve, filter, and combine data, and it provides a foundation for SQL** and other database query languages.

Each operator takes relations (tables) as input and **outputs a new relation**, making it easy to build complex queries by combining operations.



Basic Operators in Relational Algebra

1. Selection (σ)
The Selection operator filters rows based on a specific condition.

Example: Suppose we have a Students table and want to find students who scored above 80.

Student_ID	Name	Age	Score
1	Alice	22	85
2	Bob	20	75
3	Charlie	23	90
4	David	21	65

Query: σ(Score > 80) Students

Result:

Student_ID	Name	Age	Score
1	Alice	22	85
3	Charlie	23	90

2. Projection (π)
The Projection operator retrieves specific columns from a table, discarding others.

Example: To view only the Name and Score columns from a Students table:

Student_ID	Name	Age	Score
1	Alice	22	85
2	Bob	20	75
3	Charlie	23	90
4	David	21	65

Query: π(Name, Score) Students

Result:

Name	Score
Alice	85
Bob	75
Charlie	90
David	65

3. Union (U)
The Union operator combines rows from two tables into a single result, removing any duplicate rows. Both tables must have the same structure (same columns).

Example: Combine two tables Class_A and Class_B to get a single list of students:

Student_ID	Name
1	Alice
2	Bob

Student_ID	Name
3	Charlie
2	Bob

Query: Class_A U Class_B

Result:

Student_ID	Name
1	Alice
2	Bob
3	Charlie

4. Set Difference (-)
The Set Difference operator returns rows that are in one table but not in the other. Both tables must have the same structure.

Example: Find students in Class_A who are not in Class_B

Student_ID	Name
1	Alice
2	Bob

Student_ID	Name
3	Charlie
2	Bob

Query: Class_A - Class_B

Result:

Student_ID	Name
1	Alice

5. Set Intersection (∩)
The Intersection operator returns rows that are common to both tables. Both tables must have the same structure.

Example: Find students who are enrolled in both Class_A and Class_B

Student_ID	Name
1	Alice
2	Bob

Student_ID	Name
3	Charlie
2	Bob

Query: Class_A ∩ Class_B

Result:

Student_ID	Name
2	Bob

6. Rename (ρ)
The Rename operator allows you to give a temporary name to a table or a column, helpful for complex queries.

Example: Rename the Students table to Student_Info:

Query: ρ(Student_Info) Students

Result: The table is temporarily named Student_Info, allowing easier reference in complex queries.

7. Cartesian Product (X)
The Cartesian Product (or Cross Product) operator combines every row of one table with every row of another, resulting in all possible row combinations.

Example: Combine a Students table and a Courses table to show all possible student-course pairings:

Query: Students X Courses

Student_ID	Name
1	Alice
2	Bob

Course_ID	Course_Name
101	Math
102	Physics

Result:

Student_ID	Name	Course_ID	Course_Name
1	Alice	101	Math
1	Alice	102	Physics
2	Bob	101	Math
2	Bob	102	Physics

Additional Operators

1. Natural Join (⋈)
The Natural Join operator combines rows from two tables based on a common attribute with matching values, producing a single table with columns from both tables.

Example: Join Employees and Departments tables based on the Dept_ID column:

Emp_ID	Name	Dept_ID
1	Alice	101
2	Bob	102
3	Charlie	101

Query: Employees ⋈ Departments

Result:

Emp_ID	Name	Dept_ID	Dept_Name
1	Alice	101	HR
3	Charlie	101	HR
2	Bob	102	Sales

2. Conditional Join

The Conditional Join works similarly to the natural join but allows conditions other than equality.

Example: Combine Students and Scores tables where Students.Score is greater than Scores.Min_Score.

Student_ID	Name	Score
1	Alice	85
2	Bob	75
3	Charlie	90
4	David	60

Operation: Perform a conditional join on Students and Scores where Students.Score > Scores.Min_Score.

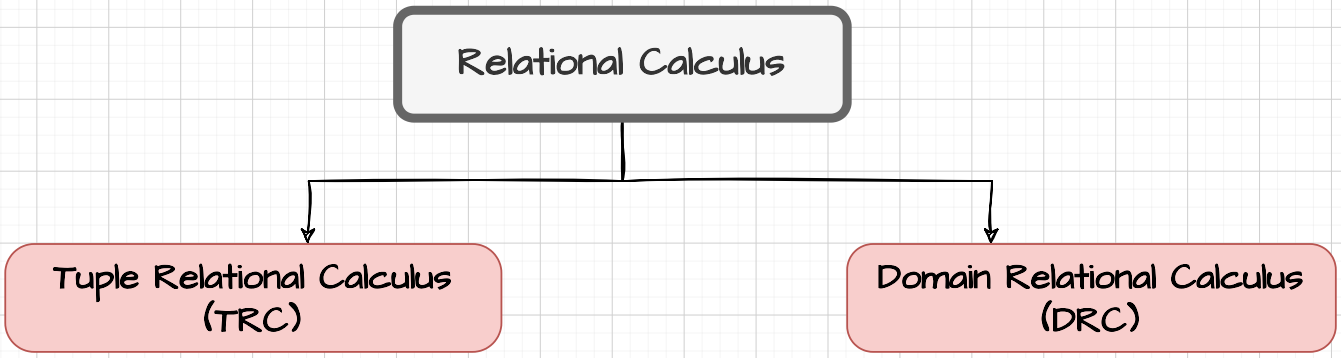
Query: Students ⋈ (Score > Min_Score) Scores

Result:

Student_ID	Name	Score	Min_Score	Level
1	Alice	85	70	Average
1	Alice	85	80	Good
2	Bob	75	70	Average
3	Charlie	90	70	Average
3	Charlie	90	80	Good
3	Charlie	90	90	Excellent

Relational Calculus

Relational Calculus is a **non-procedural query language** used in databases, meaning it focuses on **what data to retrieve rather than the steps to retrieve it**. Unlike Relational Algebra, which specifies a sequence of operations, Relational Calculus describes the desired result.



1. Tuple Relational Calculus (TRC)

In TRC, we use a tuple variable (such as t) that iterates over each row of the table, checking if the row meets the specified condition.

Syntax: { t | Condition(t) }
Here, t represents a tuple, and Condition(t) defines the condition that t must satisfy.

Example: Customer Database:

Customer_id	Name	Zip code
1	Rohit	12345
2	Rahul	13245
3	Rohit	56789
4	Amit	12345

1. Get all data for customers with Zip code 12345:

Query: { t | t ∈ Customer ∧ t.Zipcode = 12345 }

Result:

Customer_id	Name	Zip code
1	Rohit	12345
4	Amit	12345

2. Domain Relational Calculus (DRC)

In DRC, we use domain variables that represent specific columns. DRC queries specify conditions that the values in these columns must satisfy.

Syntax: { <d1, d2, ..., dn> | Condition(d1, d2, ..., dn) }
Here, d1, d2, ..., dn are domain variables, representing individual columns.

Example: Customer Database:

Customer_id	Name	Zip code
1	Rohit	12345
2	Rahul	13245
3	Rohit	56789
4	Amit	12345

1. Get all data for customers with Zip code 12345:

Query: { <d1, d2, d3> | <d1, d2, d3> ∈ Customer ∧ d3 = 12345 }

Result:

Customer_id	Name	Zip code
1	Rohit	12345
4	Amit	12345

Introduction to SQL

SQL (Structured Query Language) is a standard language used to manage and interact with relational databases. It allows users to **create, read, update, and delete data**, making it essential for data-driven applications across various industries.

- **Unified Database Management:** SQL provides a standardized way to handle relational

databases.

- **Core Functions:** It includes tools for data retrieval, manipulation, control, and definition, making it powerful for complex data queries and analysis

Characteristics of SQL

- **Declarative Language:** SQL focuses on what data is needed, not how to get it (Procedural), leaving optimization to the database engine.
- **Set-Based Operations:** SQL processes multiple rows at once, making it efficient for bulk operations.
- **Data Independence:** Users interact with data without needing to know its physical storage.
- **Standardized Syntax:** SQL is standardized across platforms (e.g., MySQL, PostgreSQL, Oracle), promoting compatibility.
- **Relational Model:** SQL uses tables (rows and columns), supporting flexible data relationships.

Advantages of SQL

- **Ease of Use:** SQL has readable, user-friendly commands (e.g., SELECT, INSERT).
- **Portability:** SQL works across platforms and database systems with minimal changes.
- **Scalability:** Suitable for small databases and large, complex datasets.
- **Data Security:** SQL supports access controls to protect data.
- **Integration:** SQL is widely supported in programming and applications, enhancing its utility in tech stacks.

SQL Data Types

Data types in SQL define the kind of data each column can hold. Here are some of the most commonly used data types:

1. Character Types:

VARCHAR(size): Stores variable-length text data. Only occupies the space needed, up to the defined maximum size.

- Example: VARCHAR(50) can store up to 50 characters. It's commonly used for names, addresses, and other variable-length text.

CHAR(size): Stores fixed-length text data, padding shorter values with spaces.

- Example: CHAR(10) will store exactly 10 characters, even if some are empty spaces.

2. Numeric Types:

INT: Stores integer (whole number) values.

- Example: Used for age, count, or other whole number fields without decimals.

DECIMAL(p, s) or NUMERIC(p, s): Stores fixed-point decimal numbers. p is the precision (total digits), and s is the scale (digits after the decimal).

- Example: DECIMAL(5, 2) allows up to 5 digits, with 2 after the decimal (e.g., 123.45).

FLOAT/REAL/DOUBLE: Stores floating-point numbers, useful for scientific and approximate calculations.

3. Date and Time Types:

- **DATE**: Stores date values as `YYYY-MM-DD`.
  - *Example*: `2023-01-15` could represent a birth date.
- **TIME**: Stores time values in `HH:MM:SS` format.
  - *Example*: `12:45:30` could represent the time of an event.
- **TIMESTAMP**: Stores both date and time, usually with time zone information.
  - *Example*: `2023-01-15 12:45:30` could log the exact time of a transaction

4. Boolean Types:

- **BOOLEAN:** Stores true or false values, typically represented as TRUE, FALSE, or NULL (unknown).

Example: Useful for fields like is_active or has_paid to indicate yes/no statuses.

5. Binary Types:

- **BLOB (Binary Large Object):** Stores large binary data, often used for files, images, or multimedia content.
- **BINARY(size) and VARBINARY(size):** Similar to CHAR and VARCHAR, but for binary data.

Literals in SQL

Literals are **fixed values explicitly written in SQL statements**. They include strings, numbers, and dates, and are commonly used in SELECT, INSERT, WHERE, and other SQL statements.

- **String Literals**:
  - Enclosed in single quotes (`'...'`).
  - *Example*: `SELECT * FROM Students WHERE name = 'Alice';`
- **Numeric Literals**:
  - Can be written without quotes, as they represent direct numeric values.
  - *Example*: `SELECT * FROM Products WHERE price > 100;`
- **Date Literals**:
  - Typically enclosed in single quotes and formatted as `YYYY-MM-DD`.
  - *Example*: `SELECT * FROM Orders WHERE order_date = '2023-01-15';`

- **Boolean Literals**:
  - Often used directly as `TRUE` or `FALSE`.
  - *Example*: `SELECT * FROM Users WHERE is_active = TRUE;`

Example Queries Using Data Types and Literals

Creating a Table with Different Data Types:

```
CREATE TABLE Employees (
    employee_id INT,
    name VARCHAR(50),
    birth_date DATE,
    salary DECIMAL(10, 2),
    is_manager BOOLEAN
);
```

Inserting Literal Values into a Table:


```
INSERT INTO Employees (employee_id, name, birth_date, salary, is_manager)
VALUES (1, 'John Doe', '1990-05-15', 75000.00, TRUE);
```

Query Using Literals in **WHERE** Clause:

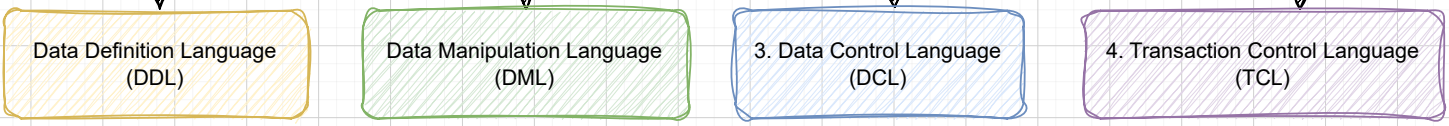
```
SELECT * FROM Employees
WHERE salary > 50000 AND birth_date < '1995-01-01';
```

In DBMS Playlist Check Lec-1.3

Types of SQL Commands

SQL commands are grouped into categories based on their functionality. These categories define how we interact with data, manage the database structure, control access, and handle transactions. Here's an overview of each command type and its typical commands:

Database Languages in DBMS



Category	Full Form	Purpose	Key Commands
DDL	Data Definition Language	Defines or modifies database structures and schema.	`CREATE`, `ALTER`, `DROP`, `TRUNCATE`, `RENAME`, `COMMENT`
DML	Data Manipulation Language	Manages and manipulates data within tables.	`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`
DCL	Data Control Language	Controls access to data and permissions.	`GRANT`, `REVOKE`
TCL	Transaction Control Language	Manages transactions in the database, ensuring data integrity.	`COMMIT`, `ROLLBACK`, `SAVEPOINT`, `SET TRANSACTION`

1. Data Definition Language (DDL)

DDL commands define and modify the structure of a database, including tables, schemas, and indexes. These commands directly affect the database schema and are often used to set up the initial database structure.

Common DDL Commands:

- **CREATE**: Creates new database objects (e.g., tables, indexes).

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(100),
    Department VARCHAR(50)
);
```

- **ALTER**: Modifies existing database objects (e.g., adding a new column to a table)

```
ALTER TABLE Employees ADD Salary DECIMAL(10, 2);
```

- **DROP**: Deletes existing objects from the database (e.g., tables, indexes).

```
DROP TABLE Employees;
```

- **TRUNCATE**: Removes all records from a table without deleting the table itself.

```
TRUNCATE TABLE Employees;
```

- **RENAME**: The RENAME command only changes the name of the object; it does not affect the data or structure of the table.

```
RENAME Employees TO Staff;
```

Purpose: DDL commands help manage the structure of the database by creating, altering, and deleting database objects.

2. Data Manipulation Language

DML commands manipulate data within existing tables. They are the core commands used for querying and modifying records in a database.

Common DML Commands:

- **SELECT**: Retrieves data from the database.

```
SELECT * FROM Employees WHERE Department = 'IT';
```

- **INSERT**: Adds new records to a table.

```
INSERT INTO Employees (EmployeeID, Name, Department) VALUES (1, 'John Doe', 'HR');
```

- **UPDATE**: Modifies existing data in the table.

```
UPDATE Employees SET Department = 'Finance' WHERE EmployeeID = 1;
```

- **DELETE**: Removes data from a table.

```
DELETE FROM Employees WHERE EmployeeID = 1;
```

- **DML commands focus on manipulating the data inside tables, as opposed to DDL, which deals with database structure.**
- **These commands are typically part of day-to-day database operations.**
- **SELECT is the most commonly used DML command for retrieving data.**

3. Data Control Language (DCL)

DCL is used to control access to the database. It includes commands that manage user permissions and access controls.

Common DCL Commands:

- **GRANT**: Gives user access privileges to database objects.

```
GRANT SELECT, INSERT ON Employees TO 'user1';
```


- **REVOKE**: Removes user access privileges.

```
REVOKE INSERT ON Employees FROM 'user1';
```

4. Transaction Control Language (TCL)

TCL commands manage transactions within the database, ensuring that groups of operations are either completely executed or not executed at all.

Common TCL Commands:

- **COMMIT**: Saves all changes made during the transaction.

```
COMMIT;
```

- **ROLLBACK**: Reverts the database to its previous state before the transaction began.

```
ROLLBACK;
```

- **SAVEPOINT**: Sets a point within a transaction to which you can roll back.

```
SAVEPOINT savepoint1;
```

SQL Operators and Their Precedence

SQL uses various operators to perform **operations on data within queries**. These operators fall into categories, each serving specific purposes, such as **calculations, comparisons, or logic-based decisions**. Understanding the order in which SQL evaluates these o**perators is essential for building accurate and efficient queries**. Here's a breakdown:

1. Arithmetic Operators

Arithmetic operators perform mathematical calculations in SQL queries. They are often used to calculate values directly within SELECT statements.

- **+ (Addition):** Adds two numbers.
  - *Example*: `SELECT salary + bonus AS Total_Pay FROM Employees;`
- **- (Subtraction):** Subtracts one number from another.
  - *Example*: `SELECT salary - deduction AS Net_Salary FROM Employees;`
- *** (Multiplication)**: Multiplies two numbers.
  - *Example*: `SELECT price * quantity AS Total_Cost FROM Orders;`
- **/ (Division):** Divides one number by another.
  - *Example*: `SELECT total_cost / items AS Cost_Per_Item FROM Purchases;`

2. Comparison Operators

Comparison operators are used to compare two values and return a boolean result (TRUE or FALSE). They are primarily used in conditions within WHERE clauses to filter data.

- **= (Equal to):** Checks if two values are equal.
  - *Example*: `SELECT * FROM Students WHERE grade = 'A';`
- **<> or != (Not equal to):** Checks if two values are not equal.
  - *Example*: `SELECT * FROM Employees WHERE department <> 'HR';`
- **> (Greater than):** Checks if the left value is greater than the right value.
  - *Example*: `SELECT * FROM Products WHERE price > 100;`

- **< (Less than):** Checks if the left value is less than the right value.
  - *Example*: `SELECT * FROM Orders WHERE quantity < 50;`
- **>= (Greater than or equal to)**: Checks if the left value is greater than or equal to the right value.
  - *Example*: `SELECT * FROM Sales WHERE amount >= 500;`
- **<= (Less than or equal to):** Checks if the left value is less than or equal to the right value.
  - *Example*: `SELECT * FROM Inventory WHERE stock <= 20;`

3. Logical Operators

Logical operators are used to combine multiple conditions in SQL queries, allowing for complex filtering. They are often used in conjunction with comparison operators.

- **AND**: Returns `TRUE` if both conditions are true.
  - *Example*: `SELECT * FROM Students WHERE grade = 'A' AND age > 18;`
- **OR**: Returns `TRUE` if at least one condition is true.
  - *Example*: `SELECT * FROM Employees WHERE department = 'Sales' OR department = 'Marketing';`
- **NOT**: Reverses the result of a condition.
  - *Example*: `SELECT * FROM Products WHERE NOT price < 50;`

4. Precedence of Operators

Operator precedence determines **the order in which SQL evaluates operators in an expression.** Understanding precedence is crucial to ensuring that SQL expressions are interpreted as intended. SQL follows a **standard precedence order**, with **higher precedence operators evaluated first**:

1. **Arithmetic Operators (*, /, +, -)**
2. **Comparison Operators (=, <>, !=, >, <, >=, <=)**
3. **Logical NOT**
4. **Logical AND**
5. **Logical OR**

Note: Use parentheses () to control or override the default precedence when needed.

Examples of Operator Precedence in SQL

Without Parentheses:

```
SELECT * FROM Employees WHERE department = 'Sales' OR department = 'HR' AND salary > 60000;
```

Interpretation: Since AND has higher precedence than OR, this query filters employees in the "HR" department with a salary greater than 60,000 or anyone in "Sales."

With Parentheses:

```
SELECT * FROM Employees WHERE (department = 'Sales' OR department = 'HR') AND salary > 60000;
```

Interpretation: The use of parentheses ensures that the query returns employees in either "Sales" or "HR" who also have a salary above 60,000.

Tables, Views, and Indexes in SQL

Tables, Views, and Indexes are fundamental database structures used to store, manage, and optimize data access in SQL.

Creating Tables: A table is a collection of rows and columns used to organize data in a relational database. To create a table, use the CREATE TABLE command, specifying the columns and data types.

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

Constraints:

PRIMARY KEY: Uniquely identifies each row in the table. Example

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(50)
);
```

FOREIGN KEY: Links two tables to enforce referential integrity. Example:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    EmployeeID INT,
    FOREIGN KEY (EmployeeID)
REFERENCES Employees(EmployeeID)
);
```


Views

A view is a virtual table based on the result of a query. It allows you to present data in a simplified or secure way without modifying the underlying tables.

```
CREATE VIEW view_name AS
SELECT columns
FROM table
WHERE condition;
```

```
CREATE VIEW EmployeeNames
AS
SELECT EmployeeID, Name
FROM Employees;
```

Advantages of Views:

- Security: Restrict access to specific data by only showing selected columns.
- Simplicity: Simplify complex queries by creating reusable views.

Indexes

An index is a database structure that improves the speed of data retrieval on columns frequently used in search conditions, such as in WHERE clauses.

Syntax:
CREATE INDEX index_name ON table_name (column_name);

Ex:
CREATE INDEX idx_employee_name ON Employees (Name);

Benefits of Indexes:

- Speed up search operations.
- Improve the performance of complex queries, especially in large tables.

Queries

Basic SELECT Queries

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


- **SELECT:** Specifies the columns to retrieve.
- **WHERE:** Filters rows based on conditions.
- **ORDER BY:** Sorts the result in ascending (ASC) or descending (DESC) order.

Example:

```
SELECT employee_id, name, department
FROM Employees
WHERE department = 'Sales';
```

```
SELECT employee_id, name, salary
FROM Employees
WHERE salary BETWEEN 30000 AND 50000
ORDER BY salary DESC;
```

Subqueries

A subquery is a query embedded within another query, typically in the WHERE, HAVING, or FROM clause.

Key Points

- Purpose: Subqueries retrieve data to be used by the main query.
- Placement: Used in SELECT, INSERT, UPDATE, DELETE statements.
- Operators: Work with =, <, >, IN, and LIKE.

```
SELECT column_name
FROM table_name
WHERE column_name operator
   (SELECT column_name FROM table_name WHERE ...);
```

Examples:

1.Retrieve Data
Get employees working in the same department as 'John':

```
SELECT NAME, DEPARTMENT
FROM EMPLOYEE
WHERE DEPARTMENT IN (SELECT DEPARTMENT FROM
EMPLOYEE WHERE NAME = 'John');
```

2.Insert Data
Insert customers from New_Customers table into Customers table:


```
INSERT INTO Customers (Name, City)
SELECT Name, City
FROM New_Customers
WHERE City = 'New York';
```

3.Delete Data
Delete products from Products table that are not in stock:

```
DELETE FROM Products
WHERE Product_ID NOT IN (SELECT Product_ID FROM Stock WHERE Quantity > 0);
```

4.Update Data
Increase salaries of employees whose department has an average salary greater than $5000:

```
UPDATE EMPLOYEE
SET SALARY = SALARY * 1.1
WHERE DEPARTMENT IN (SELECT DEPARTMENT FROM EMPLOYEE GROUP BY
DEPARTMENT HAVING AVG(SALARY) > 5000);
```

Aggregate Functions

AKTU-2023-24

Aggregate functions in SQL allow us to perform calculations on multiple rows of data to return a single value or grouped results

1. **COUNT()**: Counts the number of rows.

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

2. **SUM()**: Calculates the sum of numeric values.

```
SELECT SUM(Revenue) AS TotalRevenue
FROM Sales;
```

3. **AVG()**: Finds the average value of a numeric column

```
SELECT AVG(Salary) AS AverageSalary
FROM Employees;
```

4. **MAX()**: Returns the maximum value in a column.

```
SELECT MAX(Salary) AS HighestSalary
FROM Employees;
```


5. **MIN()**: Returns the minimum value in a column.

```
SELECT MIN(Price) AS LowestPrice
FROM Products;
```

GROUP BY Clause

The GROUP BY clause groups rows with the same values in specified columns and performs aggregate functions on each group.

Example: Group sales data by region and calculate the total revenue per region.

```
SELECT Region, SUM(Revenue) AS
TotalRevenue
FROM Sales
GROUP BY Region;
```

input:

Region	Revenue
North	1000
South	1500
East	1200
North	2000
West	1800
South	1300
East	1000
North	1700

output:

Region	TotalRevenue
North	4700
South	2800
East	2200
West	1800

HAVING Clause

The HAVING clause filters groups based on aggregate function results (similar to WHERE but for grouped data).

Example: Find regions with total revenue greater than $10,000.


```
SELECT Region, SUM(Revenue) AS TotalRevenue
FROM Sales
GROUP BY Region
HAVING SUM(Revenue) > 10000;
```

input:

Region	Revenue
North	5000
South	4000
East	3500
West	3000
North	6000
South	7000
East	4500
West	2500

output:

Region	TotalRevenue
North	11000
South	11000

Insert, Update, and Delete Operations

These commands are fundamental for managing data in SQL tables. Here's a concise explanation of their usage, syntax, and examples.

INSERT Operation

Purpose: Adds new rows to a table.

Syntax:

```
INSERT INTO TableName (Column1, Column2, ...)
VALUES (Value1, Value2, ...);
```

Example:

```
INSERT INTO Book (ISBN, Title, Author, Publisher)
VALUES ('9781234567897', 'SQL Basics', 'John Doe',
'TechBooks');
```

UPDATE Operation

Purpose: Modifies existing rows in a table.

Syntax:

```
UPDATE TableName
SET Column1 = Value1, Column2 = Value2, ...
WHERE Condition;
```

Example:

```
UPDATE Book
SET Publisher = 'NewPublisher'
WHERE ISBN = '9781234567897';
```

DELETE Operation

Purpose: Removes rows from a table.

Syntax:

```
DELETE FROM TableName
WHERE Condition;
```

Example:


```
DELETE FROM Book
WHERE Publisher = 'OldPublisher';
```

SQL Joins

Aktu-2021-22, 2022-23, 2023-24

SQL JOINs are used to combine rows from two or more tables based on a related column. Below are different types of JOINs with simplified examples.

1. INNER JOIN

Combines rows from two tables where there is a match in the common column.

Syntax:

```
SELECT table1.column1, table2.column2
FROM table1
INNER JOIN table2
ON table1.common_column =
table2.common_column;
```

Example:

Student Table:

Roll_No	Name	Branch
1	John	CSE
2	Alice	ECE
3	Bob	MECH

Course Table:

Roll_No	Course_Name
1	Math
2	Physics
4	Chemistry

```
SELECT Student.Name, Course.Course_Name
FROM Student
INNER JOIN Course
ON Student.Roll_No = Course.Roll_No;
```

Output:

Name	Course_Name
John	Math
Alice	Physics

2. LEFT JOIN

Returns all rows from the left table and the matching rows from the right table. If there is no match, NULL is returned.

Syntax:

```
SELECT table1.column1, table2.column2
FROM table1
LEFT JOIN table2
ON table1.common_column = table2.common_column;
```

Example:

Student Table:

Roll_No	Name	Branch
1	John	CSE
2	Alice	ECE
3	Bob	MECH

Course Table:

Roll_No	Course_Name
1	Math
2	Physics
4	Chemistry

```
SELECT Student.Name, Course.Course_Name
FROM Student
LEFT JOIN Course
ON Student.Roll_No = Course.Roll_No;
```

Output:

Name	Course_Name
John	Math
Alice	Physics
Bob	NULL

3. RIGHT JOIN

Returns all rows from the right table and the matching rows from the left table. If there is no match, NULL is returned.

Syntax:

```
SELECT table1.column1, table2.column2
FROM table1
RIGHT JOIN table2
ON table1.common_column = table2.common_column;
```

Example:

Student Table:

Roll_No	Name	Branch
1	John	CSE
2	Alice	ECE
3	Bob	MECH

Course Table:

Roll_No	Course_Name
1	Math
2	Physics
4	Chemistry

```
SELECT Student.Name, Course.Course_Name
FROM Student
RIGHT JOIN Course
ON Student.Roll_No = Course.Roll_No;
```

Output:

Name	Course_Name
John	Math
Alice	Physics
NULL	Chemistry

4. FULL JOIN

Combines the results of both LEFT JOIN and RIGHT JOIN. Returns all rows from both tables, with NULL where there is no match.

```
SELECT Student.Name, Course.Course_Name
FROM Student
FULL JOIN Course
ON Student.Roll_No = Course.Roll_No;
```

Output:

Name	Course_Name
John	Math
Alice	Physics
Bob	NULL
NULL	Chemistry

5. CROSS JOIN

Returns the Cartesian product of two tables, combining each row from the first table with every row from the second table.

Syntax:

```
SELECT table1.column1, table2.column2
FROM table1
CROSS JOIN table2
```


Example:

Student Table:

Roll_No	Name
1	John
2	Alice

Course Table:

Course_ID	Course_Name
101	Math
102	Physics

```
SELECT Student.Name, Course.Course_Name
FROM Student
CROSS JOIN Course;
```

Output:

Name	Course_Name
John	Math
John	Physics
Alice	Math
Alice	Physics

6. Natural Join

A Natural Join joins tables based on columns with the same names and data types in both tables.
It removes duplicate columns and keeps only one copy of the common column in the result.

Example:

Employee Table:

Emp_ID	Name	Dept_ID
1	Alice	101
2	Bob	102
3	Carol	103

Department Table:

Dept_ID	Dept_Name
101	IT
102	HR
104	Finance

```
SELECT *
FROM Employee
NATURAL JOIN Department;
```

Output:

Emp_ID	Name	Dept_ID	Dept_Name
1	Alice	101	IT
2	Bob	102	HR

Set Operations in SQL

Set operations allow combining the results of two or more queries. They work on the principle of sets in mathematics and require the queries to have the same number of columns with compatible data types.

1. UNION

- Combines the result sets of two queries.
- Removes duplicate rows by default.
- Use UNION ALL to include duplicates.

Syntax:

```
SELECT column1, column2
FROM table1
UNION
SELECT column1, column2
FROM table2;
```

Example:

Emp-1 Table:

Emp_ID	Name
1	Alice
2	Bob

Emp-2 Table:

Emp_ID	Name
2	Bob
3	Carol

```
SELECT Emp_ID, Name
FROM Employee_Table1
UNION
SELECT Emp_ID, Name
FROM Employee_Table2;
```

Output:

Emp_ID	Name
1	Alice
2	Bob
3	Carol

2. INTERSECT

- Retrieves common rows from the result sets of two queries.

Syntax:

```
SELECT column1, column2
FROM table1
INTERSECT
SELECT column1, column2
FROM table2;
```

Example:

Emp-1 Table:

Emp_ID	Name
1	Alice
2	Bob

Emp-2 Table:

Emp_ID	Name
2	Bob
3	Carol

```
SELECT Emp_ID, Name
FROM Employee_Table1
INTERSECT
SELECT Emp_ID, Name
FROM Employee_Table2;
```

Output:

Emp_ID	Name
2	Bob

3. MINUS

- Retrieves rows from the first query that are not present in the second query.

Syntax:

```
SELECT column1, column2
FROM table1
MINUS
SELECT column1, column2
FROM table2;
```

Example:

Emp-1 Table:

Emp_ID	Name
1	Alice
2	Bob

Emp-2 Table:

Emp_ID	Name
2	Bob
3	Carol

```
SELECT Emp_ID, Name
FROM Employee_Table1
MINUS
SELECT Emp_ID, Name
FROM Employee_Table2;
```

Output:

Emp_ID	Name
1	Alice

What is a Cursor?

A cursor is a temporary memory structure or workspace allocated by the database server to handle row-by-row data processing. It is used during Data Manipulation Language (DML) operations such as INSERT, UPDATE, DELETE, and SELECT. Cursors facilitate operations on a result set one row at a time, which is useful when row-specific logic is required.

Key Features:

- **Temporary Storage:** Acts as a buffer to hold query results.
- **Row-by-Row Processing:** Unlike set-based operations, cursors enable operations on individual rows.

Types of Cursors:

1. Implicit Cursors:

- Automatically created by the database when you execute simple queries like INSERT, UPDATE, or DELETE.
- You don't need to define or control it.

2. Explicit Cursors:

- Created by you (the user) when you need more control, like going through rows one by one and performing some actions.
- You manually declare, open, fetch data, and close it.

How Does a Cursor Work?

1. **Declare**: You define the cursor and the query that gives the list of rows to process.
2. **Open:** The cursor is prepared, and the query runs.
3. **Fetch:** The cursor moves to the first row and processes it. Then it moves to the next row, and so on.
4. **Close**: Once all rows are processed, the cursor is closed to free resources.

What is Triggers?

Aktu-2022-23, 2021-22

A trigger is a special kind of stored procedure that is executed automatically when a specific event (such as an insert, update, or delete) occurs on a table or view. Triggers are used to automate tasks, enforce business rules, maintain data integrity, audit changes, and replicate data.

Types of Triggers

1.Data Manipulation Language (DML) Triggers: DML triggers are executed when a DML operation like INSERT, UPDATE, or DELETE is performed on a table or view.

-> AFTER Triggers:

These triggers are executed after the DML statement completes but before the changes are committed to the database.

-> INSTEAD OF Triggers:

These triggers replace the triggering DML action (INSERT, UPDATE, DELETE) with a different action defined in the trigger body.

2.Data Definition Language (DDL) Triggers: DDL triggers are fired when a DDL operation like CREATE, ALTER, DROP, GRANT, REVOKE, or UPDATE STATISTICS is executed.

-> DATABASE-scoped DDL triggers

fire when DDL statements that affect the database schema (like creating tables or altering procedures) are executed.

-> SERVER-scoped DDL triggers

fire when server-level changes are made (e.g., creating or dropping databases, managing logins).

3. LOGON Triggers:

- LOGON triggers are automatically executed after a successful user login but before the user session is established.
- If authentication fails, the LOGON trigger is not executed.

CLR Triggers:

- CLR (Common Language Runtime) triggers are written using .NET languages such as C# or VB.NET.
- These triggers are useful for scenarios that require heavy computation or interaction with objects outside SQL, such as web services or external applications.

Example: Automatically Update Stock Levels After a Sale

Scenario:

We have two tables:

1. Products: Stores product details and stock levels.
2. Sales: Stores sales transactions.

When a sale is recorded in the Sales table, we want the stock quantity in the Products table to update automatically.

ProductID	ProductName	Stock
1	Laptop	50
2	Phone	100

SaleID	ProductID	Quantity
1	1	2

```
CREATE TRIGGER UpdateStockAfterSale
AFTER INSERT ON Sales
FOR EACH ROW
BEGIN
    UPDATE Products
    SET Stock = Stock - NEW.Quantity
    WHERE ProductID = NEW.ProductID;
END;
```

How It Works:

1. A new sale is added to the Sales table (e.g., 2 laptops sold).
2. The trigger automatically reduces the stock in the Products table.

Products Table Before:

ProductID	ProductName	Stock
1	Laptop	50
2	Phone	100

Insert a sale:

```
INSERT INTO Sales (SaleID, ProductID, Quantity)
VALUES (2, 1, 2);
```

Products Table After:

ProductID	ProductName	Stock
1	Laptop	48
2	Phone	100

Procedures in PL/SQL

PL/SQL (Procedural Language/Structured Query Language) is a block-structured programming language that allows combining SQL commands with procedural constructs like loops and conditions. A stored procedure in PL/SQL is a precompiled block of SQL statements stored in the database catalog for reuse.

A procedure serves as a reusable program that performs a specific task, which can be invoked by:

- Triggers
- Other procedures

When a procedure is executed, all its statements are sent to the Oracle engine at once, improving processing speed and reducing network traffic.

Advantages of Procedures in PL/SQL

- Improved Performance
- Reduced Network Traffic
- Code Reusability

Disadvantages of Procedures in PL/SQL

- Increased Memory Usage
- Complexity for Developers
- Lack of Debugging in MySQL

Example of a Stored Procedure in PL/SQL

Let's create a stored procedure to calculate the total salary of employees in a specific department.

EMP_ID	NAME	DEPT_ID	SALARY
1	John	101	5000
2	Alice	102	6000
3	Bob	101	5500
4	Charlie	103	7000

Stored Procedure

```
CREATE PROCEDURE CalculateTotalSalary (
    dept_no IN NUMBER,
    total_salary OUT NUMBER
)
AS
BEGIN
    -- Calculate the total salary of employees in the given department
    SELECT SUM(SALARY)
    INTO total_salary
    FROM Employee
    WHERE DEPT_ID = dept_no;
END;
/
```

Calling the Procedure

```
DECLARE
    dept_id NUMBER := 101; -- Input: Department ID
    total_sal NUMBER;      -- Output: Total Salary
BEGIN
    -- Call the procedure
    CalculateTotalSalary(dept_id, total_sal);
    -- Display the result
    DBMS_OUTPUT.PUT_LINE('Total Salary for Department ' || dept_id || ': ' || total_sal);
END;
/
```


Output:

```
Total Salary for Department 101: 10500
```

Aktu

Database Management System

BCS-501

Unit-3

Syllabus

Data Base Design & Normalization: Functional dependencies, normal forms, first, second, 8 third normal forms, BCNF, inclusion dependence, loss less join decompositions, normalization using FD, MVD, and JDs, alternative approaches to database design

Topics Covered

- **Introduction to Database Design & Normalization**
- **What is a Functional Dependency? - (Aktu 23-24)**
- **Armstrong's Axioms - (Aktu 21-22)**
- **Types of Functional Dependencies**
- **Canonical Cover in FDs -(Aktu 23-24)**
- **Normalization and Normal Forms**
- **Step-by-Step Explanation of Normal Forms (1st,2nd,3rd & BCNF)**
- **Prime & Non Prime Attributes**
- **How to Find Candidate Key**
- **Inclusion Dependency**
- **Lossless Join Decomposition**
- **Multivalued Dependencies (MVDs)**
- **Join Dependency (JD)**
- **4NF & 5NF**
- **Alternate Approaches to Database Design**

Introduction to Database Design & Normalization

What is Database Design?

It is the process of organizing data into structured formats that ensure consistency, efficiency, and accuracy. It involves defining:

- **Schema**: The logical structure of the database, including tables, columns, and relationships.
- **Constraints**: Rules to maintain data integrity.

- **Relationships**: How data in one table is related to data in another (e.g., primary and foreign keys).

What is Normalization?

It is a systematic process in database design to organize data into multiple tables to:

- **Reduce redundancy**: Avoid duplicate data storage.
- **Ensure data integrity**: Prevent anomalies like inconsistent updates or deletions.

It involves dividing large tables into smaller, well-structured tables while maintaining relationships between them. This process adheres to a set of rules, called normal forms, to achieve an efficient and error-free database design.

Importance of Normalization

- Eliminates Redundancy
- Improves Data Integrity
- Facilitates Scalability

Normalized databases are easier to manage and scale as the amount of data grows.

Example

Unnormalized Table:

StudentID	Name	Course	Instructor	InstructorPhone
101	Alice	Math	Dr. Smith	9876543210
101	Alice	Physics	Dr. Brown	8765432109

Problems:

- **Redundancy:** "Alice" and her details are repeated.
- **Update Anomaly:** Changing "Dr. Smith's" phone number requires multiple updates.
- **Delete Anomaly:** Deleting "Math" for Alice might remove her instructor details.

Normalized Table:

Table 1 (Students):

StudentID	Name
101	Alice

CourseID	Course	Instructor
C1	Math	Dr. Smith
C2	Physics	Dr. Brown

Table 2 (Courses):

Table 3 (Instructors):

Instructor	Phone
Dr. Smith	9876543210
Dr. Brown	8765432109

Table 4 (Enrollments):

StudentID	CourseID
101	C1
101	C2

What is a Functional Dependency?

AKTU- 2023-24

A Functional Dependency (FD) is a rule that defines how one column in a table determines the value of another column.

Denoted as X → Y, where:

- X (Determinant): The column(s) whose values decide another column.
- Y (Dependent): The column whose value depends on X.

StudentID	Name	Course
101	Alice	Math
102	Bob	Physics
103	Charlie	Chemistry

- **StudentID → Name**: If you know the StudentID, you can find the Name.
- **Course → Name**: This is not true, as the same course could be taken by multiple students.

Why is Functional Dependency Important?

- **Organize Data:** By identifying relationships between columns, we can design better databases.
- **Remove Redundancy:** Prevents storing unnecessary repeated data.
- **Avoid Errors:** Reduces issues like incorrect or inconsistent data.

AKTU- 2021-22

Armstrong's Axioms / Inference Rules for Functional Dependencies

Armstrong's axioms are basic rules to derive all possible functional dependencies (FDs) from a given set of FDs. There are three main axioms:

1. Reflexivity

- **Rule:** If a set of attributes (Y) is a subset of another set of attributes (X), then X → Y holds true.
- **Meaning:** Any attribute or group of attributes determines itself or its subset.

Roll_No	Name	Dept
101	Alice	Science
102	Bob	Arts

- **{Roll_No, Name} → Name**: Since Name is a part (subset) of {Roll_No, Name}, this FD is valid.
- **Roll_No → Roll_No**: Any column always determines itself.

2. Augmentation

- **Rule:** If X → Y is a valid FD, then X + Z → Y + Z is also valid.
- **Meaning:** Adding the same attribute(s) (Z) to both sides of a valid FD doesn't change the dependency.

Roll_No	Name	Dept	Building
101	Alice	Science	A1
102	Bob	Arts	A2

- **Roll_No → Name:** If you know Roll_No, you can determine Name.
- **By augmentation, Roll_No + Dept → Name + Dept is also valid.**

3. Transitivity

- **Rule**: If X → Y and Y → Z are valid FDs, then X → Z is also valid.
- **Meaning:** If one attribute depends on a second, and the second depends on a third, then the first indirectly determines the third.

Roll_No	Dept	Building
101	Science	A1
102	Arts	A2

- **Roll_No → Dept: Knowing Roll_No gives the Dept.**
- **Dept → Building: Knowing Dept gives the Building.**
- **By transitivity, Roll_No → Building is valid.**

Other Properties Derived from Axioms:

- **Union: If X → Y and X → Z, then X → YZ.**
- **Decomposition: If X → YZ, then X → Y and X → Z.**

Types of Functional Dependencies

1. Trivial Functional Dependency

A dependency is trivial if the dependent column is already part of the determinant.

StudentID	Name
101	Alice
102	Bob

- **{StudentID, Name} → Name:** This is trivial because Name is already part of the determinant {StudentID, Name}.
- **StudentID → StudentID:** This is also trivial.

2. Non-Trivial Functional Dependency

A dependency is non-trivial if the dependent column is not part of the determinant.

StudentID	Name	Course
101	Alice	Math
102	Bob	Physics

- **StudentID → Name:** Knowing StudentID gives us the Name, which is not part of StudentID.

3. Multivalued Functional Dependency

It occurs when one column determines multiple columns that are independent of each other.

StudentID	Hobby	Skill
101	Painting	Python
101	Dancing	Python

- **StudentID → {Hobby, Skill}, but Hobby and Skill do not depend on each other.**

4. Transitive Functional Dependency

A dependency is transitive if one column depends on another through a third column.

StudentID	Class	Teacher
101	10A	Mr. Smith
102	10B	Ms. Johnson

- **StudentID → Class and Class → Teacher, so StudentID → Teacher is transitive.**

5. Partial Functional Dependency

It occurs when a non-key attribute depends only on part of a composite key, not the whole key.

CourseID	Semester	Credits
C101	Fall	3
C102	Spring	4

- **If {CourseID, Semester} is the composite key, but only CourseID → Credits, it's a partial dependency.**

6. Fully Functional Dependency

A dependency is fully functional when a column depends on the entire composite key, not just part of it.

OrderID	ProductID	Quantity
O101	P001	10
O102	P002	5

- **{OrderID, ProductID} → Quantity** depends on the full composite key, so it's fully functional.

Canonical Cover in Functional Dependencies

AKTU- 2023-24

It is the simplified version of a set of Functional Dependencies (FDs). It removes unnecessary dependencies and makes the FD set smaller without changing its meaning. This helps in tasks like normalization, decomposition, and checking dependency preservation.

Steps to Calculate Canonical Cover

Let's break it down step by step with a simple example:

Given: Functional Dependencies (FDs)

We have a relation $R(A, B, C, D)$ with the following FDs:

1. $A \rightarrow BC$
2. $B \rightarrow C$
3. $A \rightarrow B$
4. $AB \rightarrow C$

Step 1: Split RHS of FDs

If any FD has multiple attributes on the right-hand side, split it into multiple FDs with one attribute on the right side.

- $A \rightarrow BC$ becomes:
  - $A \rightarrow B$
  - $A \rightarrow C$

Now the updated FDs are:

1. $A \rightarrow B$
2. $A \rightarrow C$
3. $B \rightarrow C$
4. $A \rightarrow B$
1. 5. $AB \rightarrow C$

Step 2: Remove Redundant FDs

Check if any FD is unnecessary (i.e., it can be derived from the others).

- **FD $A \rightarrow B$ is repeated, so remove the duplicate.**

Updated FDs:

1. $A \rightarrow B$
2. $A \rightarrow C$
3. $B \rightarrow C$
4. $AB \rightarrow C$

Step 3: Remove Redundant Attributes from LHS

Check if any attribute in the left-hand side of an FD is unnecessary.

- **Let's analyze $AB \rightarrow C$:**
  - **Test if $A \rightarrow C$ can already imply $AB \rightarrow C$.**
  - **Yes, because $A \rightarrow C$ is already present.**

So, $AB \rightarrow C$ is redundant. Remove it.

Updated FDs:

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

Step 4: Final Canonical Cover
The final set of simplified FDs is:

- $A \rightarrow B$
- $A \rightarrow C$
- $B \rightarrow C$

Normalization and Normal Forms

Normalization is the process of organizing data in a database to reduce redundancy and improve data integrity. It involves dividing a database into smaller tables and defining relationships between them to ensure consistency and eliminate anomalies like update, delete, and insert anomalies.

Importance of Normalization:

- **Minimizes Redundancy:** Reduces duplicate data.
- **Improves Data Integrity:** Ensures that data is accurate and consistent.
- **Simplifies Maintenance:** Makes it easier to update data.
- **Optimizes Queries:** Improves database performance by reducing unnecessary joins.

Step-by-Step Explanation of Normal Forms

1.First Normal Form (1NF):

A relation is in 1NF if:

- Each column contains atomic (indivisible) values.
- Each record is unique.

Example: Improper 1NF Table

StudentID	Name	Courses
1	Alice	Math, Physics
2	Bob	Chemistry

Solution (Convert to 1NF):

StudentID	Name	Course
1	Alice	Math
1	Alice	Physics
2	Bob	Chemistry

Prime and Non-Prime Attributes

Prime Attributes

- A prime attribute is an attribute that is part of at least one candidate key of a relation.
- **Candidate Key**: A minimal set of attributes that can uniquely identify every tuple (row) in a relation.

Non-Prime Attributes

- A non-prime attribute is an attribute that is not part of any candidate key.

Example

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

1. $AB \rightarrow C$
2. $C \rightarrow D$

Step 1: Find the Candidate Key(s):

- AB is a **candidate key** because it can uniquely identify all attributes in the relation.

Step 2: Identify Prime and Non-Prime Attributes:

- **Prime Attributes:** A, B (because they are part of the candidate key AB).
- **Non-Prime Attributes:** C, D (because they are not part of any candidate key).

How to Find Candidate Key

Given Relation:

$R(A, B, C, D)$

Functional Dependencies:

1. $AB \rightarrow C$
2. $C \rightarrow D$

Step 1: List All Attributes

$R = \{A, B, C, D\}$

Step 2: Identify Closures

- $AB^+ = \{A, B, C, D\}$: AB **determines all attributes.**
- $A^+ = \{A\}$: A **does not determine all attributes.**
- $B^+ = \{B\}$: B **does not determine all attributes.**
- $C^+ = \{C, D\}$: C **determines** D**, but not** A **or** B**.**

Step 3: Check Superkeys

- AB **is a superkey because** AB^+ **contains all attributes.**

Step 4: Identify Candidate Key

- AB **is the candidate key because it is minimal and can determine all attributes.**

Tips

1. Always start by calculating closures.
2. Look for minimal sets of attributes that can uniquely identify all attributes.
3. Eliminate redundancy to identify candidate keys.

2.Second Normal Form (2NF):

A table is in Second Normal Form (2NF) if:

- It is in First Normal Form (1NF)
- It has no partial dependencies, meaning no **non-prime attribute depends on a part of a composite key.**

Steps to Achieve 2NF

1. Start with a 1NF table.

Ensure no multi-valued or repeating attributes exist.

2. Check for Partial Dependencies.
Identify if any non-prime attribute depends on only a portion of a composite key.

3. Remove Partial Dependencies.
Create new tables to separate these dependencies.

4. Reorganize the Table.
Ensure that every non-prime attribute depends on the whole candidate key.

Example
Table: Student_Subject

Roll_No	Subject_Code	Subject_Name	Student_Name
1	MATH01	Math	Alice
2	PHYS01	Physics	Bob
1	PHYS01	Physics	Alice
2	MATH01	Math	Bob

Candidate Key: $\{Roll\_No, Subject\_Code\}$
Non-Prime Attributes: $Subject\_Name, Student\_Name$

Step 1: Check for Partial Dependencies

- **Subject_Name** depends only on **Subject_Code** (a part of the composite key).
  - $Subject\_Code \rightarrow Subject\_Name \rightarrow$ **Partial Dependency**
- **Student_Name** depends only on **Roll_No** (a part of the composite key).
  - $Roll\_No \rightarrow Student\_Name \rightarrow$ **Partial Dependency**

Step 2: Remove Partial Dependencies

Decompose into two tables:

1. Student Table

Roll_No	Student_Name
1	Alice
2	Bob

2. Subject Table

Subject_Code	Subject_Name
MATH01	Math
PHYS01	Physics

3. Student_Subject Table (Relationship Table)

Roll_No	Subject_Code
1	MATH01
1	PHYS01
2	MATH01
2	PHYS01

HomeWork for 2nd NF

StudentID	CourseID	StudentName	CourseName
1	101	Alice	Math
1	102	Alice	Physics

3.Third Normal Form (3NF):

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

- It is in Second Normal Form (2NF).
- There are no transitive dependencies, meaning no non-prime attribute depends on another non-prime attribute.

Steps to Achieve 3NF

1.Start with a 2NF table.
Ensure there are no partial dependencies.

2.Check for Transitive Dependencies.
Identify if any non-prime attribute depends on another non-prime attribute.

3.Remove Transitive Dependencies.
Create new tables to separate these dependencies.

4.Reorganize the Table.
Ensure that every non-prime attribute is directly dependent on the key, not on another non-prime attribute.

Example
Table: Employee

Emp_ID	Dept_ID	Dept_Name	Dept_Location
1	D01	HR	New York
2	D02	IT	San Francisco
3	D01	HR	New York
4	D03	Admin	Chicago

Candidate Key: Emp_ID
Non-Prime Attributes: $Dept_ID, Dept_Name, Dept_Location$

Step 1: Check for Transitive Dependencies

- $Emp\_ID \rightarrow Dept\_ID$
- $Dept\_ID \rightarrow Dept\_Name, Dept\_Location$

Problem:

- $Dept\_Name$ **and** $Dept\_Location$ **are dependent on** $Dept\_ID$**, not directly on Emp_ID**

Step 2: Remove Partial Dependencies

Decompose into two tables:

1. Employee Table

Emp_ID	Dept_ID
1	D01
2	D02
3	D01
4	D03

2. Department Table

Dept_ID	Dept_Name	Dept_Location
D01	HR	New York
D02	IT	San Francisco
D03	Admin	Chicago

4.Boyce-Codd Normal Form (BCNF):

A table is in BCNF if:

- It is in Third Normal Form (3NF).
- For every functional dependency (X → Y), X (determinant) must be a superkey.

Key Terms

- **Superkey:** An attribute or a set of attributes that can uniquely identify a row.
- **Determinant:** The left-hand side (X) of a functional dependency $X \rightarrow Y$.

Steps to Achieve BCNF

1. **Check if the Table is in 3NF.**
   Start from a 3NF table.
2. **Verify Functional Dependencies.**
   For each dependency $X \rightarrow Y$, check if X is a superkey.
3. **Decompose if Needed.**
   If X is not a superkey, decompose the table into smaller tables until every determinant is a superkey.

Example
Table: Student

Roll_No	Subject	Teacher
1	Math	Mr. A
2	Physics	Mr. B
3	Math	Mr. A
4	Chemistry	Mr. C

Functional Dependencies:

1. $Roll\_No \rightarrow Subject$
2. $Subject \rightarrow Teacher$

Candidate Key: $Roll\_No$
Problem:

- $Subject \rightarrow Teacher$: $Subject$ **is not a superkey.**
- **This violates BCNF.**

Step 1: Decompose the Table

- **Split into two tables:**

1. Student_Subject Table

Roll_No	Subject
1	Math
2	Physics
3	Math
4	Chemistry

2. Subject_Teacher Table

Subject	Teacher
Math	Mr. A
Physics	Mr. B
Chemistry	Mr. C

Step 2: Verify BCNF

1. **In Student_Subject,** $Roll\_No \rightarrow Subject$**, and** $Roll\_No$ **is a superkey.**
2. **In Subject_Teacher,** $Subject \rightarrow Teacher$**, and** $Subject$ **is a superkey.**

AKTU- 2023-24

Q: Why do we normalize databases?

Normalization reduces redundancy, improves data integrity, avoids anomalies, and makes databases easier to maintain.

AKTU- 2022-23

Q: List all prime and non-prime attributes

- **Relation R(A, B, C, D, E)**
- **FDs:** $AB \rightarrow C, B \rightarrow E, C \rightarrow D$
- **Candidate Key:** AB.

Answer:
Prime Attributes: A, B.
Non-Prime Attributes: C, D, E.

Q: Find Key for R and Decompose into 2NF and 3NF

AKTU- 2022-23

Relation R: {A, B, C, D, E, F, G, H, I, J}
Functional Dependencies:
AB-> C
A-> DE
B->F
F->GH
D->IJ

Step 1: Find the Candidate Key

Given Functional Dependencies:

1. $AB \rightarrow C$
2. $A \rightarrow DE$
3. $B \rightarrow F$
4. $F \rightarrow GH$
5. $D \rightarrow IJ$

Attributes: $A, B, C, D, E, F, G, H, I, J$

Step-by-Step Process:

- AB **determines** C.
- A **determines** D, E **and** B **determines** F.
- F **determines** G, H **and** D **determines** I, J.

By combining, AB **can determine all other attributes (**$AB^+ = \{A, B, C, D, E, F, G, H, I, J\}$.

Candidate Key: AB **is the only candidate key.**

Step 2: Decompose into 2NF

To achieve 2NF, remove partial dependencies (when a non-prime attribute depends only on a part of the candidate key).

Partial Dependencies:

1. $A \rightarrow DE$ **(Depends on part of the key** A**).**
2. $B \rightarrow F$ **(Depends on part of the key** B**).**

Decomposition into 2NF Relations:

1. **Relation** $R1(A, D, E)$**: Includes** $A \rightarrow DE$.
    - **Attributes:** A, D, E.
    - **Key:** A.
2. **Relation** $R2(B, F)$**: Includes** $B \rightarrow F$.
    - **Attributes:** B, F.
    - **Key:** B.
3. **Relation** $R3(F, G, H)$**: Includes** $F \rightarrow GH$.
    - **Attributes:** F, G, H.
    - **Key:** F

4. **Relation $R4(D,I,J)$: Includes $D \rightarrow IJ$.**
    - **Attributes:** D,I,J.
    - **Key:** D.
5. **Relation $R5(AB,C)$: Includes $AB \rightarrow C$.**
    - **Attributes:** A,B,C.
    - **Key:** AB.

Step 3: Decompose into 3NF

3NF eliminates transitive dependencies, where a non-prime attribute depends on another non-prime attribute.

Steps for 3NF Decomposition:

- **Check each relation from 2NF:**
    1. $R_1(A,D,E)$: **No transitive dependency, already in 3NF.**
    2. $R_2(B,F)$: **No transitive dependency, already in 3NF.**
    3. $R_3(F,G,H)$: **No transitive dependency, already in 3NF.**
    4. $R_4(D,I,J)$: **No transitive dependency, already in 3NF.**
    5. $R_5(A,B,C)$: **No transitive dependency, already in 3NF.**

Final Decomposed Relations

The 3NF decomposition results in the following relations:

1. $R_1(A,D,E)$ **with** $A \rightarrow DE$.
2. $R_2(B,F)$ **with** $B \rightarrow F$.
3. $R_3(F,G,H)$ **with** $F \rightarrow GH$.
4. $R_4(D,I,J)$ **with** $D \rightarrow IJ$.
5. $R_5(A,B,C)$ **with** $AB \rightarrow C$.

Subscribe Multi Atoms & Multi Atoms Plus

What is Inclusion Dependency?

An Inclusion Dependency is a constraint that specifies that the values in one set of columns in a table (relation) must appear in another set of columns in another table (or the same table). It expresses a subset relationship between attributes.

Student Table

student_id	course_id
101	CS101
102	CS102

Course Table

course_id	course_name
CS101	Computer Basics
CS102	Data Structures

- The course_id column in Students must match a value in the course_id column in Courses.
- This is an Inclusion Dependency.

It ensures that every course a student takes (in the Students table) actually exists in the list of courses (in the Courses table).

- **Constraint:** $\pi_{course_id}(Student) \subseteq \pi_{course_id}(Course)$

Why Is It Important?

1. **Keeps Data Consistent**: It prevents errors, like assigning a student to a course that doesn't exist.
2. **Maintains Relationships**: It keeps the "connections" between tables intact.
3. **Avoids Data Loss:** If a course is removed, any student enrolled in that course can also be removed safely, keeping the database clean.

Lossless Join Decomposition

When we split a large table (relation R) into smaller tables (like $R1$ and $R2$), **lossless join decomposition** ensures that **no data is lost** when we combine (join) these smaller tables back to reconstruct the original table.

Why is it Important?

- If decomposition is **not lossless**, some data may be missing or duplicated when we try to reconstruct the original table.
- A lossless decomposition guarantees that the original data can always be obtained by a natural join operation on the decomposed tables.

Conditions for Lossless Join Decomposition

To check if the decomposition of R into $R1$ and $R2$ is lossless, the following conditions must hold:

1.Union of Attributes:

The combined attributes of $R1$ and $R2$ must equal all the attributes of R. Mathematically:

$$\text{Attributes}(R1) \cup \text{Attributes}(R2) = \text{Attributes}(R)$$

If this condition fails, the decomposition is not lossless.

2.Intersection of Attributes:

There must be at least one common attribute between $R1$ and $R2$.
Mathematically:

$$\text{Attributes}(R1) \cap \text{Attributes}(R2) \neq \Phi$$

If $R1$ and $R2$ have no common attributes, they can't be joined to reconstruct R.

3.Key Property of Common Attributes:

The common attribute(s) must form a key in at least one of the smaller tables ($R1$ or $R2$).
Mathematically:

$$\text{Attributes}(R1) \cap \text{Attributes}(R2) \rightarrow \text{Attributes}(R1) \quad \text{or} \quad \text{Attributes}(R1) \cap \text{Attributes}(R2) \rightarrow \text{Attributes}(R2)$$

This ensures that the common attribute(s) can uniquely identify tuples in at least one of the decomposed tables.

Aktu-2022-23

Q. Given the following set of FDs on schema R (V,W,X,Y,Z)

{Z→V, W→Y, XY→Z, V→WX}

State whether the following decomposition are loss-less-join decompositions or not.
(i) R1=(V,W,X) , R2=(V,Y,Z)
(ii) R1=(V,W,X), R2=(X,Y,Z)

Decomposition (i): $R1 = (V, W, X),\ R2 = (V, Y, Z)$

1. **Condition 1:**
   Union of attributes:

   $R1 \cup R2 = (V, W, X) \cup (V, Y, Z) = (V, W, X, Y, Z)$

   ✅ **Matches R.**

2. **Condition 2:**
   Intersection of attributes:

   $R1 \cap R2 = (V, W, X) \cap (V, Y, Z) = (V)$

   ✅ **Not empty.**

3. **Condition 3:**
   Common attribute V is a key:

   - **In $R1$:** $V \rightarrow WX$ ✅

   ✅ **Lossless join!**

Decomposition (ii): $R1 = (V, W, X)$, $R2 = (X, Y, Z)$

1. **Condition 1:**
   Union of attributes:

   ✅ **Matches** R.
2. **Condition 2:**
   Intersection of attributes:

   ✅ **Not empty.**
3. **Condition 3:**
   Common attribute X **is not a key in** $R1$ **or** $R2$**:**
   - X **alone doesn't determine all attributes in** $R1$ **or** $R2$.

   ❌ **Not lossless join!**

Final Answer

1. **Decomposition (i): Lossless join** ✅
2. **Decomposition (ii): Not lossless join** ❌

Multivalued Dependencies (MVDs)

- A **Multivalued Dependency (MVD)** exists in a relation when one attribute determines multiple independent values of another attribute. Unlike functional dependencies (FDs), where one attribute determines exactly one value of another, MVD allows multiple values.
- **Notation:**
  If $A \twoheadrightarrow B$, this means A determines all possible combinations of values for B, independent of other attributes.
- **Example:**
  Consider a relation **Student(StuID, Course, Hobby):**
  - A student can have **multiple courses**.
  - A student can have **multiple hobbies**.
  - These two attributes $Course$ and $Hobby$ are **independent** of each other.

  MVD: $StuID \rightarrow\rightarrow Course$ and $StuID \rightarrow\rightarrow Hobby$

Trivial vs Non-Trivial MVD:

- **Trivial MVD:** If X and Y overlap completely or their union equals the whole table.
- **Non-Trivial MVD:** If X and Y are separate and unrelated.

Example:

- Trivial: $Teacher \twoheadrightarrow Teacher, Subject$.
- Non-Trivial: $Teacher \twoheadrightarrow Subject$ (when Subject doesn't overlap with Teacher).

Join Dependency (JD):

- **What is JD?**
- - JD specifies that a relation RR can be decomposed into smaller relations $R_1, R_2, \ldots, R_n$ such that the original relation can be perfectly reconstructed (lossless join).
- **How is it denoted?**
  - $JD(R_1, R_2, \ldots, R_n)$: $R_1, R_2, \ldots, R_n$ are subsets of R.
- **When is JD Trivial?**
  - If any one of the relations $R_1, R_2, \ldots, R_n$ is equal to the entire relation R, the JD is trivial.

- **Example of JD:**
  - Consider $R(Faculty, Subject, Committee)$.
    - Decompose R into:
      - $R_1(Faculty, Subject)$ and $R_2(Faculty, Committee)$.
    - Using a **lossless join**, R can be reconstructed:
      - $R = \Pi_{Faculty, Subject}(R_1) \bowtie \Pi_{Faculty, Committee}(R_2)$.

4NF (Fourth Normal Form)

- A table is in 4NF if it is in BCNF (Boyce-Codd Normal Form) and does not have any non-trivial Multivalued Dependencies (MVDs).
- In simple terms, 4NF eliminates redundancy caused by independent multivalued facts.

Example:

Teacher	Subject	Committee
John	Math	Placement
John	Science	Placement
John	Math	Scholarship
John	Science	Scholarship

- **Problem:**
  - The **Subject** and **Committee** are independent of each other.
  - $Teacher \twoheadrightarrow Subject$ and $Teacher \twoheadrightarrow Committee$ are **independent multivalued dependencies**.

Decomposition into 4NF:

Teacher and Subject:

Teacher	Subject
John	Math
John	Science

Teacher and Committee:

Teacher	Committee
John	Placement
John	Scholarship

5NF (Fifth Normal Form)

- A table is in 5NF if it is in 4NF and cannot be decomposed further without losing data.
- 5NF deals with Join Dependencies (JD).

Example:

Student	Course	Teacher
Alice	Math	Mr. Smith
Alice	Science	Mrs. Johnson
Bob	Math	Mr. Smith
Bob	Science	Mrs. Johnson

- **Problem:**

  This table has a join dependency: The relationship between Student, Course, and Teacher can be split into three smaller relationships.

Decomposition into 5NF:

Student and Course:

Student	Course
Alice	Math
Alice	Science
Bob	Math
Bob	Science

Course and Teacher:

Course	Teacher
Math	Mr. Smith
Science	Mrs. Johnson

Student and Teacher:

Student	Teacher
Alice	Mr. Smith
Alice	Mrs. Johnson
Bob	Mr. Smith
Bob	Mrs. Johnson

Alternate Approaches to Database Design

Database design typically follows the normalization approach, but there are alternative methods that may be more appropriate depending on the application's requirements. These approaches focus on optimizing database performance, reducing redundancy, and ensuring data integrity, sometimes diverging from strict normalization principles.

1. Denormalization

- **What it is:** Combines multiple tables into one to make data retrieval faster.
- **When to use:** For read-heavy applications where performance is more important than reducing redundancy.
- **Example:** Instead of separate Customer and Orders tables, combine them into one table with all details.
- **Advantage:** Faster reads.
- **Disadvantage:** Data redundancy increases.

2. Schema-less Design (NoSQL)

- **What it is**: No fixed structure for the database; uses flexible formats like JSON.
- **When to use**: For unstructured or frequently changing data, like in social media apps.
- **Example**: Store a customer and their orders in a single document.
- **Advantage:** Flexible and scales easily.
- **Disadvantage:** Queries can become complex.

3. Agile Database Design

- **What it is:** Start with a simple design and add features as needed over time.
- **When to use:** In rapidly changing projects like startups.
- **Example:** Begin with just a Users table and add Orders or Addresses tables later.
- **Advantage:** Adapts to new requirements quickly.
- **Disadvantage:** May need rework later.

4. Graph-Based Design

- **What it is:** Stores data as nodes (items) and edges (relationships).
- **When to use:** For highly interconnected data like social networks.
- **Example:** A User node connected to Friends or Posts nodes.
- **Advantage:** Great for relationship-heavy queries.
- **Disadvantage:** Not ideal for standard tabular data.

Aktu | Database Management System | BCS-501

Unit-4

Syllabus

Transaction Processing Concept: Transaction System, Testing of Serializability, Serializability of Schedules, Conflict & View Serializable Schedule, Recoverability, Recovery from Transaction Failures, Log Based Recovery, Checkpoints, Deadlock Handling. Distributed Database: Distributed Data Storage, Concurrency Control, Directory System.

Content Unit-4

- Transaction System
- ACID properties - (2021-22)
- Operations During a Transaction
- State diagram - (2022-23)
- Schedule & Its Types - (2022-23)
- Serializability
- Testing of Serializability
- Conflict & View Serializable
- Recoverability
- Recoverable , Cascadeless & Strict
- Recovery System
- Types of Failures
- Log-Based Recovery - 2023-24
- Deferred Database Modification 2023-24
- Immediate Database Modification
- Checkpoints
- Deadlock
- Characteristics of Deadlock / Necessary Conditions
- Deadlock Handling -2023-24 & 2021-22
- Distributed Database & its Components

Transaction System

A transaction is a series of database operations performed as a single logical unit of work, ensuring the database stays consistent even if something goes wrong. It guarantees all-or-nothing execution, which means either all the steps of the transaction are completed, or none are.

- **Example:** Imagine you are transferring ₹500 from Account A to Account B:

1. Deduct ₹500 from Account A.
2. Add ₹500 to Account B.

If either step fails (e.g., insufficient balance in Account A), the transaction should rollback to avoid inconsistent states (like ₹500 disappearing).

AKTU- 2021-22

List ACID properties of a transaction. Explain the usefulness of each. What is the importance of log?

The ACID properties ensure reliability and consistency for all database transactions.

1. Atomicity:

- **Meaning:** A transaction is treated as a single unit. It either completes entirely or fails entirely.
- **Example:** In a train booking, deducting a seat and assigning it to a passenger must both succeed; otherwise, neither action is taken.
- **Why Useful?:** Prevents incomplete changes that leave the database in an inconsistent state.

2. Consistency:

- **Meaning:** A transaction must take the database from one valid state to another, maintaining all integrity rules.
- **Example:** If a student's marks are updated, total marks must still reflect correctly.
- **Why Useful?:** Ensures database rules (like constraints or relationships) are always followed.

3. Isolation:

- **Meaning:** Multiple transactions can occur simultaneously, but they should not interfere with each other.
- **Example:** Two people trying to book the last flight seat will not both succeed; one will complete first.
- **Why Useful?:** Prevents data corruption or incorrect results in a concurrent environment.

4. Durability:

- **Meaning:** Once a transaction is committed, the changes are permanent, even if the system crashes.
- **Example:** After transferring money, the balance update remains saved despite power failure.
- **Why Useful?:** Provides reliability and trust in the system.

Importance of Log:

A log is a record of all operations performed by transactions. It is critical for failure recovery.

How it works:

Logs contain:

- **UNDO information:** Used to reverse incomplete transactions.
- **REDO information:** Used to reapply committed transactions.

Why important?

1. **If a transaction fails midway, the log ensures the database can roll back changes to its previous state.**
2. **After a system crash, logs are used to restore the database to a consistent state.**

Operations During a Transaction

1. Read (R)

- The transaction reads data from the database into memory.
- Example: Read(A) copies the value of A from the database into the transaction's local memory.

2. Write (W)

- The transaction writes or updates the value in the database.
- Example: Write(A) updates the value of A in the database after making changes in memory.

3. Update

- The transaction modifies the value of a variable in memory.
- Example: A = A - 100 reduces the value of A in the transaction's memory.

4. Commit

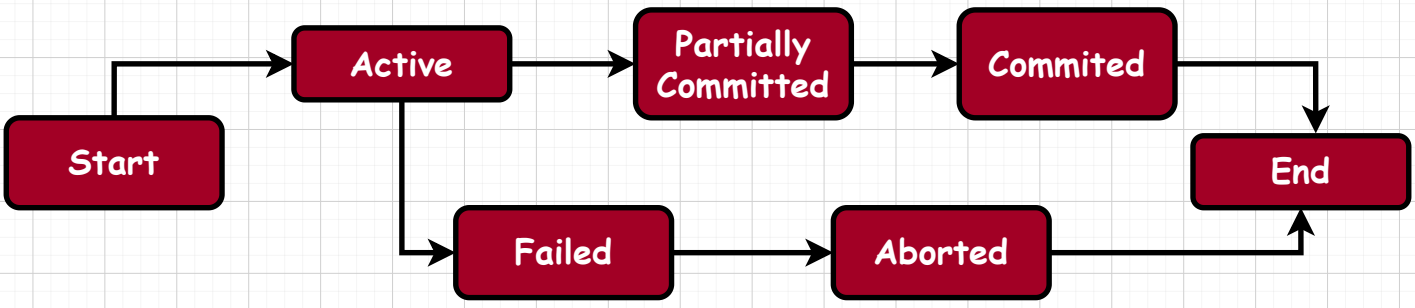
- The transaction's changes are saved permanently to the database.
- After the COMMIT operation, the changes made by the transaction become visible to other transactions.
- Example: COMMIT ensures all updates to A and B are finalized in the database.

T1	T2
	Read(A)
	A = A + 500
	Write(A)
	Read(B)
	B = B + 100
	Write(B)
	COMMIT
Read(A)	
A = A - 100	
Write(A)	
Read(B)	
B = B + 100	
Write(B)	
COMMIT	

AKTU- 2022-23

Q. Draw a state diagram and discuss the typical states that a transaction goes through during execution

A transaction moves through various states during its execution. Here's a step-by-step explanation with a diagram:



Explanation of States:

1. Active State:

- The transaction is actively executing its operations (e.g., reading or writing to the

database).

- **Example:** Deducting ₹500 from Account A.

2. Partially Committed State:

- The transaction has completed its final operation but hasn't yet saved the changes permanently.
- **Example:** Both ₹500 deduction and addition operations are completed but not yet saved to the database.

3. Committed State:

- The transaction is successfully completed, and its changes are permanently saved.
- **Example:** Account A and Account B balances are updated and stored in the database.

4. Failed State:

- Errors or issues (like system failure, invalid input, or insufficient funds) prevent the transaction from completing.
- **Example:** ₹500 deduction fails due to insufficient balance in Account A.

5. Aborted State:

- The database rolls back the transaction, undoing any changes made during execution.
- **Example:** If the money transfer fails, Account A and B are restored to their original balances.

AKTU- 2022-23

Q.When is a Transaction Rolled Back?

A rollback occurs when a transaction fails, undoing all changes made so far to maintain the database's consistency.

Scenarios for Rollback:

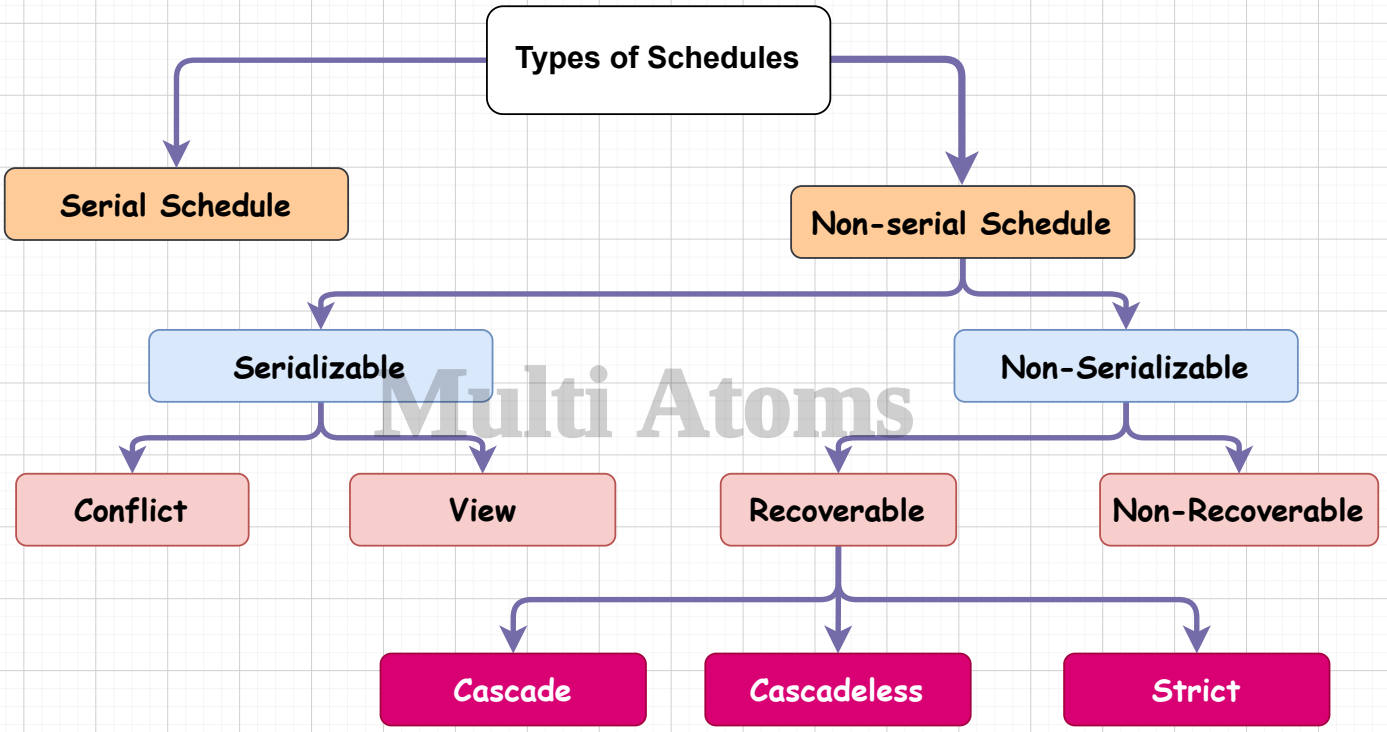
1. **System Failure:** Power outages or software crashes interrupt the transaction midway.
2. **Database Constraint Violation:** The transaction tries to break a rule (e.g., inserting a duplicate primary key).
3. **Deadlock:** Two transactions are stuck waiting for resources held by each other, blocking progress.
4. **Explicit Cancellation:** The user or system aborts the transaction manually.

AKTU- 2022-23

Schedule

A Schedule is the order in which the operations (like Read, Write, Commit, etc.) of multiple transactions are executed.

- A schedule is valid if it preserves the sequence of operations within each transaction.
- Schedules determine how concurrent transactions affect the database.



Types of Schedules

1. Serial Schedule

- Transactions are executed one after another with no interleaving.
- Ensures database consistency.
- **Example:** If two transactions T1 and T2 exist:
- Execute all operations of T1 first, then T2.
- Or execute all operations of T2 first, then T1.

2. Non-Serial Schedule

- Allows interleaving of operations from different transactions.
- Provides better concurrency.
- Example: Operations from T1 and T2 are mixed together.

T1	T2
Read(A)	
A = A - 100	
Write(A)	
Read(B)	
B = B + 100	
Write(B)	
Commit	
	Read(A)
	A = A + 200
	Write(A)
	Commit

Serial

T1	T2
Read(A)	
A = A - 100	
	Read(A)
Write(A)	
	A = A + 200
	Write(A)
Read(B)	
B = B + 100	
Write(B)	Commit
Commit	

Non- Serial

Serializability in Non-Serial Schedules

Non-Serial schedules can be:

1. Serializable

- Behaves as if the transactions were executed in a serial order.
- **Includes two types:**

1. **Conflict Serializability:** Based on conflicts (Read-Write, Write-Write).
2. **View Serializability:** Based on the final result.

2. Non-Serializable

- **Does not guarantee consistency.**

Recoverability in Non-Serializable Schedules

Recoverability ensures no data inconsistencies occur when transactions fail. Types include:

1. Recoverable Schedule

- Ensures that if one transaction depends on another, the dependent transaction commits only after the first transaction commits.

2. Cascading Schedule

2. Cascading Schedule

- Failure in one transaction causes rollback of multiple transactions.

3. Cascadeless Schedule

- Prevents cascading rollbacks by ensuring a transaction reads only committed data.

4. Strict Schedule

- Prevents both cascading rollbacks and dirty reads.

Concurrency Problems

Concurrency problems occur when multiple transactions execute simultaneously and interact with the same data, leading to unexpected or incorrect outcomes. The most common concurrency problems are:

1. Dirty Read

- **Definition:** A transaction reads uncommitted changes made by another transaction.
- **Impact:** Inconsistent and unreliable data.
- **Prevention:** Use the Read Committed isolation level or higher.

T1 (Uncommitted Write)	T2 (Reads Dirty Data)
`Read(A=100)`	
`A = A - 50 (A=50)`	
`Write(A=50)` (Uncommitted)	`Read(A=50)` (Dirty Read)
`ROLLBACK` (Reverts A to 100)	Uses incorrect `A=50`

3. Phantom Read

- **Definition:** A transaction re-executes a query and gets a different result set because another transaction has added or removed rows that match the query criteria.
- **Impact:** The result set changes unexpectedly.
- **Prevention:** Use the Serializable isolation level.

T1 (Reads Rows)	T2 (Inserts Rows)
`SELECT * WHERE salary > 5000`	
(Returns 2 rows: Emp1, Emp2)	
	`INSERT Emp3 (salary=6000)`
	`COMMIT`
`SELECT * WHERE salary > 5000`	(Now returns 3 rows: Emp1, Emp2, Emp3)

4. Lost Update

- **Definition:** Two transactions read the same data and update it concurrently, but one update overwrites the other, leading to a loss of data.
- **Impact:** One update is lost, leading to incorrect results.
- **Prevention:** Use locks or the Serializable isolation level.

T1 (Updates A)	T2 (Updates A)
`Read(A=100)`	`Read(A=100)`
`A = A + 20 (A=120)`	
	`A = A - 30 (A=70)`
`Write(A=120)`	
	`Write(A=70)` (Overwrites A=120)

5. Deadlock

- **Definition:** Two or more transactions wait indefinitely for resources locked by each other, creating a circular wait.
- **Impact:** Transactions are unable to proceed, leading to a system stall.
- **Prevention:** Deadlock detection and recovery mechanisms (e.g., timeout, resource ordering).

T1 (Locks Resource A)	T2 (Locks Resource B)
`Lock(A)`	
	`Lock(B)`
`Waits for Lock(B)`	`Waits for Lock(A)`

Types of Read-Write Conflicts

1. Write-Read Conflict (Uncommitted Read)

- Occurs when one transaction writes to a data item, and another transaction reads it before the first transaction commits.
- Problem: The second transaction may read incorrect or uncommitted data.

T1: Write(X)
T2: Read(X) // Before T1 commits

2. Read-Write Conflict (Dirty Write)

- Happens when a transaction reads a data item, and another transaction writes to it before the first transaction completes.
- **Problem:** The written data might overwrite uncommitted changes, causing inconsistency.

T1: Read(X)
T2: Write(X) // Before T1 commits

T2: Write(X) // Before T1 commits

3. Write-Write Conflict (Overwriting)

- Occurs when two transactions write to the same data item without considering the order of execution.
- **Problem:** One write operation overwrites the other's changes, leading to loss of data.

T1: Write(X)
T2: Write(X) // Without checking T1's commit

What is Serializability?

- A schedule is serializable if its result is the same as a serial schedule (i.e., transactions execute without overlapping).
- Serializable schedules ensure database consistency is maintained even when transactions are executed concurrently.

Importance of Serializability

1. Ensures database consistency.
2. Avoids unexpected behaviors or conflicts between transactions.
3. Is a key property for concurrency control mechanisms in DBMS.

Types of Serializability

AKTU- 2023-24, 2021-22

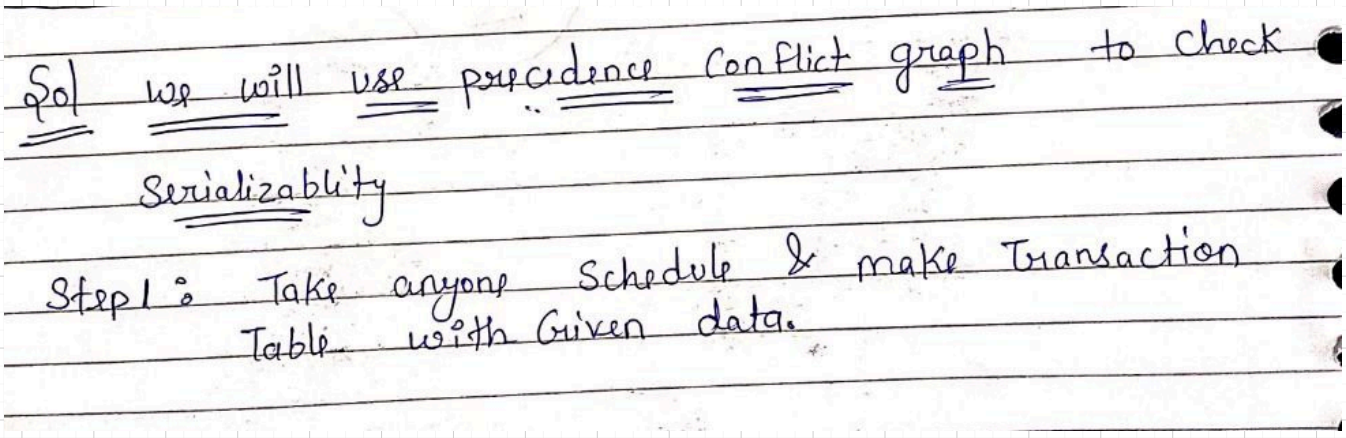
1. Conflict Serializability

- A schedule is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations (read/write on different data items).
- Conflict serializability is determined using a precedence graph
- A schedule is conflict-serializable if its precedence graph (serialization graph) has no cycles.

- **Steps to Create a Precedence Graph:**
  1. Create a node for each transaction in the schedule.
     T_i
     $\rightarrow T$ T T
  2. Draw a directed edge $_j$ if a conflicting operation in $_i$ precedes $_j$.
     (Conflicts: Read-Write, Write-Read, Write-Write on the same data item.)
  3. If the graph contains a **cycle**, the schedule is **not conflict serializable**. Otherwise, it is conflict serializable.

AKTU- 2022-23

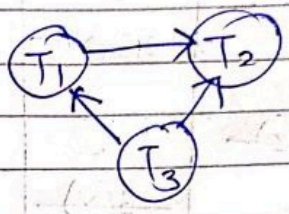
Consider the three transactions T1, T2, and T3, and the schedules S1 and S2given below. State whether each schedule is serializable or not. If a schedule is serializable, write down the equivalent serial schedule(s).
T1: r1 (X); r1 (Z); w1 (X);
T2: r2 (Z); r2 (Y); w2 (Z); w2 (Y);
T3: r3 (X); r3 (Y); w3 (Y);
S1: r1 (X); r2 (Z); r1 (Z); r3 (X); r3 (Y); w1 (X); w3 (Y); r2 (Y); w2 (Z); w2 (Y);
S2: r1 (X); r2 (Z); r3 (X); r1 (Z); r2 (Y); r3 (Y); w1 (X); w2 (Z); w3 (Y); w2 (Y);



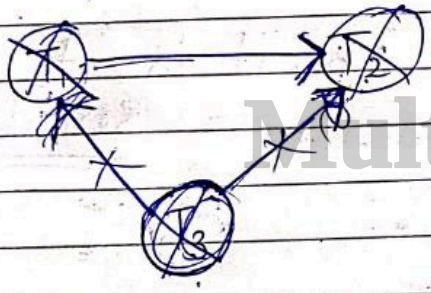
① to S1 Schedule.

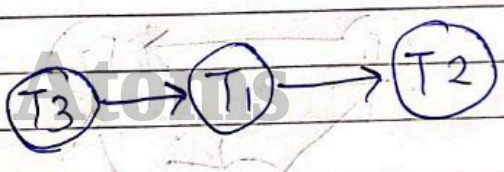
T1	T2	T3
~~r1(x)~~		
	~~r2(z)~~	
~~r1(z)~~		
		~~r3(x)~~
		~~r3(y)~~
~~w1(x)~~		
		~~w3(y)~~
	~~r2(x)~~	
	~~w2(z)~~	
	~~w2(y)~~	

check.
read - write
write - read
write - write.



yes → it is Serializable

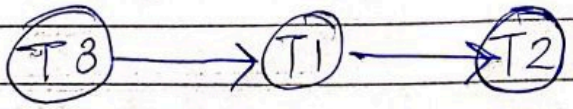




→ check cycle if there is cycle this is not Serializable

→ If not. ① remove zero degree edges. and repeat.

② Track the node & make Serializable Schedule.

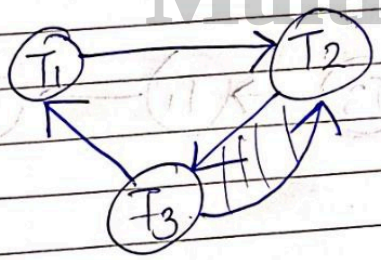


(11)

S2

T1	T2	T3
r1(x)		
	r2(z)	
		r3(x)
r1(z)		
	r2(y)	
		r3(y)
w1(x)		
	w2(z)	
		w3(y)
	w2(y)	

check.
r - w
w - r
w - w.



No, it is not serializable.

View Serializability

View serializability ensures that a non-serial schedule produces the same results as a serial schedule by maintaining the consistency of database operations.

A schedule is view serializable if it is view equivalent to a serial schedule.

Two schedules S1 and S2 are view equivalent if they satisfy the following conditions:

Steps to Check View Serializability

1. **Check Initial Read:** Compare the initial reads for each data item in the non-serial and serial schedules.
2. **Check Updated Read:** Verify that each transaction reads data updated by the same transaction in both schedules.
3. **Check Final Write**: Ensure the final writes for each data item are performed by the same transaction.

If all three conditions are satisfied, the schedule is view equivalent and hence view serializable.

S1

T1	T2
$R(A)$	
$A = A + 10$	
$W(A)$	
$R(B)$	
$B = B + 20$	
$W(B)$	
	$R(A)$
	$A = A + 10$
	$W(A)$
	$R(B)$
	$B = B \times 1.1$
	$W(B)$

S2

T1	T2
$R(A)$	
$A = A + 10$	
$W(A)$	
	$R(A)$
	$A = A + 10$
	$W(A)$
$R(B)$	
$B = B + 20$	
$W(B)$	
	$R(B)$
	$B = B \times 1.1$
	$W(B)$

1. Initial Read

1. Initial Read

The first transaction that reads a data item in S_1 and S_2 should be the same.

- For A:
  - In S_1, T_1 performs $R(A)$ first.
  - In S_2, T_1 performs $R(A)$ first.
    ✅ **Condition is satisfied for A.**
- For B:
  - In S_1, T_1 performs $R(B)$ first.
  - In S_2, T_1 performs $R(B)$ first.
    ✅ **Condition is satisfied for B.**
- In $2, 1$ performs $R(B)$ first.
  ✅ **Condition is satisfied for B.**

✅ Condition is satisfied for B.

2. Updated Read

If a transaction T_i reads a value X updated by another transaction T_j, then the same order of updates must be preserved in both schedules.

- For A:
  - In S_1, T_2 reads A updated by T_1 (via $W(A)$).
  - In S_2, T_2 reads A updated by T_1 (via $W(A)$).
    ✅ **Condition is satisfied for A.**
- For B:
  - In S_1, T_2 reads B updated by T_1 (via $W(B)$).
  - In S_2, T_2 reads B updated by T_1 (via $W(B)$).
    ✅ **Condition is satisfied for B.**

3. Final Write

The transaction that performs the last write on a data item in S_1 and S_2 should be the same.

- For A:
  - In S_1, the final write on A is performed by T_2.
  - In S_2, the final write on A is performed by T_2.
    ✅ **Condition is satisfied for A.**
- For B:
  - In S_1, the final write on B is performed by T_2.
  - In S_2, the final write on B is performed by T_2.
    ✅ **Condition is satisfied for B.**

1. **Conflict Serializable = View Serializable:**
   - If a schedule is conflict serializable, it is always view serializable.
2. **View Serializable ≠ Conflict Serializable:**
   - A view serializable schedule may not be conflict serializable if it involves blind writes (a write operation without a preceding read).

Aspect	Conflict Serializability	View Serializability
Definition	A schedule is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.	A schedule is view serializable if it is view equivalent to a serial schedule.
Focus	Focuses on **conflicting operations** (read/write, write/write) and their order.	Focuses on the **logical outcome** of the transactions (initial read, updated read, and final write).
Condition	Checks for **conflict equivalence** between schedules.	Checks for **view equivalence** between schedules.
Practicality	Easier to check and implement as it only involves swapping operations and checking conflicts.	More general and complex as it involves checking all three conditions: initial read, updated read, and final write.
Includes Blind Writes	Does **not include blind writes** (write operations with no preceding read).	Includes **blind writes** in its validation process.
Complexity	Relatively simpler as it involves identifying and reordering conflicts.	More complex as it evaluates logical equivalence beyond conflicts.
Relation	All conflict-serializable schedules are view serializable.	Not all view-serializable schedules are conflict serializable.
Example of Exclusion	Schedules with **blind writes** are not conflict serializable but may be view serializable.	Schedules that are conflict serializable are also view serializable.

Recoverability

AKTU- 2022-23

Recoverability ensures that the database remains consistent even if a transaction fails or rolls back. It means one transaction should commit only if all the other transactions it depends on have committed successfully.

Non-Recoverable Schedule (Bad)

T1	T2	Explanation
Write(A)		T1 writes data to A.
	Read(A)	T2 reads the data written by T1.
	Commit	T2 commits its transaction (dependent on T1's data).
Rollback		T1 fails and rolls back, making the data used by T2 invalid.

1. **Problem**: T2 committed before T1 committed, and then T1 failed.
2. **Result**: Database is in an inconsistent state because T2 used invalid data.

Recoverable Schedule (Good)

T1	T2	Explanation
Write(A)		T1 writes data to A.
	Read(A)	T2 reads the data written by T1.
Commit		T1 commits its transaction.
	Commit	T2 commits after ensuring T1's data is valid.

1. **Why Good?:** T2 commits after T1 commits, ensuring consistency.

Types of Schedules in Terms of Recoverability

1. Recoverable Schedule:

- A schedule is recoverable if a transaction commits only after the transactions it depends on have committed.
- **Example:** If T2 reads a value written by T1, T2 should not commit until T1 has committed.
- Why important? Prevents inconsistency caused by committing a transaction that depends on another uncommitted transaction.

2. Cascadeless Schedule:

- A stricter type of schedule where a transaction is not allowed to read uncommitted data from another transaction.
- **Example**: T2 cannot read A until T1 has committed its changes to A.
- Why important? Prevents cascading rollbacks where one failure causes many transactions to fail.

T1	T2	Explanation
Write(A)		T1 writes data to A.
Commit		T1 commits **before** any other transaction reads its data.
	Read(A)	T2 reads data only after T1 commits.
	Commit	T2 commits.

1. **Why Better?:** T2 waits for T1 to commit before reading its data, avoiding cascading rollbacks.

3. Strict Schedule:

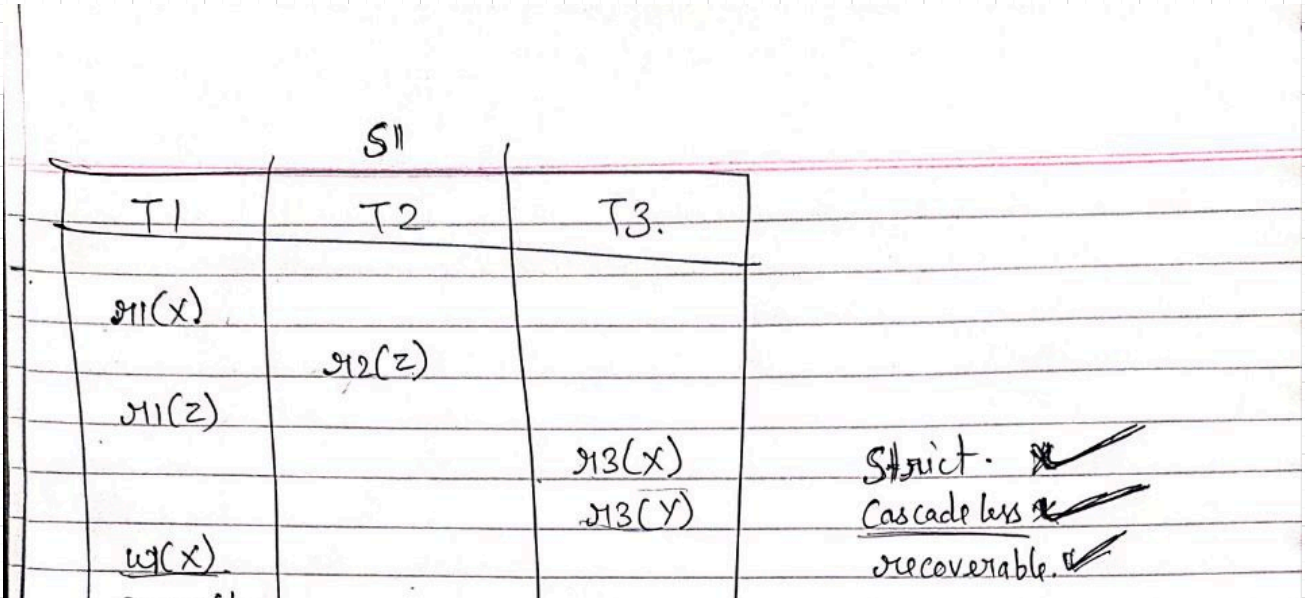
- The strictest type where no transaction can read or write a value modified by another transaction until that transaction has committed or rolled back.
- Why important? Makes recovery easier because no uncommitted changes are accessed by other transactions.

T1	T2	Explanation
Write(A)		T1 writes data to A.
	Wait	T2 is not allowed to read or write A until T1 commits.
Commit		T1 commits its transaction.
	Read(A)	T2 reads A only after T1 commits.
	Commit	T2 commits.

1. **Why Best?:** T2 neither reads nor writes A until T1 commits, ensuring maximum safety.

AKTU- 2022-23

Consider schedules S1, S2, and S3 below. Determine whether each schedule is strict, cascade less, recoverable, or non recoverable. (Determine the strictest recoverability condition that each schedule satisfies.)
S1: r1 (X); r2 (Z); r1 (Z); r3 (X); r3 (Y); w1 (X); c1; w3 (Y); c3; r2 (Y); w2 (Z); w2 (Y); c2;
S2: r1 (X); r2 (Z); r1 (Z); r3 (X); r3 (Y); w1 (X); w3 (Y); r2 (Y); w2 (Z); w2 (Y); c1;c2; c3;
S3: r1 (X); r2 (Z); r3 (X); r1 (Z); r2 (Y); r3 (Y); w1 (X); c1; w2 (Z); w3 (Y); w2 (Y);c3; c2;



Commit		
	w3(Y)	
	Commit.	Strictest Condition for S1 : Strict.
r2(Y)		
w2(Y)		
w2(Z)		
w2(Y)		
Commit		

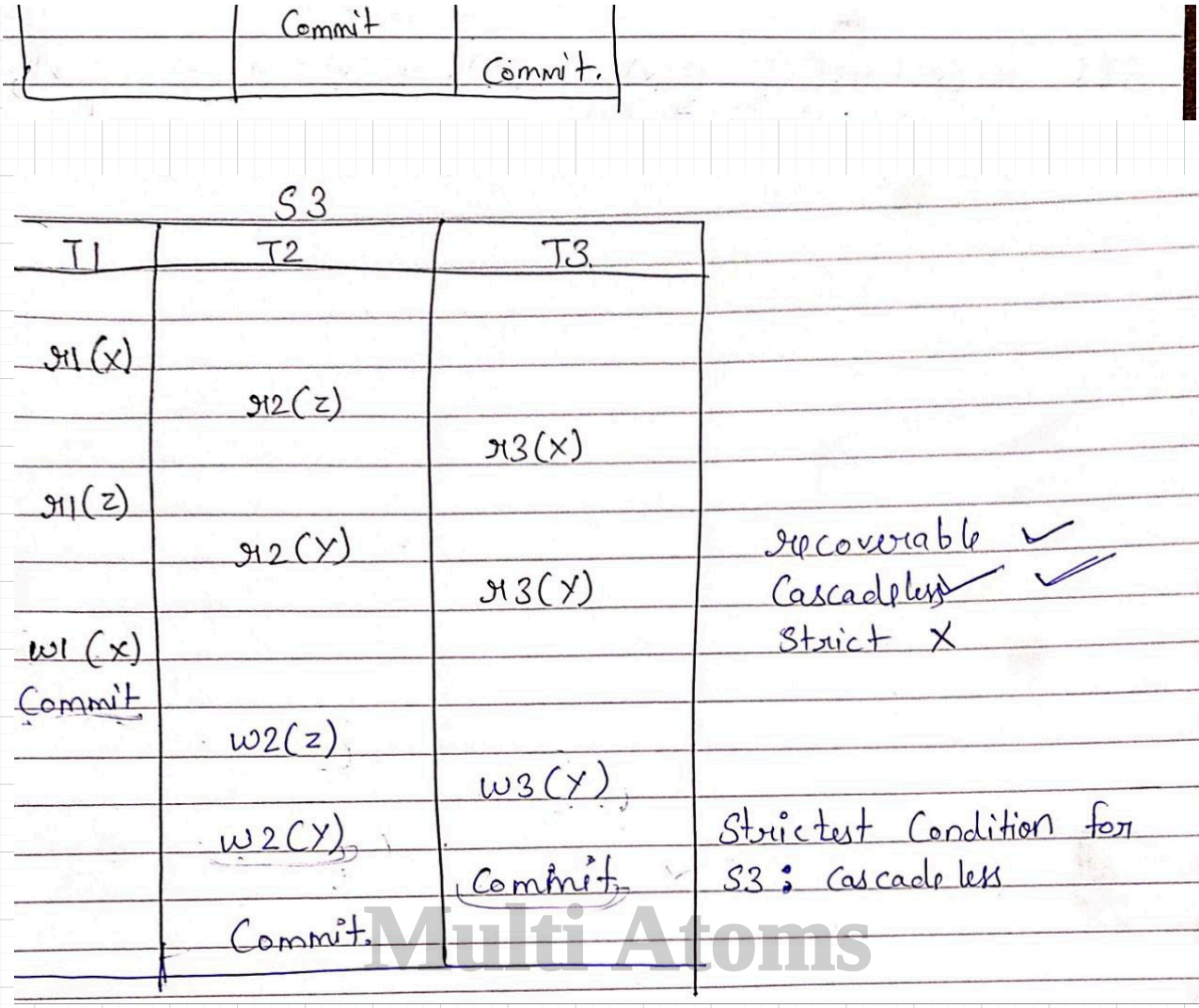
Recoverability → Commit of Modified (write of operation) Transaction is first before the commit of another transaction.

Cascadeless Schedule → Commit operation of T1 is Just (No dirty read) before the Read operation of T2

Strict Schedule → Data Modified (write) by one Transaction is allowed to perform R/W Operation after Commit by T1 transaction.

S2

T1	T2	T3	
r1(X)			
	r2(Z)		Recoverable ✗
r1(Z)			Cascadeless ✗
		r3(X)	Strict ✗
		r3(Y)	
w1(X)			Strictest Condition for S2 : Non-Recoverable.
-		w3(Y)	
	r2(Y)		
	w2(Z)		
	w2(Y)		
Commit			



Recovery System

It is responsible for ensuring the database's consistency and integrity after failures. It restores the database to its last consistent state using recovery techniques.

Types of Failures

Failures in DBMS can occur at various levels and are classified to identify the cause and restore the database to a consistent state.

1. Transaction Failure

Occurs when a transaction cannot complete or reach a consistent state.

Reasons for Transaction Failure

- **Logical Errors:**

  Errors in the transaction's logic (e.g., division by zero, constraint violations).
  Example: A transaction tries to withdraw more money than available in an account.

- **Syntax Errors:**

- Caused by system-imposed restrictions (e.g., deadlocks, resource unavailability).
- The DBMS may terminate the transaction automatically.
- Example: A transaction fails due to a deadlock.

2. System Crash

Occurs due to issues at the system level, like hardware or software failures.

Causes of System Crash

- **Power Failure:** Sudden loss of power halts the system.
- **Hardware Failure:** Malfunctioning components like RAM or processor.
- **Software Failure:** Errors in the operating system or DBMS.

Fail-Stop Assumption

- Non-volatile storage (e.g., hard drives) is assumed to remain unaffected during system crashes.

3. Disk Failure

Related to physical storage media.

Causes of Disk Failure

- **Bad Sectors:** Corruption in parts of the disk.
- **Head Crashes:** Physical damage to the disk's read/write head.
- **Disk Unreachability:** System fails to access the disk due to connectivity issues.

Impact of Disk Failure

- Partial or complete loss of stored data.
- Requires backup restoration or specialized recovery tools.

1. Undo Operation

- **Purpose:** Reverts changes made by uncommitted transactions.
- **When Used:** If a transaction fails or aborts, its changes to the database must be rolled back to maintain consistency.

Example of Undo
Transaction T1:

1. Write(A = 50)
2. Write(B = 30)

If T1 fails before committing:

Undo changes:

- Restore A to its original value.
- Restore B to its original value.

2. Redo Operation

- **Purpose:** Reapplies changes made by committed transactions to ensure durability.
- **When Used:** If a system crash occurs after a transaction commits but before the changes are written to the database, those changes need to be reapplied.

Transaction T2:

1. Write(A = 100)
2. Write(B = 200)
3. Commit

If a crash occurs after the commit but before changes are fully applied:

Redo changes:

- Reapply A = 100.
- Reapply B = 200.

Log-Based Recovery

Aktu 2023-24

A log is a sequence of records stored in stable storage to enable recovery of the database after a failure. Each database operation is logged before it is applied.

How Logs Are Maintained

1. **Start of Transaction**
   - **<Tn, Start>:** Indicates the transaction Tn has started.
2. **Modification Log**
   - **<Tn, Account_Balance, 1000, 1200>:** Logs the old and new values after a transaction modifies the Account_Balance.
3. **Commit Log**
   - **<Tn, Commit>:** Indicates that the transaction Tn has been successfully completed.

4. Abort Log
   - **<Tn,abort>:** Indicates Any failure occur

Database Modification Approaches

Aktu 2023-24

1. Deferred Database Modification

- **Definition:** Database changes are applied only after the transaction commits.
- **Log Requirement:**

1. Operations are logged, but changes are deferred until the transaction commits.
2. No undo is required, as changes are applied only after ensuring transaction success.

```
<T0. Start>
<T0. A. 850. 800>
<T0. B. 1000, 1050>
<T0. Commit>
<T1. Start>
<T1, C, 600, 500>
```

T0 (Transaction)	T1 (Transaction)
Start	
Write(A: 850 → 800)	
Write(B: 1000 → 1050)	
Commit	
	Start
	Write(C: 600 → 500)

Recovery Process:

- If the system crashes after <T0, Commit>, redo T0 as its changes are in the log.
- If the system crashes before <T1, Commit>, ignore T1's changes, as they are not applied.

2. Immediate Database Modification

- **Definition:** Changes to the database are applied immediately, even before the transaction commits.
- **Log Requirement:**

1. Every operation writes to the log before the actual database modification.
2. Both undo and redo operations are needed in case of failure.

Advantages:
Faster updates during transaction execution.

<T0, Start>
<T0, A, 850, 800>
<T0, B, 1000, 1050>
<T0, Commit>
<T1, Start>
<T1, C, 600, 500>

T0 (Transaction)	T1 (Transaction)
Start	
Write(A: 850 → 800)	
Write(B: 1000 → 1050)	
Commit	
	Start
	Write(C: 600 → 500)

Recovery Process:

- **If the system crashes after <T0, Commit>, redo T0 since it is committed.**
- **If the system crashes before <T1, Commit>, undo T1 since it is not committed.**

Checkpoints

A checkpoint is a mechanism that marks a point in the transaction log where the system was in a consistent state. It ensures efficient log management by discarding older logs after the checkpoint and recording new ones after the checkpoint.

Recovery Process:

1. **Log Scanning:** The recovery system scans the logs in reverse order, starting from the most recent transaction logs and going back to the checkpoint.

- **Redo List:** Transactions that were committed (i.e., <Tn, Start> and <Tn, Commit>) and need to be reapplied.
- **Undo List:** Transactions that were incomplete (i.e., <Tn, Start> but no <Tn, Commit> or <Tn, Abort>) and need to be rolled back.

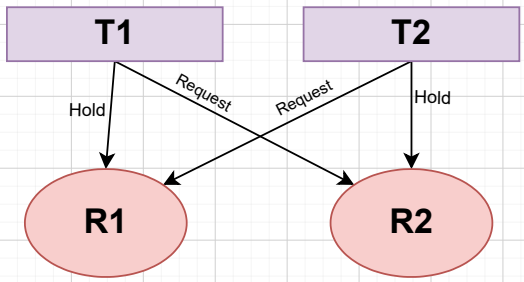
Example of Recovery:
Let's assume we have the following log after a checkpoint:

Log Record	Transaction T1	Transaction T2	Transaction T3	Transaction T4
<T1, Start>	X			
<T2, Start>		X		
<T3, Start>			X	
<T2, Commit>		X		
<T4, Start>				X
<T3, Commit>			X	

Redo transactions: T2 and T3 (both have <Tn, Start> and <Tn, Commit>).
Undo transactions: T1 (only <Tn, Start>) and T4 (only <T4, Start>).

Deadlock in DBMS

A deadlock occurs in a database when **two or more transactions are stuck because each is waiting for the other to release a resource. This creates a cycle where no transaction can proceed, halting the system.** Deadlocks are a significant challenge in multi-user environments and can severely impact the system's performance and reliability.



Characteristics of Deadlock / Necessary Conditions

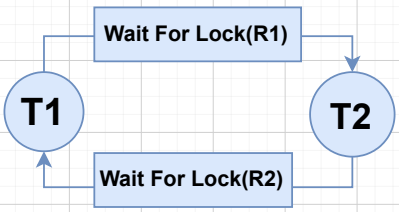
1. **Mutual Exclusion:** Only one transaction can hold a specific resource at a time.
2. **Hold and Wait:** A transaction holding resources may request additional resources held by others.
3. **No Preemption:** Resources cannot be forcibly taken from a transaction; they must be released voluntarily.
4. **Circular Wait:** A set of transactions exists where each is waiting for the next to release a resource.

Deadlock Handling

1. Deadlock Detection

Wait-For Graph:

- Transactions are represented as nodes. If a cycle is detected in the graph, a deadlock exists.
- **Action:** Abort one transaction in the cycle to break the deadlock.



2. Deadlock Avoidance

- **Resource Ordering**: Always access resources in a predefined order.
- **Release Quickly:** Release resources immediately after use.
- **Lock Granularity**: Use finer locks (e.g., row-level instead of table-level) to reduce conflicts.

1. **Example:**
   - For a **Students** and **Exams** table, always access Students first, then Exams. This consistency avoids scenarios where transactions hold conflicting locks.

3. Deadlock Prevention

Deadlock prevention ensures that the system allocates resources in a way that avoids circular waits, which are the main cause of deadlocks. Two common schemes for prevention are Wait-Die and Wound-Wait.

Wait-Die Scheme

- If an older transaction requests a resource held by a younger transaction, it waits.
- If a younger transaction requests a resource held by an older transaction, the younger one is aborted and restarted later.
- Key Idea: Older transactions have higher priority, and younger ones are rolled back when conflicts occur.

Example:

- Transaction T1 (older) requests a resource held by T2 (younger) → T1 waits.
- Transaction T2 (younger) requests a resource held by T1 (older) → T2 is aborted and restarted.

Wound-Wait Scheme

- If an older transaction requests a resource held by a younger transaction, the younger transaction is aborted (wounded) and restarted later.
- If a younger transaction requests a resource held by an older transaction, it waits until the resource is released.
- Key Idea: Older transactions can preempt younger ones to avoid deadlocks.

Example:

- Transaction T1 (older) requests a resource held by T2 (younger) → T2 is aborted, and T1 proceeds.
- Transaction T2 (younger) requests a resource held by T1 (older) → T2 waits.

Aspect	Wait-Die Scheme	Wound-Wait Scheme
Technique	Non-preemptive	Preemptive
Older Transaction	Waits for younger transactions.	Forces younger transactions to abort.
Younger Transaction	Aborted if it requests older transactions.	Waits for older transactions to release.
Number of Rollbacks	Higher due to frequent aborts of younger transactions.	Lower as younger transactions wait instead of rolling back often.
Resource Allocation	Older transactions are more patient.	Older transactions are more aggressive.
System Overhead	Higher due to increased rollbacks.	Lower as fewer rollbacks occur.

Q. Discuss the procedure of deadlock detection and recovery in transaction?

AKTU- 2021-22 & 2023-24

In multi-user database systems, deadlocks can occur when two or more transactions wait indefinitely for resources held by each other. To ensure the system operates smoothly, it is crucial to detect and recover from deadlocks efficiently.

Deadlock Detection

Deadlock detection identifies cycles of waiting transactions that prevent further progress. The primary approach is the Wait-for Graph.

Wait-for Graph (WFG)

- **Definition:** A directed graph where each node represents a transaction, an edge T1 -> T2 indicates that transaction T1 is waiting for a resource held by transaction T2.
- **Cycle:** If the graph contains a cycle, a deadlock is present.

Detection Steps:

1. **Monitor Resources:** The DBMS tracks transactions and their resource requests.
2. **Graph Construction:** The system constructs a wait-for graph using transaction states and resource allocations.
3. **Cycle Detection:** Algorithms such as depth-first search (DFS) are used to find cycles in the graph.
4. **Deadlock Confirmation:** If a cycle exists, the transactions involved are declared

deadlocked.

Deadlock Recovery

Once a deadlock is detected, the system must resolve it to allow progress. Common recovery strategies include:

1. Transaction Abortion

Key Idea: Abort one or more transactions involved in the deadlock to break the cycle.
Criteria for Selection:

- **Priority:** Abort younger or less critical transactions first.
- **Resource Usage:** Choose transactions holding fewer resources.
- **Progress:** Prefer aborting transactions with minimal progress to avoid wasting work.

2. Rollback Transactions

Key Idea: Undo the actions of the aborted transactions.
Steps:

- Use log-based recovery techniques to undo changes made by the transaction.
- Ensure data consistency and integrity by restoring the database to its previous state.

3. Timeout Mechanism

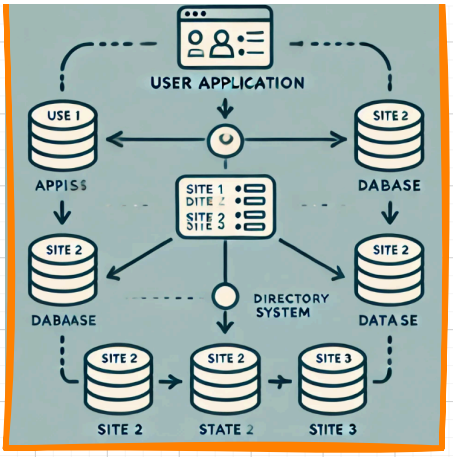
Key Idea: Automatically terminate transactions waiting too long for resources.

- **Implementation:** A transaction is forcefully aborted if it exceeds the timeout threshold.

Distributed Database

It is a collection of data spread across multiple locations, interconnected via a network. Each site in a distributed database system functions independently, but together they form a unified database system.

Key components of distributed databases include **Distributed Data Storage, Concurrency Control, and Directory System**, explained below:



1. Distributed Data Storage

Distributed Data Storage refers to the practice of splitting and storing a database across multiple physical locations, which could be on different servers or geographical regions. This allows for better performance, scalability, and fault tolerance.

Techniques:

1. Fragmentation:

- The database is divided into smaller pieces called fragments. These fragments can be stored across multiple locations.
- Horizontal Fragmentation: Divides a table by rows (e.g., all customer data for a specific region).
- Vertical Fragmentation: Divides a table by columns (e.g., only storing certain fields like customer names or addresses).

2. Replication:

- Copies of the same data are stored at multiple sites to ensure availability and faster access.
- Full Replication: All data is copied to every site.
- Partial Replication: Only some data is copied across sites, based on access patterns or other criteria.

3. Hybrid Approach:

- Combines both fragmentation and replication to ensure that data is divided efficiently and replicated for fault tolerance.

Advantages:

- **Faster local access to data:** Data can be stored closer to users or applications, reducing latency.
- **Improved reliability and fault tolerance:** Even if one site fails, data is still available from other sites that store replicas.

Challenges:

- **Data Synchronization across Sites:** Keeping data consistent across multiple locations can be complex, especially in cases of updates or changes.
- **Increased Storage Requirements:** Replicating data across multiple sites requires additional storage space, which can increase costs.

2. Concurrency Control

Concurrency Control ensures that multiple transactions running simultaneously across different locations in a distributed database do not cause inconsistencies or violations of data integrity.

Goals:

- **Maintain Data Integrity:** Ensures that concurrent transactions do not interfere with each other, keeping the database consistent.
- **Prevent Conflicts during Concurrent Updates:** Prevents issues like lost updates, temporary inconsistency, or conflicting updates.
- **Preserve Transaction Isolation and Consistency:** Ensures that transactions are isolated from one another and the system remains in a consistent state even when multiple transactions are executed concurrently.

Techniques:

1. **Lock-Based Protocols:**

   - **Distributed Two-Phase Locking (2PL):** A protocol where each transaction locks resources before it starts and releases locks after completing. It ensures consistency by maintaining a consistent order of locking across sites, ensuring that all required locks are acquired before a transaction can be executed.

2. **Time-Stamp Ordering:**

   - Each transaction is assigned a unique global timestamp, and conflicts are resolved by the order of their timestamps. This ensures transactions follow the correct sequence without interfering with one another.

3. **Optimistic Concurrency Control:**

   - Transactions execute without locks or restrictions, but before committing, they are validated to check if any conflicts occurred during their execution. If no conflicts are found, the transaction is committed; otherwise, it is rolled back and retried.

4. **Quorum-Based Protocols:**

   - In this method, a transaction requires approval from a majority (quorum) of the nodes before it can proceed. This ensures that data is not modified by transactions that are not fully validated by the majority.

3. Directory System

The directory system in a distributed database maintains metadata about the database's structure, data locations, fragmentation, and replication. It functions like a "map" that tracks where data resides and how it's organized, enabling efficient data retrieval and management across multiple sites.

Responsibilities:

- **Locate Data or Fragments:** It helps in finding where specific data or its fragments are stored across the distributed system.
- **Manage Replication:** It tracks duplicate copies of data to ensure they are available and up to date across different sites.
- **Transparent Access:** The directory system ensures that users and applications can access data without needing to know the physical location of the data or where it is replicated.

Types:

1. **Centralized Directory:**
   - A single directory that holds all the metadata. It's simple to implement but creates a single point of failure, which could be problematic for reliability and availability.
2. **Distributed Directory:**
   - Metadata is distributed across multiple locations or nodes. This improves reliability, fault tolerance, and access speed, as it eliminates the risks associated with a single point of failure.
3. **Hierarchical Directory:**
   - This approach combines both centralized and distributed systems, organizing the metadata in a tree-like structure to balance efficiency and scalability.

Aktu | Database Management System | BCS-501

Unit-5

Syllabus

Concurrency Control Techniques: Concurrency Control, Locking Techniques for Concurrency Control, Time Stamping Protocols for Concurrency Control, Validation Based Protocol, Multiple Granularity, Multi Version Schemes, Recovery with Concurrent Transaction, Case Study of Oracle.

Content of Unit-5

- Concurrency Control
- Concurrency Problems
- Lock-Based Protocols in Concurrency Control
- Time Stamping Protocols for C.C. - 2PYQ
- Validation Based Protocol - 3PYQ
- Multiple Granularity
- Multi Version Schemes - 2PYQ
- Recovery with Concurrent Transaction
- Case Study of Oracle
- Graph Based Locking Protocol - 1PYQ

Concurrency Control

Concurrency control is a mechanism in database systems to manage simultaneous transactions while ensuring data consistency, integrity, and isolation. It prevents problems that can arise when multiple transactions access and manipulate the same data concurrently.

Importance of Concurrency Control

1. **Data Consistency:** Ensures the database remains accurate and reliable despite simultaneous transactions.
2. **Isolation:** Guarantees that one transaction's intermediate states are not visible to others.
3. **Fair Resource Sharing:** Allows multiple users to perform transactions efficiently.
4. **Prevention of Anomalies:** Resolves conflicts like lost updates, dirty reads, and other concurrency-related issues.

Concurrency Problems

Concurrency problems occur when multiple transactions execute simultaneously and interact with the same data, leading to unexpected or incorrect outcomes. The most common concurrency problems are:

1. Dirty Read

- **Definition:** A transaction reads uncommitted changes made by another transaction.
- **Impact:** Inconsistent and unreliable data.
- **Prevention:** Use the Read Committed isolation level or higher.

T1 (Uncommitted Write)	T2 (Reads Dirty Data)
`Read(A=100)`	
`A = A - 50 (A=50)`	
`Write(A=50)` (Uncommitted)	`Read(A=50)` (Dirty Read)
`ROLLBACK` (Reverts A to 100)	Uses incorrect `A=50`

3. Phantom Read

- **Definition:** A transaction re-executes a query and gets a different result set because another transaction has added or removed rows that match the query criteria.
- **Impact:** The result set changes unexpectedly.
- **Prevention:** Use the Serializable isolation level.

T1 (Reads Rows)	T2 (Inserts Rows)
`SELECT * WHERE salary > 5000`	
(Returns 2 rows: Emp1, Emp2)	
	`INSERT Emp3 (salary=6000)`
	`COMMIT`
`SELECT * WHERE salary > 5000`	(Now returns 3 rows: Emp1, Emp2, Emp3)

4. Lost Update

- **Definition:** Two transactions read the same data and update it concurrently, but one update overwrites the other, leading to a loss of data.
- **Impact:** One update is lost, leading to incorrect results.
- **Prevention:** Use locks or the Serializable isolation level.

T1 (Updates A)	T2 (Updates A)
`Read(A=100)`	`Read(A=100)`
`A = A + 20 (A=120)`	
	`A = A - 30 (A=70)`
`Write(A=120)`	
	`Write(A=70)` (Overwrites A=120)

5. Deadlock

- **Definition:** Two or more transactions wait indefinitely for resources locked by each other, creating a circular wait.
- **Impact:** Transactions are unable to proceed, leading to a system stall.
- **Prevention:** Deadlock detection and recovery mechanisms (e.g., timeout, resource ordering).

T1 (Locks Resource A)	T2 (Locks Resource B)
`Lock(A)`	
	`Lock(B)`
`Waits for Lock(B)`	`Waits for Lock(A)`

Techniques for Concurrency Control

1. **Lock-Based Protocols:** Control access using locks (e.g., shared, exclusive).
2. **Timestamp Ordering Protocols:** Assign timestamps to transactions and execute based on these timestamps.
3. **Validation-Based Protocols:** Transactions validate their operations before committing.

Example: Lost Update Problem

Let's consider a bank database:

1. Account Balance = ₹10,000.
2. Transaction 1 (T1): Withdraws ₹2,000.
3. Transaction 2 (T2): Deposits ₹3,000.

Time	T1 (Withdraw ₹2,000)	T2 (Deposit ₹3,000)	Account Balance
T1: Read	₹10,000		₹10,000
T2: Read		₹10,000	₹10,000
T1: Write	₹8,000		₹8,000
T2: Write		₹13,000	₹13,000

Here, T2 overwrites T1's result, leading to a lost update. The correct balance should be ₹11,000, but it's now incorrect.

Concurrency Control Solution
Using locks, T1 locks the account balance during withdrawal. T2 waits until T1 finishes, ensuring the balance is updated correctly.

Lock-Based Protocols in Concurrency Control

- To achieve isolation between transactions, locking operations are used in lock-based protocols.
- Isolation ensures that transactions are executed in such a way that their effects are not visible to others until they are complete.
- By locking operations, transactions are restricted from performing conflicting read/write actions on the same data, maintaining data integrity.

Types of Locks Used

1. Shared Lock (S):

- Prevents write operations but allows read operations on the data.
- It is also called a read-only lock.
- Example: Multiple transactions can read the balance of a bank account, but no one can modify it until the lock is released.
- Symbol: S

2. Exclusive Lock (X):

- Allows both read and write operations.
- This lock is for exclusive access, meaning no other transaction can read or modify the data while the lock is held.
- Example: When one transaction updates a bank account balance, no other transaction can access it until the update is complete.
- Symbol: X

Lock Compatibility

Current Lock	Requested Lock	Compatible?
Shared (S)	Shared (S)	Yes
Shared (S)	Exclusive (X)	No
Exclusive (X)	Shared (S)	No
Exclusive (X)	Exclusive (X)	No

Lock-Based Protocols:

1. Simplistic Lock Protocol:

- **Description:** This is a simple approach where a transaction acquires locks on all data items it needs to modify before performing any write operations. Once the transaction completes the write, it releases the locks.
- **Usage:** It is not widely used because it can lead to unnecessary locking and may cause delays in other transactions.

Example:
Scenario: T1 wants to update the account balance by depositing ₹200, and T2 wants to read the balance.

Steps:

1. T1 acquires a lock on the account balance.
2. T1 updates the balance: 500 + 200 = 700.
3. T1 releases the lock.
4. Now T2 can read the balance (700).

Problem: While T1 is holding the lock, T2 must wait, even though it only wants to read the data and doesn't interfere with the update. This makes it inefficient.

2. Pre-claiming Lock Protocol:

- **Description:** Before a transaction starts executing, it first claims all the locks it needs for the data items involved. It requests all the locks at once, and if all the locks are granted, it proceeds with the operations. If any lock is not granted, the transaction is rolled back.
- **Usage:** This protocol is less flexible because it locks all data at the beginning, which can lead to inefficiencies or deadlocks.

Example:
Scenario: T1 wants to update the balance, and T2 wants to read the balance.

Steps:

1. Before starting, T1 requests a write lock on the balance.
2. If the system grants the lock, T1 updates the balance: 500 + 200 = 700.
3. After completing the update, T1 releases the lock.
4. T2 can now acquire a read lock and read the updated balance (700).

What if locks are unavailable?
If T2 already holds a read lock, T1 rolls back (doesn't proceed with its operation).

Problem: Pre-claiming locks upfront may cause deadlocks (e.g., T1 and T2 waiting for each other's locks) or make transactions unnecessarily wait.

3. Two-Phase Locking Protocol (2PL):

- **Description:** This is the most commonly used protocol in databases. It consists of two phases:

1. **Growing Phase:** The transaction can acquire locks but cannot release any locks.
2. **Shrinking Phase:** Once a transaction releases any lock, it cannot acquire any more locks.

- **Advantages:** It ensures serializability (transactions appear to be executed in a serial order).

Example:

- **Phase 1 (Growing):** T1 requests and acquires locks on data items (read or write).
- **Phase 2 (Shrinking):** Once T1 starts releasing locks, no more locks can be acquired.

Scenario: T1 is transferring ₹200 from account A to account B, and T2 is reading the balance of account A.

Steps:

1. Growing Phase (Lock Acquisition):

- T1 acquires a write lock on account A (to debit ₹200).
- T1 acquires a write lock on account B (to credit ₹200).

2. Shrinking Phase (Lock Release):

- After transferring, T1 releases the lock on account B.
- Then it releases the lock on account A.

Advantages: Transactions follow a systematic order: lock first, work, then release.

Problem: T2 cannot read the balance of account A while T1 is holding the lock.

4. Strict Two-Phase Locking Protocol (Strict 2PL):

- This is a stricter version of the 2PL protocol. It makes sure that the transaction keeps the locks until it's completely finished and committed.

How it works:

- A transaction holds on to its locks and doesn't release them until it is ready to commit (finish and confirm the work).
- Once it commits, it releases all its locks at once.

Why it's useful:

- This avoids situations like cascading rollbacks (when one failure causes many other transactions to fail).
- By holding all locks until commit, it ensures that no other transaction can interfere before the work is finalized.

Scenario: T1 is transferring ₹200 from account A to account B, and T2 is reading the balance of account A.

Steps:

1. T1 acquires a write lock on account A and account B (Growing Phase).
2. T1 performs the transfer: Debit ₹200 from A: 500 - 200 = 300. & Credit ₹200 to B: 1000 + 200 = 1200.
3. T1 holds all locks until it commits.
4. After the transaction commits, T1 releases all locks (Shrinking Phase).

Advantage: Prevents cascading rollbacks. If T1 fails halfway, other transactions like T2 won't see incomplete or incorrect data.

Protocol	Locks Held	Lock Release	Example Use Case
Simplistic Lock Protocol	Locks only while writing	After writing	Updating a single data item without interference.
Pre-claiming Lock Protocol	All locks at the start	After completing the transaction	Safe execution of short transactions with known locks.
Two-Phase Locking (2PL)	Locks during growing phase	Gradually in shrinking phase	Transaction requiring multiple locks in sequence.
Strict 2PL	Locks during transaction	All at commit	Preventing cascading rollbacks in critical updates.

Timestamping Protocols

AKTU - 2023-24 & 2021-22

Timestamping protocols help in maintaining the serializability of transactions without using locks. Each transaction is assigned a unique timestamp that determines the order of execution.

1. Basic Timestamp Ordering Protocol

How it works:

1. Each transaction is assigned a timestamp when it starts.
2. All operations (read/write) are executed in the order of their timestamps.
3. The database maintains two timestamps for each data item:

- **Read Timestamp (RTS):** The largest timestamp of any transaction that successfully read the data.
- **Write Timestamp (WTS):** The largest timestamp of any transaction that successfully wrote the data.

4. **Rules:**

- If a transaction tries to read or write in a way that violates timestamp order, it is aborted and restarted with a new timestamp.

Example:

- T1 (TS = 1) starts first and writes to data item A.
- T2 (TS = 2) tries to read A after T1 has written to it.
- Since T2 > T1, the operation is valid because it respects the timestamp order.

But if T2 tries to write A before T1 completes, it would violate the timestamp order (because T2 is newer) and would be aborted.

2. Strict Timestamp Ordering Protocol

- **How it differs:**

Strict timestamp ordering is more restrictive. It ensures that:

1. Write operations by a transaction are delayed until all earlier transactions (with smaller timestamps) have finished.
2. Read operations are delayed if there is a pending write operation by an earlier transaction.

- **Why it's useful:**

Avoids issues like cascading rollbacks, which occur in basic timestamp ordering.

Example:

- T1 (TS = 1) writes to A.
- T2 (TS = 2) wants to write to A but must wait for T1 to complete.

Even if T2 is ready, it cannot proceed until T1 commits, ensuring strict order.

Key Differences

Aspect	Basic Timestamp Ordering	Strict Timestamp Ordering
Operation Execution	Operations are executed immediately if they obey timestamp rules.	Write operations wait until earlier transactions commit.
Cascading Rollbacks	May occur (due to immediate operations).	Avoided because writes wait for earlier transactions.
Complexity	Simpler but less restrictive.	More restrictive to ensure data consistency.

Example in a Timestamp Queue

Scenario:
Two transactions T1 (TS = 1) and T2 (TS = 2) are working on a bank account balance (initial = ₹500).

Basic Timestamp Ordering:

1. T1 (TS = 1) reads the balance (₹500) and writes a new balance (₹400).
   - RTS(A) = 1, WTS(A) = 1.
2. T2 (TS = 2) tries to write ₹300.
   - Allowed because TS(T2) > WTS(A).
3. T2 writes ₹300, updating WTS(A) = 2.

Result: Final balance = ₹300.

Strict Timestamp Ordering:

1. T1 (TS = 1) reads the balance (₹500) and writes a new balance (₹400).
   - T1 must commit before any operation from T2 can proceed.
2. T2 (TS = 2) waits until T1 commits before writing ₹300.

Result: Final balance = ₹300, but changes are only applied after T1 commits.

Validation-Based Protocols

AKTU - 2021-22 & 2022-23 & 2023-24

Validation-Based Protocols are also known as Optimistic Concurrency Control techniques. These protocols assume that conflicts are rare and allow transactions to execute without locking resources. Conflicts are checked only at the time of validation before committing the transaction.

Working Phases of Validation-Based Protocols

1. Read Phase:

- The transaction reads data from the database and performs calculations or operations locally.
- During this phase, no actual changes are made to the database.
- Data is stored in a temporary workspace (buffer).

2. Validation Phase:

- Before committing, the system validates whether the transaction can be executed without conflicting with other transactions.
- Validation ensures serializability, meaning the transaction can appear to have been executed in isolation.

The validation checks:

- No other transaction has modified the data read by the current transaction.
- The current transaction does not overlap with the execution of conflicting transactions.

3. Write Phase:

- If the transaction passes the validation phase, it writes changes to the database permanently.
- If the validation fails, the transaction is aborted and restarted.

Differences from 2-Phase Commit (2PC)

Aspect	Validation-Based Protocols	2-Phase Commit Protocol (2PC)
Purpose	Used to maintain concurrency without locking resources.	Used for ensuring distributed transaction commit.
Phases	Read, validate, write.	Prepare, commit (or abort).
Concurrency Control	Optimistic; validates conflicts at the end.	Pessimistic; locks resources during execution.
Conflict Handling	Conflicts are resolved at the validation phase.	Conflicts are avoided by locking resources upfront.
Usage	Best for low-conflict environments with high transaction rates.	Best for distributed systems requiring atomic commits.

Example of Validation-Based Protocol

Scenario: Two transactions T1 and T2 updating account balances.
Initial database:

- Account A = ₹500, Account B = ₹1000.

Steps:

1. T1 (Read Phase):

- Reads A = ₹500, B = ₹1000.
- Plans to debit ₹200 from A and credit ₹200 to B.

2. T2 (Read Phase):

- Reads A = ₹500, B = ₹1000.
- Plans to credit ₹300 to A and debit ₹300 from B.

3. Validation Phase:

- T1 checks:
- No other transaction modified A or B after T1 read them.
- T2 checks:
- T1 modified A and B, which conflicts with T2's plans.

4. Write Phase:

- T1 passes validation and writes changes:
- A = ₹300, B = ₹1200.
- T2 fails validation, aborts, and restarts.

Multiple Granularity

What is Granularity?

Granularity refers to the size of the data that can be locked in a database.

- **Small Granularity:** Locking a single record (fine-grained).
- **Large Granularity:** Locking an entire table or database (coarse-grained).

Why Multiple Granularity Locking?

Problem:

- If a transaction (T1) needs to lock the entire database, locking every single record is inefficient.
- If another transaction (T2) needs just one record, locking the whole database affects concurrency unnecessarily.

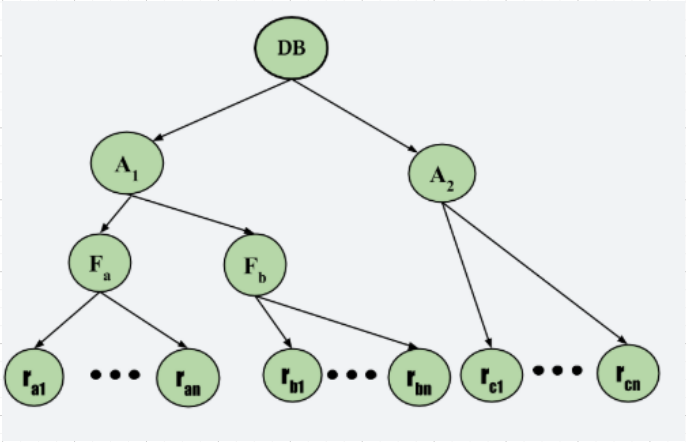
Solution:

- Multiple granularity allows locking different parts of the database (like records, files, or areas) depending on the needs of the transaction. This balance improves both efficiency and concurrency.

Hierarchy of Locks

The database is divided into levels, like a tree:

1. Database (highest level).
2. Area (e.g., a category in the database).
3. File (e.g., a table).
4. Record (lowest level, individual row in a table).



How It Works?

- A transaction can lock any node (e.g., database, table, or record).
- When a node is locked:

All descendants are implicitly locked in the same mode.
For example: If you lock a file (table) in exclusive mode, all its records are locked exclusively.

Intention Locks

To manage hierarchical locks, additional intention lock modes are introduced:

1. **Intention-Shared (IS):**
   - Indicates a plan to place shared locks at lower levels.
   - Example: Locking a table in IS mode to read specific records.
2. **Intention-Exclusive (IX):**
   - Indicates a plan to place exclusive locks at lower levels.
   - Example: Locking a table in IX mode to update specific records.
3. **Shared & Intention-Exclusive (SIX):**
   - Locks a subtree in shared mode but allows exclusive locks at lower levels.
   - Example: Locking a table for reading all records while updating specific ones.

Lock Compatibility Matrix

	IS	IX	S	SIX	X

IS	✔	✔	✔	✔	✗
IX	✔	✔	✗	✗	✗
S	✔	✗	✔	✗	✗
SIX	✔	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

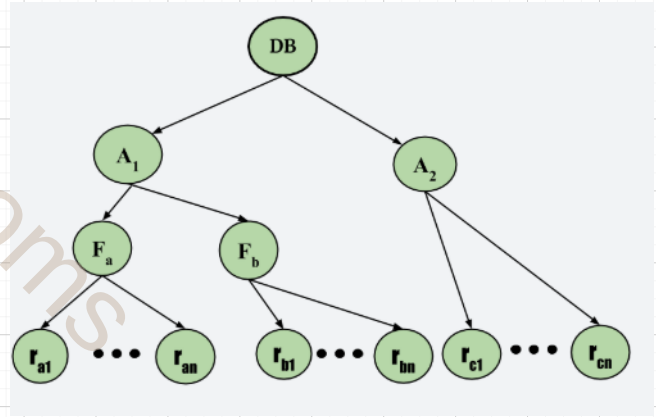
IS : Intention Shared
IX : Intention Exclusive
S : Shared
X : Exclusive
SIX : Shared & Intention Exclusive

Example Scenarios

1. Transaction T1 (Reads a Record)

- Action: T1 reads record Ra2 in file Fa.
- Locks Needed:

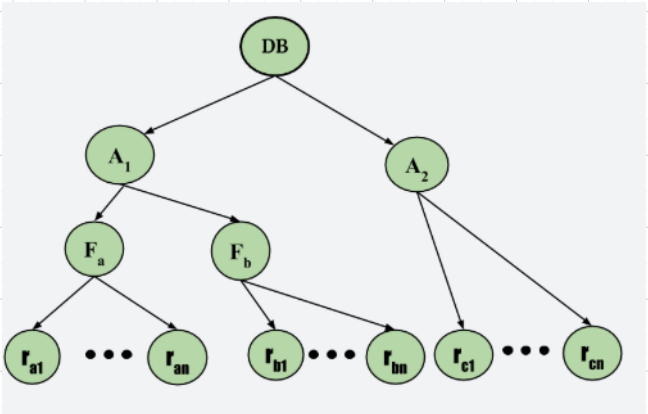
1. Database: IS mode.
2. Area A1: IS mode.
3. File Fa: IS mode.
4. Record Ra2: S mode.



2. Transaction T2 (Updates a Record)

- Action: T2 modifies record Ra9 in file Fa.
- Locks Needed:

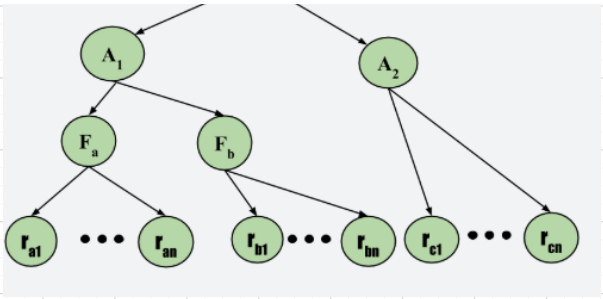
1. Database: IX mode.
2. Area A1: IX mode.
3. File Fa: IX mode.
4. Record Ra9: X mode.



3. Transaction T3 (Reads All Records in a File)

- Action: T3 reads all records in file Fa.
- Locks Needed:

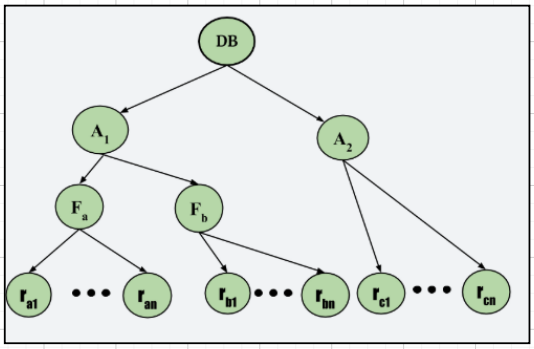
1. Database: IS mode.
2. Area A1: IS mode.
3. File Fa: S mode.



4. Transaction T4 (Reads Entire Database)

- **Action: T4 reads the entire database.**
- **Locks Needed:**

1. **Database: S mode.**



Concurrency in MGL

- **T1, T3, and T4 can run simultaneously because their operations don't conflict.**
- **T2 cannot run with T3 or T4 because it requires exclusive access.**

What is Multi-Version Scheme?

AKTU - 2023-24 & 2021-22

In multi-version schemes, the database maintains multiple versions of the same data item to allow better concurrency and avoid conflicts between transactions.

Why use it?

- To ensure **read consistency** (a transaction reads data as it existed at the start of the transaction, even if other transactions update it).
- To improve **concurrency** by letting read and write operations occur simultaneously.

How Does It Work?

1. Data Versions:

- Each data item has multiple versions, each with a timestamp or identifier.
- A new version is created when a transaction updates the data.
- Older versions remain available for transactions that need them.

2. Read and Write Operations:

- **Read Operation:** A transaction reads the version of the data valid at its start time.
- **Write Operation:** A transaction creates a new version with an updated value and timestamp.

Key Benefits

1. Read Consistency:

- Readers can access the version of the data item valid at the start of their transaction, even if other transactions update the data simultaneously.

2. No Blocking Reads:

- Read operations don't block write operations, improving concurrency.

3. Reduced Conflicts:

- Transactions can proceed without waiting for each other unless they modify the same data.

Examples of Multi-Version Schemes

1. Ticket Booking System
Scenario:

- A customer (T1) checks seat availability for a train.
- Another customer (T2) books a seat on the same train.

Process:

- When T1 starts, it reads the current version of the seat availability.
- T2 updates the seat availability, creating a new version.
- T1 still sees the old version, ensuring read consistency.

2. Banking System (Account Balance Updates)
Scenario:

- Transaction T1 reads an account balance.
- Transaction T2 updates the balance simultaneously.

Process:

- T1 reads the old version of the balance.
- T2 creates a new version after updating the balance.
- T1 finishes without being blocked by T2.

Drawbacks

1. **Increased Storage:**
   - Multiple versions require more storage space.
2. **Complexity:**
   - Managing versions and ensuring they are valid for the right transactions can be challenging.
3. **Garbage Collection:**
   - Old versions that are no longer needed must be removed, adding overhead.

Difference Between Multi-Version and Single-Version Schemes

Feature	Single-Version Scheme	Multi-Version Scheme
Concurrency	Read/write operations may block each other.	Reads are never blocked by writes.
Storage Requirements	Minimal, as only the latest version is stored.	Higher, as multiple versions are stored.
Read Consistency	May not ensure consistency.	Ensures read consistency for all transactions.

What is Recovery with Concurrent Transactions?

When multiple transactions are running simultaneously, the system must ensure that the database remains consistent, even if there's a system failure (like a crash). Recovery techniques help restore the database to a consistent state.

1. Immediate Update Techniques

Definition: Changes made by a transaction are immediately written to the database as soon as they are performed, even before the transaction commits.

Benefit: Reduces recovery time since changes are already in the database.

Challenge: If the system crashes before the transaction commits, the database may become inconsistent.

Log-Based Recovery

A log is a sequential record of all operations performed by transactions. It is critical for recovery.

Steps in Log-Based Recovery

1.Write-Ahead Logging (WAL):

- Changes are logged before being applied to the database.
- Ensures that the log is available for recovery after a crash.

2.Two Types of Log Entries:

- **UNDO Entries:** Used to reverse incomplete transactions.
- **REDO Entries:** Used to reapply committed transactions.

3.During Recovery:

- UNDO Phase: Rollback all incomplete transactions using the log.
- REDO Phase: Reapply all changes of committed transactions.

Example:

Imagine two transactions:

- T1 (incomplete): Updated balance from ₹5000 to ₹4500.
- T2 (committed): Updated balance from ₹4500 to ₹4800.

Recovery steps:

- UNDO T1: Revert balance to ₹5000 using its UNDO log.
- REDO T2: Reapply balance update to ₹4800 using its REDO log.

Checkpoints

To optimize recovery, the system periodically creates checkpoints in the log.

What is a Checkpoint?

- A checkpoint is a snapshot of the current state of the database.
- It marks a point where all committed transactions have been written to the database.

Benefits of Checkpoints:

1.Speeds up Recovery:

- During recovery, the system starts from the last checkpoint instead of scanning the entire log.

2. Reduces Overhead:

- Limits the amount of undo/redo work during recovery.

Example of Checkpoints in Action:
At checkpoint C1, the database writes all changes from committed transactions up to that point.

If the system crashes after C1:

- **Undo:** Transactions started after C1 but didn't commit.
- **Redo:** Transactions committed after C1.

Single/Multi-User Environments

Single-User Environment:

- Easier to handle because only one transaction is active at a time.
- Recovery involves straightforward undo/redo of that transaction.

Multi-User Environment:

- More complex due to concurrent transactions.
- Requires logs, timestamps, and locking protocols to manage recovery and ensure consistency.

Case Study on Oracle: Concurrency Control Techniques

Oracle Database, developed by Oracle Corporation, is one of the most popular relational database management systems (RDBMS) globally. It is designed to handle large-scale data management needs for enterprise applications. Oracle implements robust Concurrency Control Techniques to ensure data integrity, consistency, and isolation when multiple transactions are executed simultaneously.

Key Concurrency Control Techniques in Oracle

1. Multiversion Concurrency Control (MVCC):

- Oracle uses MVCC to provide a non-blocking read mechanism. When a transaction modifies data, the database does not overwrite the original data. Instead, it creates a new version while retaining the old version for other transactions to read.
- This ensures read consistency, where each query sees a consistent snapshot of the database as of the query's start time.

Example:
If a user reads data while another transaction is modifying it, the user will see the old version of the data until the modifying transaction commits.

3. Undo Tablespace:
Oracle uses undo segments to store the original state of modified data. This enables:

- Rollback of uncommitted transactions.
- Generation of consistent snapshots for queries using MVCC.

Impact: Even if a transaction is rolled back, Oracle ensures that other users' views of the data remain unaffected.

4. Automatic Conflict Resolution:
Oracle employs sophisticated algorithms to handle write conflicts between transactions. When a conflict occurs:

- The database determines the transaction with the higher priority based on timing or access order.
- The lower-priority transaction may be delayed or restarted.

Advantages of Oracle's Concurrency Control:

- **High performance with minimal blocking.**
- **Improved scalability for multi-user environments.**
- **Data consistency and integrity are guaranteed.**

Graph-Based Locking Protocols

Aktu- 2023-24

Graph-based locking is a concurrency control mechanism designed to manage data item access and prevent deadlocks by defining a partial ordering of data items. This protocol ensures that transactions access data in a specific order, avoiding circular waits that cause deadlocks.

Key Concepts

1.Partial Ordering of Data Items:

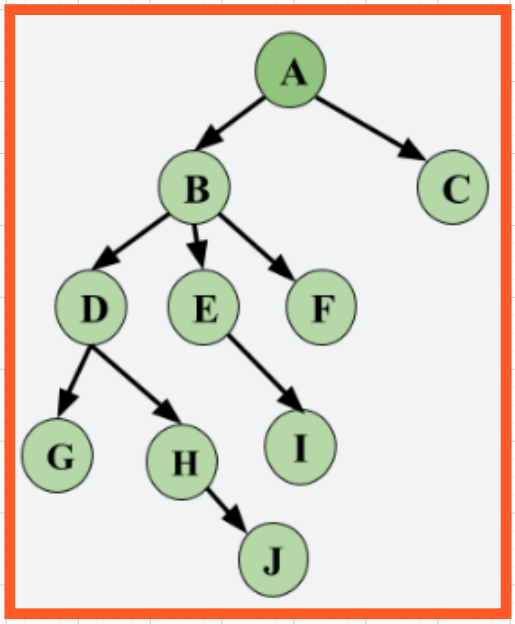
- All data items are organized in a directed acyclic graph (DAG).
- Each node represents a data item.
- Edges represent allowed access sequences (e.g., if there's an edge from A → B, a transaction must lock A before locking B).

2. Protocol Rules:

- A transaction can acquire a lock on a data item only if it has already locked its parent in the graph.
- Once a transaction releases a lock, it cannot acquire any more locks.

3. Deadlock Avoidance:

- By enforcing a strict order of locking based on the graph structure, circular waits are avoided, thereby eliminating deadlocks.



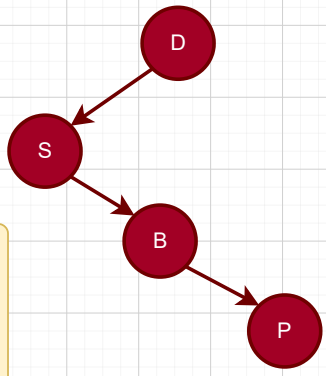
Steps in Graph-Based Locking

1. Identify the hierarchy of data items and arrange them in a DAG.
   (Example: Database → Table → Row → Column.)
2. A transaction locks data items starting from the highest node in the hierarchy.
3. The transaction can only lock children of a locked parent.
4. When finished, the transaction releases locks in a reverse order (from leaf to root).

Example

Hierarchy of Data Items in a Library System:

Database → Section → Book → Page



Transactions:

1. T1: Read a specific page (P1) of a book (B1):
   - Lock order: Database → Section → Book (B1) → Page (P1).
2. T2: Modify a book (B2):
   - Lock order: Database → Section → Book (B2).
3. T3: Read all books in a section:
   - Lock order: Database → Section.

Deadlock Prevention:

- If T1 locks Page P1, T2 cannot directly jump to modify Page P1 without following the lock hierarchy.
- Transactions must respect the lock order, preventing circular dependencies.

Advantages of Graph-Based Locking

- **Deadlock-Free:** Ensures no circular waits by enforcing a strict locking order.
- **Efficient Use of Resources:** Transactions lock only the required data.
- **Improved Concurrency:** Minimizes unnecessary locking.

Disadvantages

- **Complex Implementation:** Managing a DAG for large datasets can be challenging.
- **Reduced Flexibility:** Transactions are forced to follow a rigid locking order.