

---

# **Subject Name: Database Management System (DBMS)**

**Subject Code: BCS501**

## **Unit 1: 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 (DDL), Data Manipulation 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 Diagram to Tables, Extended ER Model.
- Relationship of Higher Degree.

## **Introduction**

### **Overview of Database Systems**

#### **What is Data?**

**Data** refers to raw facts and figures without context. It can be in the form of numbers, text, images, or other formats that are collected for reference or analysis. Data itself does not carry any meaning until it is processed or interpreted.

**Example:** The number '2024' is data, but it does not convey any meaning until we associate it with a year or a quantity.

ASAP

---

## What is a Database?

A **Database** is an organized collection of data that can be easily accessed, managed, and updated. Databases store data in a structured format, using tables, records, and fields, which allows for efficient querying and manipulation of the data.

**Example:** A customer database in a retail store contains information like customer names, addresses, purchase history, and contact details.

## What is a DBMS (Database Management System)?

A **Database Management System (DBMS)** is software that interacts with end-users, applications, and the database itself to capture and analyze data. A DBMS provides an interface for the users to create, update, and manage databases efficiently.

**Example:** Examples of DBMS include Oracle, MySQL, Microsoft SQL Server, MongoDB, and PostgreSQL.

## What is the Need for a DBMS?

A DBMS is needed for the following reasons:

- **Data Redundancy Control:** Minimizes data duplication and ensures data consistency across multiple locations.
- **Data Integrity:** Maintains the accuracy and consistency of data over its lifecycle.
- **Data Security:** Protects sensitive data by controlling access through authentication and authorization mechanisms.
- **Concurrent Access:** Manages multiple users accessing data simultaneously without conflicts.
- **Backup and Recovery:** Provides tools to recover data in case of system failures or data corruption.
- **Data Independence:** Allows changes in data structure without affecting the application programs.

## Example of DBMS:

- **Oracle:** A popular commercial DBMS used for enterprise-level applications.
- **MongoDB:** A NoSQL database that uses a document-oriented data model, suitable for handling unstructured data.
- **MySQL:** An open-source relational DBMS commonly used in web applications.
- **Microsoft SQL Server:** A relational DBMS developed by Microsoft, widely used in corporate environments.
- **PostgreSQL:** An open-source object-relational DBMS known for its robustness and compliance with SQL standards.

---

## Advantages of DBMS

A Database Management System (DBMS) offers several advantages:

- **Data Integrity and Consistency:** Ensures that data remains accurate, consistent, and reliable across the database.
- **Data Security:** Provides robust security measures to protect data from unauthorized access and breaches.
- **Data Independence:** Separates data structure from application programs, allowing for flexibility in modifying the database without affecting the application layer.
- **Efficient Data Access:** Uses indexing, query optimization, and caching techniques to enhance data retrieval and manipulation speed.
- **Concurrent Access and Crash Recovery:** Allows multiple users to access the data simultaneously and ensures data recovery in case of system failures.
- **Reduced Data Redundancy:** Minimizes duplicate data storage by centralizing data in one place, reducing redundancy.
- **Improved Data Sharing:** Facilitates data sharing among multiple users or applications while maintaining data consistency and integrity.

## Disadvantages of DBMS

While a DBMS provides numerous benefits, it also has some disadvantages:

- **High Cost of Implementation:** Setting up a DBMS involves significant costs related to hardware, software, and trained personnel.
- **Complexity:** Requires skilled professionals to manage, maintain, and troubleshoot the system effectively.
- **Performance Overhead:** Due to its general-purpose nature, a DBMS may introduce overhead in terms of processing time and storage requirements.
- **Vulnerability to Failure:** Centralized databases are vulnerable to hardware and software failures, which can affect the entire system.
- **Maintenance and Upgradation Costs:** Continuous maintenance, updates, and backups can be costly and require dedicated resources.

## What is a Database User?

A **Database User** is any person or application that interacts with a database to perform various operations such as querying, updating, or managing the data. Different types of users interact with the database system based on their roles and access rights.

---

## Types of Database Users

- **Database Administrators (DBAs):** Responsible for managing and maintaining the overall database environment, including user management, backup, recovery, and security.  
**Example:** A DBA in a large corporation might configure database servers, monitor performance, and handle disaster recovery planning.
- **Application Programmers:** Developers who write application programs that interact with the database. They use programming languages like Java, Python, or SQL to access and manipulate data.  
**Example:** An application programmer might create an e-commerce application that retrieves product data from a database and displays it on a website.
- **End Users:** The individuals who interact with the database through applications to perform tasks like data entry, retrieval, or reporting.  
**Example:** A bank customer using an online portal to check their account balance is an end user of the database.
- **System Analysts:** Professionals who design and develop the overall system architecture, including database design, to meet business requirements.  
**Example:** A system analyst might work with both end-users and DBAs to create a database schema that supports new business processes.
- **Database Designers:** Individuals responsible for designing the structure of the database, including defining schemas, relationships, and constraints.  
**Example:** A database designer might define the relationships between tables in a hospital management system.
- **Naive Users:** Users who interact with the database through pre-defined applications without writing any queries or using advanced features.  
**Example:** A cashier at a retail store using a point-of-sale system to process transactions is a naive user of the database.

## Who is a Data Administrator?

A **Data Administrator (DA)** is a professional responsible for managing, defining, and maintaining data standards and policies across an organization. The DA focuses on the data itself, rather than the physical aspects of database management. Their primary role is to ensure that the organization's data assets are managed effectively and align with business goals.

## Functions of a Data Administrator

The main functions of a Data Administrator include:

- **Data Modeling:** Designing and creating data models that represent the data structures required by the business, such as entities, relationships, and data flows.  
**Example:** Defining the entities (e.g., customers, orders) and their relationships in a retail database.

- 
- **Data Policy Development:** Establishing policies, standards, and procedures for data management, ensuring data quality, consistency, and security across the organization.

**Example:** Developing guidelines for data entry to reduce errors and maintain consistency.

- **Data Standardization:** Ensuring uniformity in data formats, definitions, and representations to facilitate data integration and interoperability among different systems.

**Example:** Standardizing date formats across multiple databases (e.g., using YYYY-MM-DD).

- **Data Security and Privacy:** Establishing rules and protocols to protect sensitive data from unauthorized access, breaches, and misuse, in compliance with legal and regulatory requirements.

**Example:** Implementing data masking techniques for personal identifiable information (PII).

- **Data Quality Management:** Monitoring and managing data accuracy, completeness, consistency, and reliability to ensure high-quality data across the organization.

**Example:** Setting up data validation rules to detect and correct errors during data entry.

- **Data Lifecycle Management:** Overseeing the complete lifecycle of data, from creation and storage to archiving and deletion, ensuring that data is properly managed throughout its lifespan.

**Example:** Developing retention policies that specify how long certain types of data should be stored.

- **Collaboration with IT and Business Teams:** Working closely with database administrators (DBAs), system analysts, and business users to align data management strategies with organizational objectives.

**Example:** Coordinating with the IT team to ensure data backup and disaster recovery processes are in place.

- **Documentation:** Maintaining comprehensive documentation of data models, standards, policies, and procedures to support data governance and compliance initiatives.

**Example:** Creating a data dictionary that details data definitions, formats, and relationships.

## Data Abstraction

**Data Abstraction** refers to the process of hiding the complexities of the database from the user and providing a simplified view of the data. It helps in managing the large amounts of data stored in the database by abstracting its details, enabling users to interact with the data without needing to understand its internal structure or storage details.

Data abstraction is achieved through three different levels:

- **Physical Level:** This is the lowest level of data abstraction, which describes how the data is physically stored in the database. It deals with the storage of data on storage media, such as hard drives, and the implementation details like file organization, indexing, and data compression techniques.

**Example:** At this level, the data administrator might work with storage blocks, and sectors, or manage how data is indexed in the database for quick retrieval.

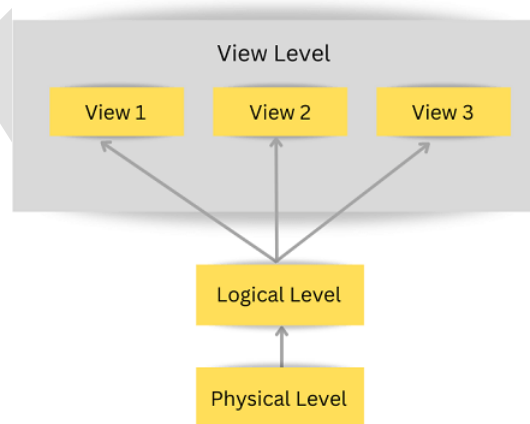
- **Logical Level:** This level provides a higher level of abstraction and focuses on what data is stored in the database and what the relationships are among those data. It describes the structure of the entire database for a group of users. This level is independent of how the data is stored physically and provides a logical view of the data.

**Example:** At this level, the data might be represented using tables, columns, rows, and relationships like one-to-one, one-to-many, or many-to-many, without concern for physical storage details.

- **View Level:** This is the highest level of data abstraction and describes only a part of the entire database. The view level simplifies the interaction for the end-users by providing only the relevant data needed for their specific tasks or applications. It is also used to enhance security by restricting access to certain data.

**Example:** A bank employee might only see the customer details relevant to their role, like name and account balance, without access to sensitive data like Social Security numbers.

### Illustration of Data Abstraction Levels



Levels of Data Abstraction in DBMS

Aspect	Physical Level	Logical Level	View Level
<b>Definition</b>	Describes how data is physically stored in storage devices.	Describes what data is stored and the relationships among those data.	Shows only a subset of the database that is relevant to the user or application.
<b>Focus</b>	Storage structure and access methods.	Overall data structure, schema, and relationships.	User-specific views, simplifying data interaction.
<b>Visibility</b>	Low-level details visible to DBAs only.	Mid-level abstraction visible to developers and designers.	High-level abstraction visible to end-users.
<b>Data Independence</b>	Provides low data independence; changes affect physical storage.	Provides logical data independence; changes do not affect storage.	Offers external data independence; changes do not affect internal schema.
<b>Security</b>	Minimal impact on security; deals with storage details.	Moderate impact; focuses on logical data security.	High impact; restricts user access to sensitive data.
<b>Example</b>	File storage formats, indexing, data compression.	ER diagrams, tables, relationships.	Customer view, employee view, product catalog view.
<b>Users</b>	Database Administrators (DBAs).	Database Designers and Developers.	End-users and Application Programmers.

## Difference Between DBMS and File System

**Example:** Suppose a bank uses a file system to manage its customer information. The system may face challenges like redundancy, inconsistency, and difficulty in managing concurrent access by multiple users. In contrast, a DBMS will handle these issues efficiently by providing features like indexing, data normalization, and transactions.

## Database System Concept and Architecture

### *DBMS Architecture and Its Types*

A Database Management System (DBMS) architecture refers to the design and structure that defines how different components of a DBMS interact with each other. There are primarily three types of DBMS architectures: 1-tier, 2-tier, and 3-tier architecture.

- **1-Tier Architecture:**

In 1-tier architecture, the database is directly accessible to the user without any intermediary application. The user directly interacts with the DBMS, which is usually installed on their local machine. This architecture is mainly used for development purposes, where the developer directly communicates with the database for testing and design.



Aspect	DBMS	File System
<b>Definition</b>	A software system that facilitates the creation, management, and manipulation of databases.	A method for storing, organizing, and retrieving files on a storage device.
<b>Data Redundancy</b>	Minimizes redundancy by using normalization techniques.	High redundancy due to independent file storage, leading to data duplication.
<b>Data Consistency</b>	Ensures data consistency through integrity constraints and transactions.	Lacks mechanisms for maintaining data consistency across multiple files.
<b>Data Security</b>	Provides robust security features, including access control, encryption, and user authentication.	Limited security features; relies on operating system security measures.
<b>Backup and Recovery</b>	Offers automated and systematic backup and recovery processes.	Backup and recovery processes are manual and less reliable.
<b>Data Access</b>	Supports complex querying and data manipulation using SQL or similar languages.	Limited to basic file operations (create, read, update, delete).
<b>Concurrency Control</b>	Manages multiple users accessing the data simultaneously through concurrency control mechanisms.	Lacks concurrency control; file locking is often required to prevent conflicts.
<b>Data Integrity</b>	Maintains data integrity through constraints, triggers, and rules.	No built-in support for enforcing data integrity rules.
<b>Performance</b>	Optimized for large-scale data management and complex operations.	May have performance issues with large volumes of data or complex operations.

Table 1: Differences Between DBMS and File System

**Example:** SQL\*Plus, Oracle Forms, etc., where the developer interacts directly with the database system.

- **2-Tier Architecture:**

In 2-tier architecture, the DBMS system is split into two parts: the client side and the server side. The client directly communicates with the database server. This type of architecture is used in small to medium-sized applications where the client (user interface) directly connects to the server (database) through an application

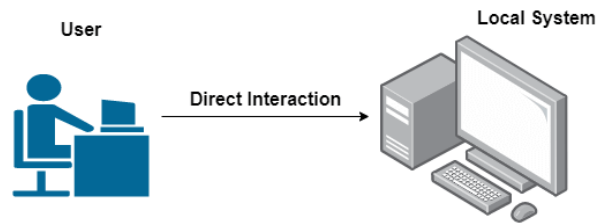
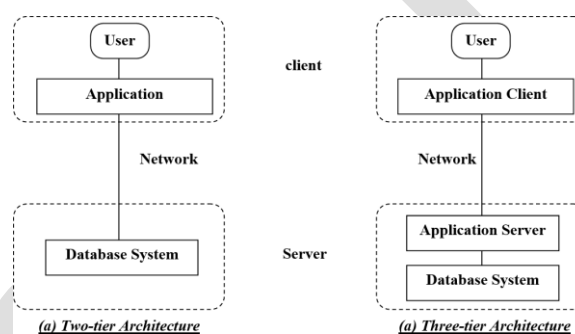


Figure 1: 1-Tier Architecture of DBMS

interface like ODBC or JDBC.

**Example:** Applications using client-server models like Microsoft Access and Fox-Pro.



### • 3-Tier Architecture:

The 3-tier architecture is the most commonly used architecture for DBMS systems. It divides the application into three layers: the presentation layer (client), the application layer (business logic), and the database layer (server). The client interacts with the application server, which further communicates with the database server. This architecture offers better security, scalability, and flexibility.

**Example:** Web applications where the client (browser) sends requests to the web server (application server), which then interacts with the database server.

article graphicx amsmath

## Data Models in Databases

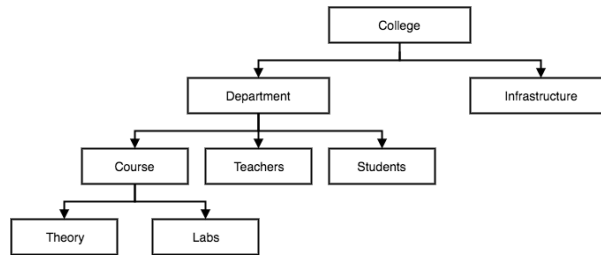
A data model defines how data is structured, stored, and manipulated in a database. It provides a framework for representing relationships between different data elements.

### Types of Data Models

#### – Hierarchical Data Model

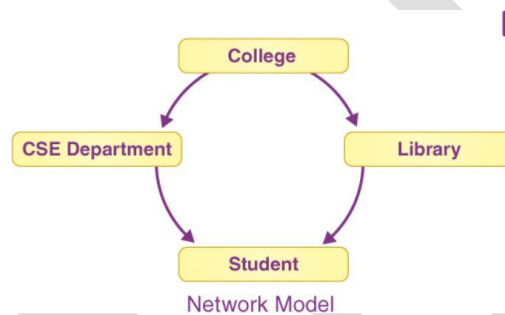
- \* **Explanation:** Organizes data in a tree-like structure where each record has a single parent and multiple children. Suitable for applications with hierarchical data relationships.

\*



## – Network Data Model

- \* **Explanation:** Extends the hierarchical model by allowing multiple relationships between records (many-to-many). Uses a graph structure where nodes represent records and edges represent relationships.



\*

## – Relational Data Model

- \* **Explanation:** Organizes data in tables (relations) consisting of rows and columns. Each table has a unique key, and relationships between tables are defined using foreign keys.

### Relational Model in DBMS

Course	Duration	Type
Data Science	5 Months	Cohort Based
Full Stack	5 Months	Cohort Based
Software Development	6 Months	1:1
Product Management	4 Months	Cohort Based

Primary Key

Tuples(Rows)

Attributes (Columns)

\*

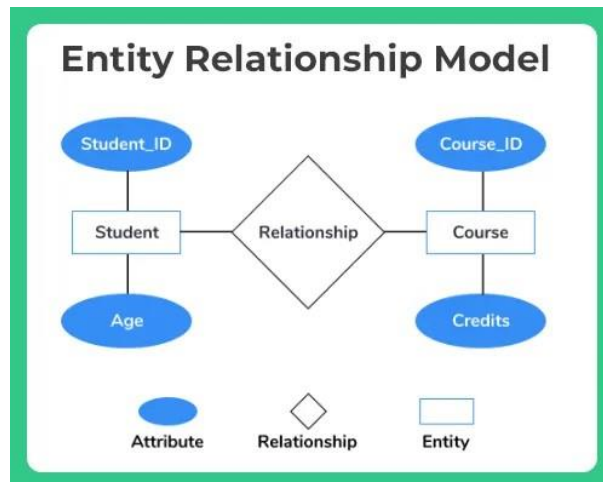
BCQRD

## – Entity-Relationship (ER) Model

- \* **Explanation:** Represents data using entities (objects) and relationships between them. Widely used for conceptual modeling of databases.

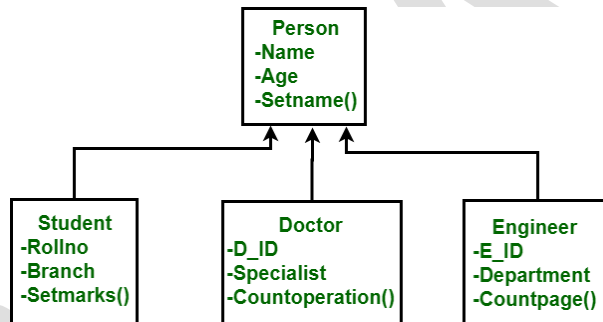
Figures	Symbols	Represents
Rectangle		Entities in ER Model
Ellipse		Attributes in ER Model
Diamond		Relationships among Entities
Line		Attributes to Entities and Entity Sets with Other Relationship Types
Double Ellipse		Multi-Valued Attributes
Double Rectangle		Weak Entity

\* newline



### • Object-Oriented Data Model

- **Explanation:** Combines object-oriented programming principles with database management. Data is represented as objects, similar to classes in object-oriented languages.



### Data Schema

- **Definition:** The logical structure of the database that defines the organization of data, such as tables, fields, and relationships.

### Instances

- **Definition:** The actual content stored in the database at a given time. For example, in a relational model, instances are the rows stored in a table.

### Data Independence and Database Language

- **Data Independence:** The capacity to change the schema at one level of a database system without having to change the schema at the next higher level. -
- **Logical Data Independence:** Modification in conceptual schema without altering external schema. -
- **Physical Data Independence:** Modification in internal schema without altering conceptual schema.

article amsmath

---

## Database Languages and Interfaces

Database languages are used to define, manipulate, control, and query the data within a database. They include several types:

### – DDL (Data Definition Language)

- \* Used to define the structure of the database, such as creating, altering, or deleting tables and other objects.
- \* Commands include CREATE, ALTER, DROP, and TRUNCATE.
- \* Examples: Defining the schema of a table or modifying an existing table structure.

### – DML (Data Manipulation Language)

- \* Used for accessing and manipulating data stored in the database.
- \* Commands include INSERT, UPDATE, DELETE, and MERGE.
- \* Examples: Inserting a new record, updating an existing record, or deleting records from a table.

### – DCL (Data Control Language)

- \* Used to control access to data within a database.
- \* Commands include GRANT and REVOKE.
- \* Examples: Granting or revoking permissions to a user or role.

### – DQL (Data Query Language)

- \* Used to query or retrieve data from the database.
- \* The primary command is SELECT.
- \* Examples: Fetching data from one or more tables, filtering data using conditions, or joining tables.

### – VDL (View Definition Language)

- \* Used to define and manage views in the database.
- \* Commands include CREATE VIEW and DROP VIEW.
- \* Examples: Creating a virtual table that is a result set of a SELECT query.

**Example:** SQL (Structured Query Language) is a widely used language that includes commands for DDL, DML, DCL, DQL, and VDL.

- **DBMS Interface:** A DBMS (Database Management System) interface provides a means for users to interact with the database. Different types of interfaces are designed to cater to various user requirements, from casual users to advanced administrators.

### – Menu-Based Interface

- 
- \* Provides a list of options or commands in a menu format.
  - \* Users can navigate through different menus to execute specific database operations.
  - \* Commonly used in applications where users prefer easy and guided navigation.

#### – **Forms-Based Interface**

- \* Allows users to enter data and interact with the database using forms.
- \* Suitable for data entry tasks where structured input is required.
- \* Often used in applications where non-technical users interact with the database.

#### – **Graphical User Interface (GUI)**

- \* Provides a visual interface with icons, buttons, and other graphical elements.
- \* Allows users to interact with the database through point-and-click actions.
- \* Commonly used in modern applications to enhance usability and user experience.

#### – **Natural Language Interface**

- \* Enables users to interact with the database using natural language queries.
- \* Suitable for users who are not familiar with query languages like SQL.
- \* Relies on natural language processing (NLP) to interpret user input.

#### – **Speech Input and Output Interface**

- \* Allows interaction with the database through voice commands.
- \* Can provide audio feedback or responses from the database.
- \* Useful in applications where hands-free operation is necessary.

#### – **Interfaces for Database Administrators (DBAs)**

- \* Provides advanced tools and options for managing the database system.
- \* Includes functionalities like performance monitoring, backup, and security management.
- \* Tailored for experienced users who need to maintain and optimize the database.

---

## Overall Database Structure

A DBMS (Database Management System) supports various operations for creating, maintaining, and manipulating databases. It consists of multiple components that work together to provide efficient data management.

### – Storage Manager:

- \* Responsible for managing the storage of data on disk.
- \* Handles tasks like data allocation, organization, retrieval, and buffering.
- \* Components include the *buffer manager*, *file manager*, and *disk space manager*.

### – Query Processor:

- \* Translates high-level SQL queries into low-level instructions that the database engine can execute.
- \* Performs query parsing, optimization, and execution.
- \* Includes components like the *query parser*, *query optimizer*, and *query executor*.

### – Transaction Manager:

- \* Ensures that all database transactions are processed reliably and adhere to ACID properties (Atomicity, Consistency, Isolation, Durability).
- \* Manages transaction logs, concurrency control, and recovery processes.
- \* Includes components like the *lock manager* and *log manager*.

### – Buffer Manager:

- \* Manages the buffer pool in main memory to reduce disk I/O operations.
- \* Decides which data pages to cache in memory and which to flush back to disk.
- \* Ensures efficient data retrieval by optimizing access patterns.

### – Index Manager:

- \* Handles the creation, maintenance, and use of indexes to speed up data retrieval.
- \* Manages index structures like B-trees, hash tables, and bitmap indexes.
- \* Improves query performance by reducing the search space for data.

### – Authorization and Integrity Manager:

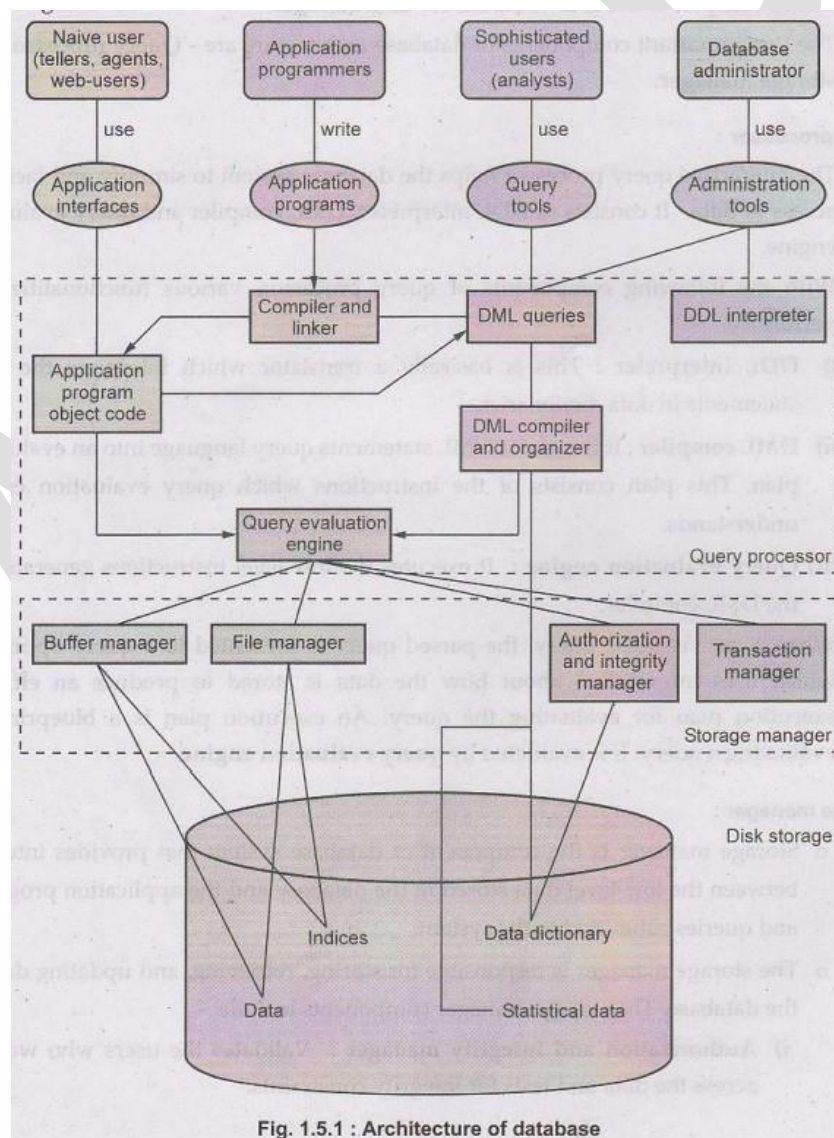
- \* Enforces security policies and user access control.
- \* Ensures data integrity constraints (e.g., primary key, foreign key, check constraints) are met.
- \* Manages user roles, permissions, and auditing.

### – Metadata Manager:

- \* Manages metadata, which includes information about the database schema, objects, and storage.
- \* Maintains data dictionaries and system catalogs.
- \* Provides information about data structures and their relationships.

### – Recovery Manager:

- \* Ensures database consistency and durability in case of failures.
- \* Implements techniques for backup, restore, and log-based recovery.
- \* Works closely with the transaction manager to provide rollback and commit functionalities.



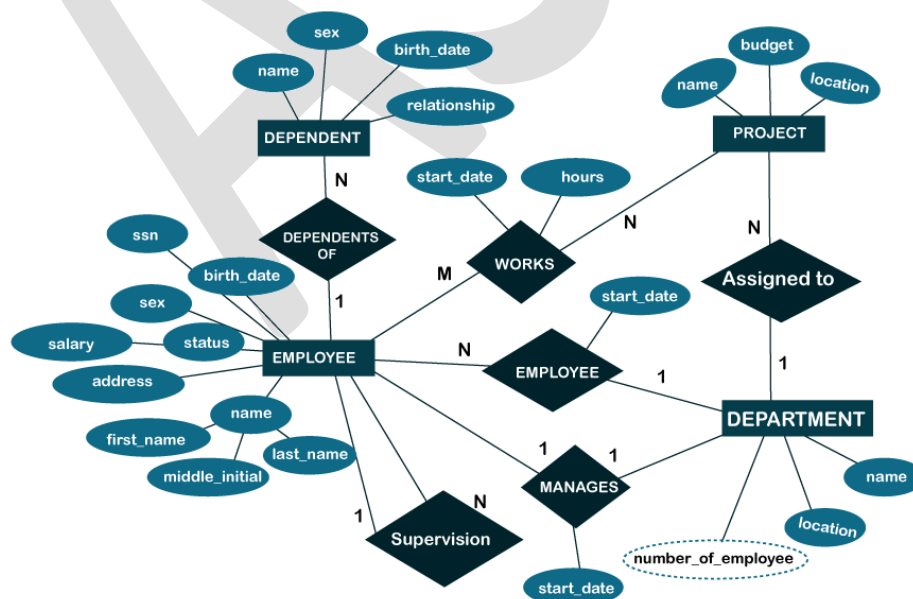


## – Data Modeling Using the Entity-Relationship (ER) Model:

- \* **ER Model Concepts:** The ER model is a high-level data model that defines the data elements and their relationships in a database system. It uses entities, attributes, and relationships to represent the real-world data. An
- \* *entity* represents an object or thing in the real world, an
- \* *attribute* represents properties of an entity, and a
- \* *relationship* represents associations between entities.
- \* **Notation for ER Diagram:** An ER diagram uses various symbols to depict entities (usually as rectangles), attributes (as ovals), and relationships (as diamonds). Lines connect these elements to show how they interact or are associated with each other.

### Purpose of an ER Diagram:

- \* **Visual Representation:** ER diagrams provide a visual representation of the database structure, making it easier for stakeholders, including database designers, developers, and non-technical users, to understand the relationships between different entities within the database.
- \* **Database Design:** Helps in the conceptual design phase of the database, allowing designers to map out the entities, attributes, and relationships before implementing the database. This reduces the risk of design errors and ensures a more organized and efficient database schema.
- \* **Clarifying Requirements:** Facilitates communication between database designers, developers, and business analysts by providing a clear and detailed diagram of the data requirements. This ensures that the final database design meets the business needs and expectations.

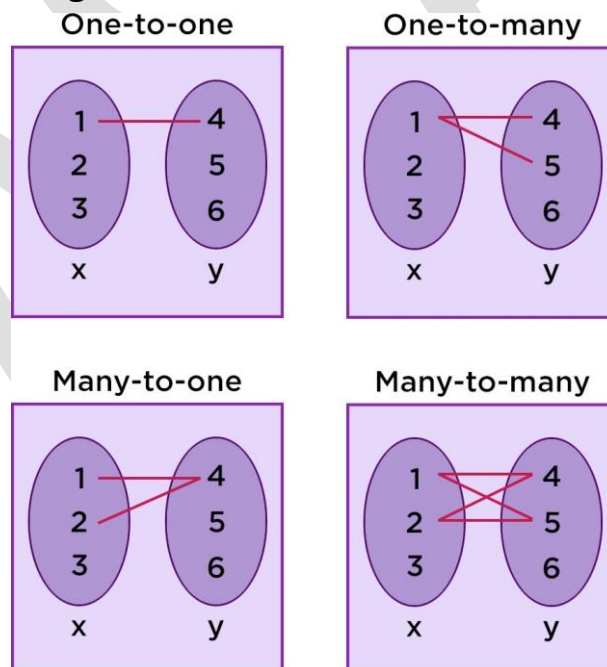


## – Mapping Constraints and Keys:

\* **Mapping Constraints:** Define how entities are associated with one another in a database. They specify the cardinality of relationships, which determines the number of occurrences of one entity that can be associated with occurrences of another entity. The types include:

- *One-to-One (1:1):* Each entity in the first entity set is associated with at most one entity in the second entity set, and vice versa.
- *One-to-Many (1:N):* An entity in the first entity set can be associated with multiple entities in the second entity set, but each entity in the second entity set is associated with at most one entity in the first entity set.
- *Many-to-One (M:1):* Multiple entities in the first entity set can be associated with a single entity in the second entity set, but each entity in the second entity set can be associated with at most one entity in the first entity set.
- *Many-to-Many (M:N):* Entities in the first entity set can be associated with multiple entities in the second entity set, and entities in the second entity set can be associated with multiple entities in the first entity set.

**Image:**



item **Keys:** Attributes or sets of attributes used to uniquely identify entities within an entity set. There are different types of keys:

- **Super Key:** A set of one or more attributes that can uniquely identify an entity. It may contain additional attributes beyond what is necessary for uniqueness.

**Example:** In a student database, a combination of *StudentID* and

*Email* can be a super key if each combination uniquely identifies a student.

- **Candidate Key:** A minimal super key, meaning it contains no redundant attributes. Each candidate key can uniquely identify an entity without any unnecessary attributes.

**Example:** In the same student database, *StudentID* alone can be a candidate key if it is sufficient to uniquely identify each student.

- **Primary Key:** A candidate key chosen by the database designer to uniquely identify each entity within an entity set. It must be unique and not null.

**Example:** *StudentID* might be chosen as the primary key in the student database because it uniquely identifies each student and is not null.

- **Composite Key:** A key that consists of two or more attributes that together uniquely identify an entity.

**Example:** In an enrollment database, a combination of *StudentID* and *CourseID* might be used as a composite key to uniquely identify each enrollment record.

- **Alternate Key:** A candidate key that was not chosen as the primary key but can still uniquely identify an entity.

**Example:** In the student database, *Email* might be an alternate key if *StudentID* is chosen as the primary key.

- **Foreign Key:** An attribute or set of attributes in one table that refers to the primary key in another table. It establishes a link between the two tables.

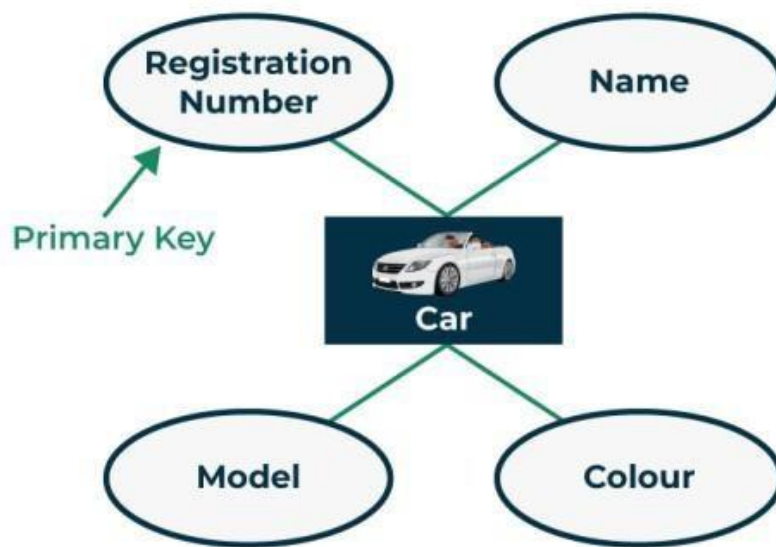
**Example:** In an enrollment database, *CourseID* in the Enrollment table might be a foreign key that references the *CourseID* primary key in the Course table.

article geometry a4paper, margin=1in graphicx array

Table 2: Differences Between Super Key, Candidate Key, and Primary Key

Key Type	Super Key
<b>Definition</b>	A set of one or more attributes that can uniquely identify an entity.
<b>Uniqueness</b>	Must uniquely identify each entity, but can include extra attributes.
<b>Redundancy</b>	May contain redundant attributes.
Key Type	Candidate Key
<b>Definition</b>	A minimal super key, meaning it has no redundant attributes.
<b>Uniqueness</b>	Uniquely identifies each entity without any unnecessary attributes.
<b>Redundancy</b>	No redundancy; minimal set of attributes needed for uniqueness.

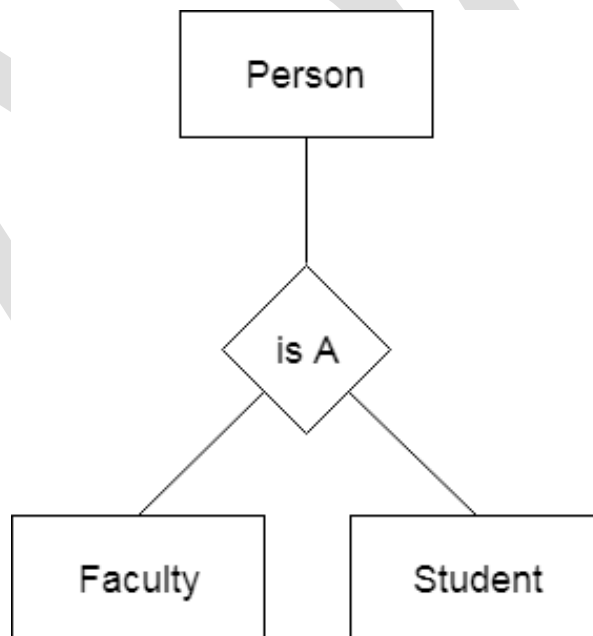
Key Type	Primary Key
<b>Definition</b>	A candidate key selected by the database designer to uniquely identify entities.
<b>Uniqueness</b>	Must be unique and not null.
<b>Redundancy</b>	Does not contain redundancy; chosen from among candidate keys.



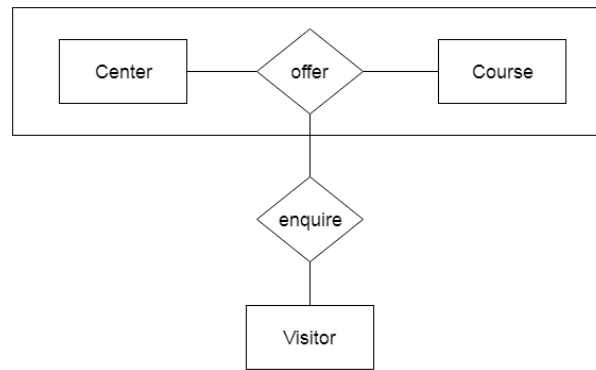
ER Diagram of Strong Entity Set

\* **Generalization, Aggregation, and Reduction of an ER Diagram to Tables:**

- **Generalization:** A process of extracting common characteristics from multiple entities and creating a generalized entity that represents these shared characteristics. This is often used in hierarchical data modeling.



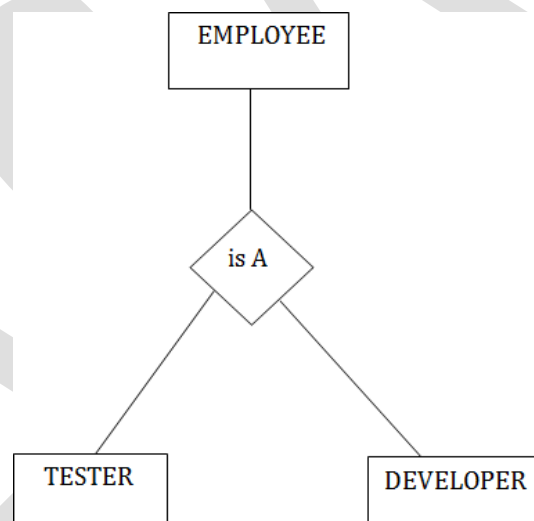
- **Aggregation:** A concept used to express a relationship between a relationship set and an entity set. It is used when we need to express a relationship between an entity and the relationship itself.



## Specialization

### Definition

Specialization is a process where a general entity is divided into more specific sub-entities or subclasses. Each subclass inherits attributes and relationships from the general entity but may also have additional attributes or relationships.



gin=1in array

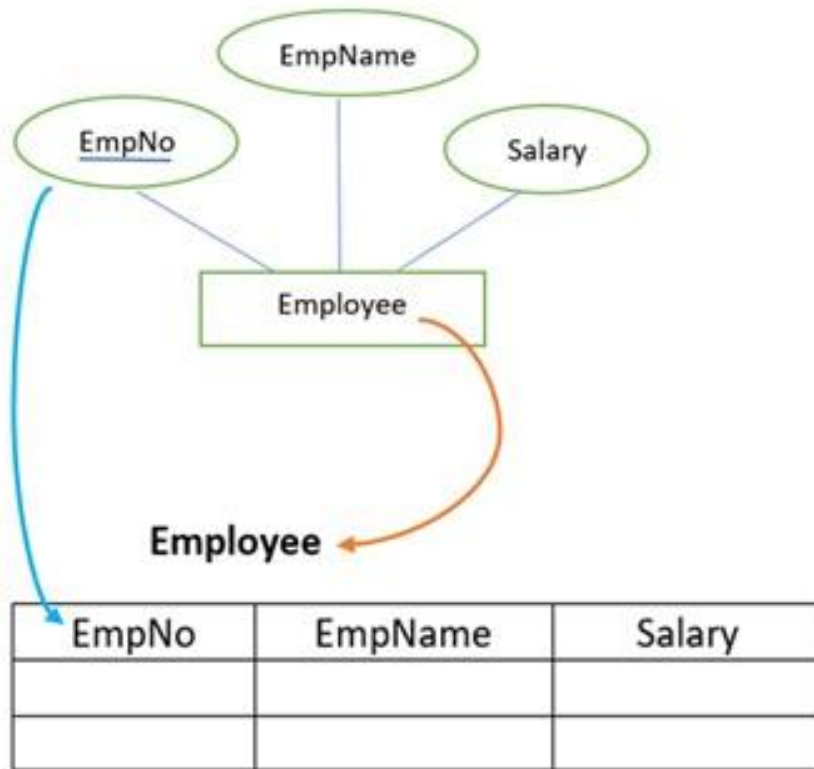
### Differences Between Generalization, Specialization, and Aggregation

Aspect	Generalization	Specialization	Aggregation
<b>Definition</b>	The process of extracting common characteristics from multiple entities and combining them into a generalized entity.	The process of defining a new subclass from an existing class to capture more specific characteristics.	A process of creating a higher-level abstraction by combining several entities into a single entity.
<b>Purpose</b>	To simplify and unify similar entities by identifying common attributes.	To refine and categorize a general entity into more specific sub-entities.	To simplify complex ER diagrams by grouping related entities into a single abstraction.
<b>Direction</b>	From specific entities to a generalized entity (top-down).	From a generalized entity to more specific sub-entities (bottom-up).	Combining multiple entities into a higher-level entity (horizontal aggregation).
<b>Use Case</b>	Useful when multiple entities share common attributes or relationships.	Useful when an entity needs to be divided into more detailed types to represent specific characteristics.	Useful when representing complex relationships between entities in a simplified manner.
<b>ER Diagram Representation</b>	Generalization is often represented with a triangle pointing to a single generalized entity.	Specialization is represented with a hierarchy, where a general entity is at the top and specific sub-entities are below.	Aggregation is represented by a diamond or an oval encompassing multiple entities to show a higher-level relationship.

Table 3: Differences Between Generalization, Specialization, and Aggregation

- **Reduction of an ER Diagram to Tables:** The process of converting an ER model into relational tables involves several steps. Each entity, relationship, and attribute in the ER diagram is transformed into tables, columns, and keys in the relational model. This process ensures that the database design can be implemented effectively in a relational database management system (RDBMS). The steps involved are:
  1. **Convert Entities to Tables:**
  5. **Convert Relationships to Tables:**
  6. For each relationship in the ER diagram, a corresponding table is created if the relationship has attributes of its own or if it is a many-to-many relationship.
  7. The primary key of the relationship table includes the primary keys of the participating entities as foreign keys.

- 
8. If the relationship has attributes, these attributes become additional columns in the relationship table.
  9. **Handle Primary and Foreign Keys:**
  10. Primary keys in entity tables are used to establish relationships with other tables.
  11. Foreign keys are added to tables to represent the relationships between entities. They are columns that reference the primary keys of other tables.
  12. Ensure that referential integrity is maintained, meaning that foreign keys must correspond to valid primary keys in the referenced tables.
  13. **Convert Multi-Valued Attributes:**
  14. Multi-valued attributes (attributes that can have multiple values for a single entity) are handled by creating a new table.
  15. The new table includes a foreign key that references the primary key of the entity and a column for the multi-valued attribute.
  16. This table essentially captures the one-to-many relationship between the original entity and the multi-valued attribute.
  17. **Convert Weak Entities:**
  18. Weak entities (entities that do not have a sufficient primary key on their own) are handled by creating a table that includes the primary key of the strong entity it depends on.
  19. The table includes the partial key of the weak entity along with the foreign key from the strong entity.
  20. The combination of the strong entity's primary key and the weak entity's partial key forms the primary key of the weak entity's table.
  21. **Normalize the Tables:**
  22. After creating the tables, normalize them to eliminate redundancy and ensure data integrity.
  23. Apply normalization rules (1NF, 2NF, 3NF) to ensure that the tables are free from anomalies and that the relationships between tables are accurately represented.



- **Extended ER Model (EER):** Extends the original ER model by adding more modeling constructs, such as specialization, generalization, categorization, and inheritance, to better represent more complex database designs.

article geometry a4paper, margin=1in

## Unified Modeling Language (UML) Diagrams

### Activity Diagram

- **Purpose:** Represents the workflow of a system or business process. It shows the sequence of activities and the flow of control from one activity to another.
- **Components:**
  - **Activities:** Represent tasks or operations performed in the workflow.
  - **Transitions:** Arrows showing the flow from one activity to another.
  - **Decision Nodes:** Points where the flow can branch based on conditions.
  - **Start/End Nodes:** Indicate the beginning and end of the workflow.
  - **Use Cases:** Modeling business processes, system workflows, and use case scenarios.



---

## Use Case Diagram

- **Purpose:** Shows the interactions between users (actors) and the system. It defines the functional requirements of the system from the user's perspective.
- **Components:**
- **Actors:** Represent users or other systems interacting with the system.
- **Use Cases:** Functionalities or services provided by the system.
- **Relationships:** Lines connecting actors to use cases, including associations, generalizations, and include/extend relationships.
- **Use Cases:** Understanding system requirements and user interactions.

## Interaction Overview Diagram

- **Purpose:** Provides a high-level view of the interactions within a system, combining aspects of activity and sequence diagrams.
- **Components:**
- **Activities:** Represent the actions or processes.
- **Interactions:** Sub-diagrams showing detailed interactions.
- **Control Flows:** Show the flow of control between activities and interactions.
- **Use Cases:** Analyzing overall system interactions and workflows.

## Timing Diagram

- **Purpose:** Illustrates the behavior of objects over time, showing how they interact at specific time points.
- **Components:**
- **Lifelines:** Represent objects or participants in the interaction.
- **Time Intervals:** Show the passage of time along the diagram.
- **Events:** Indicate actions or changes occurring over time.
- **Use Cases:** Modeling time-dependent behaviors and interactions.

## Sequence Diagram

- **Purpose:** Shows how objects interact in a particular sequence of events, focusing on the order of messages exchanged.
- **Components:**
- **Objects:** Entities that participate in the interaction.
- **Messages:** Arrows showing the communication between objects.
- **Lifelines:** Vertical dashed lines representing the lifetime of objects.
- **Use Cases:** Detailing interactions between objects and understanding the flow of messages.

---

## Class Diagram

- **Purpose:** Describes the static structure of a system, showing classes, attributes, methods, and the relationships between classes.
- **Components:**
  - **Classes:** Represent entities with attributes and methods.
  - **Attributes:** Data members of a class.
  - **Methods:** Functions or operations of a class.
  - **Relationships:** Include associations, generalizations, and dependencies between classes.
- **Use Cases:** Designing and understanding the structure of the system.

**Subject Name: DATABASE MANAGEMENT  
SYSTEM**

**Subject Code: BCS501**

**UNIT 2**

**UNIT 2 SYLLABUS**

- Relational Data Model Concepts
- Integrity Constraints: Entity Integrity, Referential Integrity
- Key Constraints, Domain Constraints
- Relational Algebra
- Relational Calculus: Tuple Calculus, Domain Calculus
- SQL Introduction: Characteristics, Advantages
- SQL Data Types and Literals
- SQL Commands: Types
- 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

ASAP

# 1 Relational Data Model Concepts

The relational data model is one of the most popular models for organizing data in databases. In this model, data is structured into relations, which are conceptually represented as tables. Each table consists of rows (also known as tuples) and columns (called attributes). The relational model was proposed by E. F. Codd in 1970 and forms the foundation of relational databases such as MySQL, PostgreSQL, and Oracle.

**Definition 1:** A relation is a two-dimensional table with the following characteristics:

- Each row in the table represents a unique tuple.
- Each column in the table represents an attribute of the data.
- All values in a column come from the same domain (a predefined set of values, such as integers, characters, etc.).

**Definition 2:** In the relational data model, relationships between different data entities are represented through foreign keys and primary keys, ensuring data consistency and referential integrity.

**Key Components of the Relational Model:**

- Tables (Relations): A collection of data organized into rows and columns.
- Attributes: Columns in a table that represent properties or characteristics of the data.
- Tuples: Rows in a table that represent individual records or data points.
- Primary Key: A unique attribute (or a combination of attributes) used to identify each tuple in a relation.
- Foreign Key: An attribute in one table that refers to the primary key of another table, establishing a relationship between the two.

**Example 1:**

Consider a table 'Students' that stores basic information about students in a college. The table contains the following attributes: 'StudentID', 'Name', and 'Age'.

StudentID	Name	Age
101	Shyam	20
102	Ram	21
103	Sita	19
104	Radha	22
105	Mohan	20

**Explanation:**

- Each row in this table represents a tuple (or record) of a student.
- The columns represent the attributes of the students, such as 'StudentID', 'Name', and 'Age'.
- The StudentID is a unique identifier for each student, which can be considered as the \*\*primary key.

- The Age attribute is restricted to a specific domain (positive integers).

### Example 2:

Now consider a second table called 'Courses', which lists the courses taken by the students:

CourseID	StudentID	CourseName
501	101	Database Systems
502	102	Operating Systems
503	101	Data Structures
504	104	Algorithms
505	103	Computer Networks

### Explanation:

- The 'Courses' table has three attributes: 'CourseID', 'StudentID', and 'CourseName'.
- The 'StudentID' here is a foreign key that references the 'StudentID' in the 'Students' table. This establishes a relationship between the 'Students' and 'Courses' tables.
- For example, the record with 'StudentID = 101' in the 'Courses' table indicates that student Shyam (from the 'Students' table) is enrolled in the courses "Database Systems" and "Data Structures."

**Relationship Between Tables:** The two tables ('Students' and 'Courses') are linked via the 'StudentID' column. The use of foreign keys allows for the establishment of relationships across multiple tables without duplicating data. This concept of referential integrity ensures that the relationships between tables remain consistent.

### Relational Model in Action

The relational model enables you to efficiently query and manipulate data. For instance, you can retrieve all the courses taken by a specific student (e.g., Shyam) using the 'StudentID' as the common link between the 'Students' and 'Courses' tables.

### SQL Query Example:

To retrieve the courses taken by student 'Shyam':

```
SELECT CourseName FROM Courses C JOIN Students S ON C.StudentID = S.StudentID WHERE S.Name = 'Shyam';
```

This query fetches all the courses taken by Shyam from both the 'Courses' and 'Students' tables by leveraging the relational model's linking feature.

## 2 Constraints and Their Types

In relational databases, constraints are rules applied to the data to maintain accuracy, consistency, and reliability. There are several types of constraints:

### 2.1 Types of Constraints

- **NOT NULL:** Ensures that a column cannot have a null (empty) value.
- **UNIQUE:** Ensures all values in a column are distinct.
- **DEFAULT:** Provides a default value for a column when no value is specified.
- **CHECK:** Ensures that all values in a column satisfy a specific condition.
- **Key Constraints:**
  - **Primary Key:** A column (or a combination of columns) that uniquely identifies each row in a table.
  - **Foreign Key:** A column in one table that is a primary key in another table, used to establish a link between the two tables.
- **Domain Constraints:** Ensures that all values in a column fall within a specified domain (set of acceptable values).

## 3 Integrity Constraints

Integrity constraints are essential for ensuring that the data in the database is accurate and remains consistent across operations. There are three main types of integrity constraints:

**Entity Integrity:** Ensures that the primary key of a table is unique and does not contain null values. Every row must have a unique identifier.

**Referential Integrity:** Ensures that foreign keys in a table accurately reference valid primary keys in another table, ensuring consistent relationships between tables.

**Domain Integrity:** Ensures that the values entered into a column are valid according to the domain constraint set for that column (e.g., a column for age can only accept integer values between 0 and 100).

**Key Integrity:** Ensures that the key constraints (primary key and foreign key) are always maintained.

**Unique Integrity:** Ensures that specific columns marked as unique will have distinct values in every row.

**Null Integrity:** Ensures that the columns marked as NOT NULL will always contain a value.

**Check Integrity:** Ensures that the CHECK constraints on a column will restrict the data to a specific condition (e.g., salary  $\geq$  0).

## 4 Example of Integrity Constraints

To understand integrity constraints, let's consider the following example of two tables, *Students* and *Courses*:

**Students Table:**

StudentID	Name	Age
101	Ram	20
102	Shyam	21

**Courses Table:**

CourseID	StudentID	CourseName
501	101	Database Systems
502	102	Operating Systems

**Entity Integrity Constraint:** In the *Students* table, the primary key is 'StudentID'. According to the entity integrity constraint: - No 'StudentID' can be null. - Every 'StudentID' must be unique.

Thus, the table ensures that each student is uniquely identifiable, and no student can have a missing or duplicate 'StudentID'.

**Referential Integrity Constraint:** In the *Courses* table, 'StudentID' is a foreign key that references the 'StudentID' in the *Students* table. The referential integrity constraint ensures that: - Every 'StudentID' in the *Courses* table must exist in the *Students* table.

For example, if we try to add a course with 'StudentID = 103', it would violate the referential integrity constraint because no student with 'StudentID = 103' exists in the *Students* table.

**Domain Constraint:** A domain constraint specifies the permissible values for a column. For instance: - The 'Age' column in the *Students* table should only accept positive integer values.

If we try to enter a value like 'Age = -5', it would violate the domain constraint.

## 5 Key Constraints and Domain Constraints

**Key Constraints:** These constraints ensure that each row in the table can be uniquely identified.

- **Primary Key:** In the *Students* table, the 'StudentID' is the primary key, ensuring that each student has a unique identifier.
- **Foreign Key:** In the *Courses* table, the 'StudentID' acts as a foreign key referencing the primary key in the *Students* table, maintaining a relationship between the two tables.

**Domain Constraints:** These constraints define the valid set of values for a column. For example:

- The 'Age' column in the *Students* table has a domain constraint that limits values to integers between 18 and 25.



## 6 Relational Algebra

Relational algebra is a procedural query language that operates on relations (tables). It provides a set of operations for manipulating relations to retrieve desired data. The basic operations include:

### 6.1 Basic Operations in Relational Algebra

- **Selection ( $\sigma$ ):** Filters rows that satisfy a given predicate (condition). It is denoted by  $\sigma$  and works on a single relation.

**Example:**

$$\sigma_{major='CS'}(Student)$$

This selects all students whose major is Computer Science from the *Student* table.

- **Projection ( $\pi$ ):** Selects specific columns from a relation. It removes duplicates and keeps only the desired attributes.

**Example:**

$$\pi_{name, address}(Student)$$

This selects the 'name' and 'address' columns from the *Student* table.

- **Set Difference ( $-$ ):** Retrieves tuples that are in one relation but not in another.

**Example:**

$$\pi_{ssn}(Student) - \pi_{ssn}(Registered)$$

This returns the SSNs of students who have not registered for any courses.

- **Cartesian Product ( $\times$ ):** Combines each row of the first relation with every row of the second relation.

**Example:**

$$Student \times Course$$

This gives all possible combinations of students and courses, resulting in a large relation with all attributes of both relations.

- **Rename ( $\rho$ ):** Changes the name of a relation or its attributes.

**Example:**

$$\rho_{S(A,B,C,D)}(Student)$$

This renames the *Student* table to *S* and its attributes to *A*, *B*, *C*, and *D*.

### 6.2 Example Relations

Consider the following relations:

<b>Student</b> ( <i>ssn</i> , <i>name</i> , <i>address</i> , <i>major</i> )				
ssn	name	address	major	
101	Ram	Varanasi	CS	
102	Shyam	Delhi	IT	

<b>Course</b> ( <i>code</i> , <i>title</i> )	
code	title
CS320	Database Systems
CS150	Operating Systems

<b>Registered</b> ( <i>ssn, code</i> )	<b>ssn</b>	<b>code</b>
	101	CS320
	102	CS150

### 6.3 Relational Algebra Queries

- **a)** Select the names of students enrolled in course CS320:

$$\pi_{\text{name}}(\sigma_{\text{code} = \text{'CS320'}}(\text{Registered} \bowtie \text{Student}))$$

- **b)** Find which subject Ram is taking:

$$\pi_{\text{title}}(\sigma_{\text{name} = \text{'Ram'}}(\text{Registered} \bowtie \text{Student} \bowtie \text{Course}))$$

- **c)** Find who teaches CS150:

$$\pi_{\text{lecturer}}(\sigma_{\text{code} = \text{'CS150'}}(\text{Subject}))$$

- **d)** Find who teaches both CS150 and CS302:

$$\pi_{\text{lecturer}}(\sigma_{\text{code} = \text{'CS150'}}(\text{Subject})) \cap \pi_{\text{lecturer}}(\sigma_{\text{code} = \text{'CS302'}}(\text{Subject}))$$

- **e)** Find who teaches at least two different subjects:

$$\pi_{\text{lecturer}}(\text{Subject}) \cap \pi_{\text{lecturer}}(\text{Subject})$$

- **f)** Find the names of students in both CS150 and CS307:

$$\pi_{\text{name}}(\sigma_{\text{code} = \text{'CS150'}}(\text{Registered} \bowtie \text{Student})) \cap \pi_{\text{name}}(\sigma_{\text{code} = \text{'CS307'}}(\text{Registered} \bowtie \text{Student}))$$

- **g)** Find the names of students in both CS150 and CS1200:

$$\pi_{\text{name}}(\sigma_{\text{code} = \text{'CS150'}}(\text{Registered} \bowtie \text{Student})) \cap \pi_{\text{name}}(\sigma_{\text{code} = \text{'CS1200'}}(\text{Registered} \bowtie \text{Student}))$$

### 6.4 Additional Operations in Relational Algebra

- **Set Intersection ( $\cap$ ):** Retrieves tuples that are common to both relations.

**Example:**

$$\pi_{\text{ssn}}(\text{Registered}) \cap \pi_{\text{ssn}}(\text{Graduated})$$

This finds students who are both registered and have graduated.

- **Natural Join ( $\bowtie$ ):** Combines two relations by matching columns that have the same name in both tables.

**Example:**

$$\text{Student} \bowtie \text{Registered}$$

This will join the two relations on the common attribute *ssn*.

- **Division ( $\div$ ):** Used to find tuples in one relation that are related to all tuples in another relation.

**Example:**

$$R(A, B) \div S(B)$$

This returns all values of A for which every B in S exists in R.

- **Assignment ( $\leftarrow$ ):** Stores the result of a query into a temporary relation.

**Example:**

$$T \leftarrow \pi_{ssn}(\sigma_{major = 'CS'}(Student))$$

This stores all CS students in the temporary relation  $T$ .

## 7 Relational Calculus

Relational calculus is a non-procedural query language. Instead of specifying how to retrieve data, you specify what data to retrieve. There are two types: Tuple Relational Calculus (TRC) and Domain Relational Calculus (DRC).

### 7.1 Characteristics of Relational Calculus

- **Declarative Nature:** Relational calculus focuses on describing the desired result rather than the process to achieve it. It specifies what to retrieve but not how to retrieve it.
- **Non-Procedural:** Unlike procedural languages, relational calculus does not involve specifying the steps to execute queries. Instead, it uses logical expressions to define the data requirements.
- **Logical Formulas:** Queries in relational calculus are expressed using logical formulas involving variables and predicates. The result is a set of tuples or values that satisfy the formula.
- **Tuple and Domain Variables:** There are two types of variables in relational calculus:
  - **Tuple Variables:** Represent entire tuples from relations (used in Tuple Relational Calculus).
  - **Domain Variables:** Represent individual attribute values from the domains of relations (used in Domain Relational Calculus).
- **Logical Connectives:** Uses logical connectives such as AND ( $\wedge$ ), OR ( $\vee$ ), and NOT ( $\neg$ ) to combine conditions.
- **Quantifiers:** Uses quantifiers such as EXISTS ( $\exists$ ) and FOR ALL ( $\forall$ ) to specify the presence or absence of tuples or values that meet the criteria.
- **Query Result:** The result of a relational calculus query is a set of tuples or values that meet the specified conditions.

## 7.2 Tuple Relational Calculus (TRC)

In Tuple Relational Calculus (TRC), variables represent tuples from relations. Queries are expressed as logical formulas where variables are tuples and the formula defines the condition that tuples must satisfy.

### Characteristics of TRC:

- Uses tuple variables, which represent entire tuples in the database.
- Queries are expressed in the form of logical formulas involving tuple variables.
- The result of a TRC query is a set of tuples that satisfy the given formula.

### General Form:

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

Where  $T$  is a tuple variable, and  $P(T)$  is a predicate or condition that must be true for  $T$ .

### Example Queries in TRC:

- **Find the names of students enrolled in course CS320:**

$$\{S.name \mid \exists R(Registered(R) \wedge R.ssn = S.ssn \wedge R.code = 'CS320')\}$$

This query retrieves the names of students for whom there exists a registration tuple where the student's SSN matches and the course code is CS320.

- **Find the names of students who are not enrolled in any course:**

$$\{S.name \mid \forall R(Registered(R) \rightarrow R.ssn \neq S.ssn)\}$$

This query retrieves the names of students who do not have any corresponding entries in the 'Registered' relation.

- **Find the SSNs of students who are enrolled in all courses:**

$$\{S.ssn \mid \forall C(Course(C) \rightarrow \exists R(Registered(R) \wedge R.ssn = S.ssn \wedge R.code = C.code))\}$$

This query finds SSNs of students who are registered for every course listed in the 'Course' relation.

## 7.3 Domain Relational Calculus (DRC)

In Domain Relational Calculus (DRC), variables represent values from the domains of attributes. Queries are expressed based on these domain values, using logical formulas to define the conditions that must be met.

### Characteristics of DRC:

- Uses domain variables that represent individual attribute values.
- Queries are expressed as logical formulas involving domain variables.
- The result of a DRC query is a set of values that satisfy the given formula.

### General Form:

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

Where  $v$  is a domain variable and  $P(v)$  is a predicate that must be true for  $v$ .

### Example Queries in DRC:

- **Find the names of students enrolled in course CS320:**

$\{name \mid \exists ssn, code (Student(ssn, name, address, major) \wedge Registered(ssn, code) \wedge code = 'CS320')\}$

This query retrieves all names where there exists a student with a matching SSN and a corresponding registration for course CS320.

- **Find the SSNs of students who are enrolled in at least one course:**

$\{ssn \mid \exists code (Registered(ssn, code))\}$

This query retrieves SSNs of students who have at least one entry in the 'Registered' relation.

- **Find the courses taught by a specific lecturer:**

$\{code \mid \exists title (Subject(code, title) \wedge lecturer = 'Hector')\}$

This query finds all course codes where the lecturer is Hector.

## SQL Introduction

SQL (Structured Query Language) is a standardized programming language used to manage and manipulate relational databases. It is the primary tool for interacting with data stored in relational database management systems (RDBMS).

### Definition of SQL:

- SQL is used to create, read, update, and delete data from a database.
- It allows users to define the structure of data and establish relationships between data sets.
- SQL is a declarative language, meaning users specify *what* they want, not *how* to get it.
- It follows the ANSI (American National Standards Institute) and ISO (International Organization for Standardization) standards.

**Major Categories of SQL Commands:** SQL commands can be broadly classified into two major categories:

- **Data Definition Language (DDL):** These commands are used to define and modify the structure of a database. Examples include:
  - CREATE: To create new databases, tables, or other objects.
  - ALTER: To modify an existing database structure.

- DROP: To delete databases, tables, or other objects.
- TRUNCATE: To remove all records from a table, but keep its structure.
- **Data Manipulation Language (DML):** These commands are used to manipulate the data within a database. Examples include:
  - SELECT: To query and retrieve data from the database.
  - INSERT: To insert new data into a table.
  - UPDATE: To modify existing data within a table.
  - DELETE: To remove data from a table.

### **Characteristics of SQL:**

- SQL is a declarative language, so it focuses on the *what* rather than the *how*.
- It supports both Data Definition Language (DDL) and Data Manipulation Language (DML) operations.
- SQL can handle large amounts of data efficiently.
- SQL is versatile and can be used with different database management systems like MySQL, PostgreSQL, Oracle, SQL Server, etc.

### **Advantages of SQL:**

- **Simplicity:** SQL commands are easy to learn and use, even for beginners.
- **Efficient Data Management:** SQL is designed to handle large databases and execute complex queries quickly.
- **Multiple Database Support:** SQL is compatible with most relational database systems.
- **Standardization:** SQL follows standardized syntax and rules, ensuring portability across different systems.

### **Disadvantages of SQL:**

- **Complexity in Advanced Queries:** While simple queries are easy, more complex queries (e.g., involving multiple joins or subqueries) can be difficult to construct.
- **Limited Control:** SQL abstracts the process of retrieving and manipulating data, giving users limited control over the execution of the queries.
- **System Dependence:** While SQL is standardized, some commands and features can vary across different database systems.
- **Overhead:** SQL operations, especially on very large datasets, can introduce overhead and may require optimization techniques.

## SQL Data Types and Literals

SQL supports various data types that allow users to define the kind of data that can be stored in a database table. These data types ensure data integrity and optimize storage.

### SQL supports the following data types:

- **CHAR(n)**: Fixed-length character strings, where n represents the number of characters.
- **VARCHAR(n)**: Variable-length character strings, where n represents the maximum number of characters.
- **INT**: Integer numbers used to store whole numbers.
- **SMALLINT**: A smaller range of integer numbers, typically requiring less storage than INT.
- **NUMERIC(p, d)**: Stores fixed-point numbers where p represents the precision (total number of digits) and d represents the scale (digits after the decimal point).
- **REAL or DOUBLE PRECISION**: Used for storing approximate numeric values with floating-point precision.
- **FLOAT(n)**: A floating-point number where n defines the precision in terms of the number of digits.
- **DATE**: Used to store dates in the format YYYY-MM-DD.
- **TIME**: Used to store time values in the format HH:MM:SS.

## SQL Literals

SQL literals are constant values that are used in SQL queries for comparison, insertion, and manipulation of data. They represent fixed values within the SQL statement.

### Types of Literals:

- **Character String Literals**: These are sequences of characters enclosed within single quotes. Example: 'Hello World'.
- **Bit String Literals**: These represent binary data (bits) and are usually prefixed with a B. Example: B'10101'.
- **Exact Numeric Literals**: Represent numbers without fractional parts. These can be integers or decimals. Example: 42, 123.45.
- **Approximate Numeric Literals**: Represent floating-point numbers that are approximations rather than exact values. Example: 1.23E4 (which represents  $1.23 \times 10^4$ ).

# SQL Commands:

SQL commands are broadly classified into different categories based on their functionality:

- **DDL (Data Definition Language):** These commands are used to define or alter the structure of database objects.
  - Examples: CREATE, ALTER, DROP
- **DML (Data Manipulation Language):** These commands are used to manipulate data within the database.
  - Examples: SELECT, INSERT, UPDATE, DELETE

## Common SQL Commands with Syntax:

- **INSERT:** Inserts new data into a table.

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

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

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

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

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

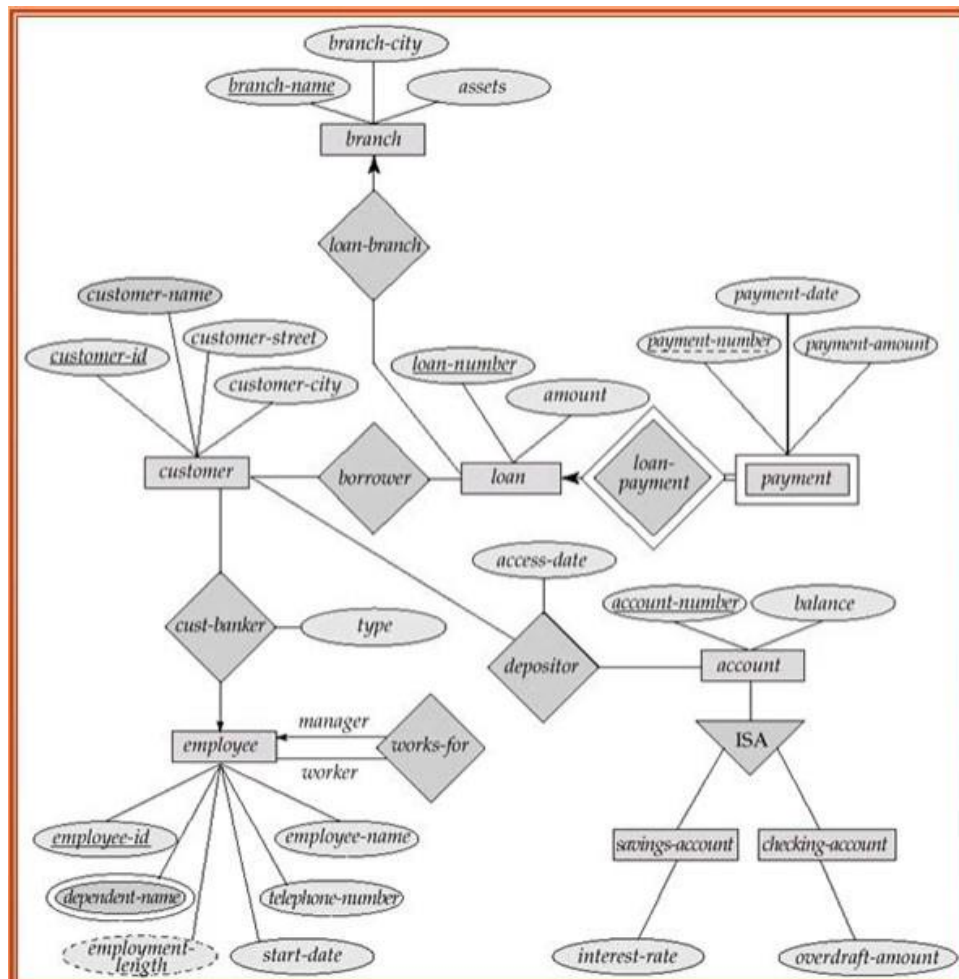
- **SELECT:** Retrieves data from one or more tables.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- **ALTER TABLE:** Modifies the structure of an existing table (e.g., adding a column).

```
ALTER TABLE table_name
ADD column_name datatype;
```





## SQL Operators and Procedures

SQL supports various operators such as arithmetic, comparison, and logical operators.

**Example:**

```
SELECT * FROM Students WHERE Age > 20;
```

## Different Types of SQL Operators

### Arithmetic Operators

- + (Addition)
- - (Subtraction)
- \* (Multiplication)
- / (Division)
- % (Modulo)

## Unary Operators

- + (Positive)
- - (Negative)

## Binary Operators

- + (Addition)
- - (Subtraction)
- \* (Multiplication)
- / (Division)

## Comparison Operators

Operator	Description
=	Equal to
!=	Not equal to
>	Greater than
<	Less than
<=	Less than or equal to
>=	Greater than or equal to

## Logical Operators

- AND
- OR
- NOT

## Set Operators

- UNION
- INTERSECT
- MINUS

## Operator Precedence

- Parentheses ()
- Unary operators +, -
- Arithmetic operators \*, /, %
- Addition and Subtraction +, -
- Comparison operators =, !=, >, <, >=, <=

- Logical operators AND, OR, NOT

article amsmath booktabs

## Relational Algebra Operations Supported in SQL

### Selection

Selection () is used to filter rows based on a specified condition. It is analogous to the SQL WHERE clause.

**Example:**

```
SELECT * FROM Employees WHERE Age > 30;
```

<b>Table: Employees</b>	EmployeeID	Name	Age
	1	Radha	28
	2	Shyam	35
	3	Ram	40

<b>Result:</b>	EmployeeID	Name	Age
	2	Shyam	35
	3	Ram	40

### Projection

Projection () is used to select specific columns from a table. It is similar to selecting columns in SQL.

**Example:**

```
SELECT Name, Age FROM Employees;
```

<b>Table: Employees</b>	EmployeeID	Name	Age
	1	Radha	28
	2	Shyam	35
	3	Ram	40

<b>Result:</b>	Name	Age
	Radha	28
	Shyam	35
	Ram	40

### Set Difference

Set Difference () finds rows present in one table but not in another. It is equivalent to SQL's EXCEPT clause.

**Example:**

```
SELECT Name FROM Employees WHERE Age < 35 EXCEPT SELECT Name FROM Employees WHERE Name = 'Ram';
```

**Table: Employees**

EmployeeID	Name	Age
1	Radha	28
2	Shyam	35
3	Ram	40

**Result:**

Name
Radha

## Cartesian Product

Cartesian Product (×) combines each row from one table with each row from another table. It is similar to a SQL join without a condition.

**Example:**

SELECT \* FROM Employees, Departments;

**Table: Employees**

EmployeeID	Name	Age
1	Radha	28
2	Shyam	35
3	Ram	40

**Table: Departments**

DeptID	DeptName
1	HR
2	IT

**Result:**

EmployeeID	Name	Age	DeptID	DeptName
1	Radha	28	1	HR
1	Radha	28	2	IT
2	Shyam	35	1	HR
2	Shyam	35	2	IT
3	Ram	40	1	HR
3	Ram	40	2	IT

## Rename

Rename () is used to rename a table or columns. It is used in SQL with the AS keyword.

**Example:**

SELECT Name AS EmployeeName FROM Employees;

**Table: Employees**

EmployeeID	Name	Age
1	Radha	28
2	Shyam	35
3	Ram	40

**Result:**

EmployeeName
Radha
Shyam
Ram

## Tables, Views, and Indexes

### Tables

Tables are the fundamental building blocks of a relational database. They store data in a structured format consisting of rows and columns. Each row represents a record, and each column represents an attribute of the record.

#### Example:

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

**Table: Students**

StudentID	Name	Age
1	Radha	20
2	Shyam	22
3	Ram	23

### Views

A view is a virtual table based on the result of a SQL query. It does not store data itself but provides a way to simplify complex queries or present data in a particular format.

#### Syntax:

```
CREATE VIEW ViewName AS SELECT Column1, Column2 FROM TableName WHERE Condition;
```

#### Example:

```
CREATE VIEW StudentView AS SELECT Name, Age FROM Students WHERE Age > 21;
```

**Table: Students**

StudentID	Name	Age
1	Radha	20
2	Shyam	22
3	Ram	23

**Resulting View: StudentView**

Name	Age
Shyam	22
Ram	23

### Indexes

Indexes are used to improve the speed of data retrieval operations on a database table. They create an internal structure that allows the database to find data quickly without scanning the entire table.

#### Syntax:

```
CREATE INDEX IndexName ON TableName (ColumnName);
```

#### Example:

```
CREATE INDEX idx_student_name ON Students(Name);
```

## Subqueries and Joins

Subqueries can be used to retrieve data from one table based on the results of a query from another table. Here, we will use subqueries to work with two tables: Students and Marks.

**Tables:**

<b>Students</b>	StudentID	Name	Age
	1	Radha	20
	2	Shyam	22
	3	Ram	23

<b>Marks</b>	StudentID	Subject	Marks
	1	Math	85
	1	Science	90
	2	Math	78
	3	Science	88

**Syntax for Subquery with Join:**

```
SELECT Name, Marks FROM Students JOIN Marks ON Students.StudentID  
= Marks.StudentID WHERE Marks.Subject = 'Math';
```

**Resulting Data:**

Name	Marks
Radha	85
Shyam	78

article amsmath booktabs enumitem graphicx

SQL Operations and Database Management Your Name September 17, 2024

## Tables, Views, and Indexes

### Tables

Tables store data in rows and columns. Each table has a unique name and consists of columns (fields) and rows (records).

**Syntax to Create a Table:**

```
CREATE TABLE TableName (Column1 DataType, Column2 DataType, ...);
```

**Example:**

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

### Views

A view is a virtual table derived from one or more tables. It does not store data itself but provides a way to simplify complex queries.

**Syntax to Create a View:**

```
CREATE VIEW ViewName AS SELECT Column1, Column2 FROM TableName WHERE  
Condition;
```

**Example:**

```
CREATE VIEW StudentAges AS SELECT Name, Age FROM Students WHERE Age  
> 21;
```

## Indexes

Indexes improve the speed of data retrieval operations on a table. They are created on columns that are frequently used in queries.

### Syntax to Create an Index:

```
CREATE INDEX IndexName ON TableName (ColumnName);
```

### Example:

```
CREATE INDEX idx_age ON Students(Age);
```

## Aggregate Functions

**Aggregate Functions** perform calculations on a set of values and return a single result.

- **SUM():** Calculates the total sum of a numeric column.
- **AVG():** Calculates the average value of a numeric column.

### Syntax for Aggregate Functions:

```
SELECT AGG_FUNC(ColumnName) FROM TableName;
```

### Example:

```
SELECT SUM(Marks) AS TotalMarks FROM Marks;
```

### Table: Marks

StudentID	Subject	Marks
1	Math	85
1	Science	90
2	Math	78
3	Science	88

### Output:

TotalMarks
341

## Relational Operations in SQL

### Eliminating Duplicates

**Purpose:** Remove duplicate rows from the result set.

### Syntax:

```
SELECT DISTINCT Column1, Column2 FROM TableName;
```

### Example:

```
SELECT DISTINCT Name FROM Students;
```

**Table: Students**

StudentID	Name	Age
1	Radha	20
2	Shyam	22
2	Shyam	22
3	Ram	23

**Output:**

Name
Radha
Shyam
Ram

## Union

**Purpose:** Combine results from two or more SELECT statements, removing duplicates.

**Syntax:**

```
SELECT Column1 FROM Table1 UNION SELECT Column1 FROM Table2;
```

**Example:**

```
SELECT Name FROM Students UNION SELECT Name FROM Teachers;
```

**Table: Students**

StudentID	Name	Age
1	Radha	20
2	Shyam	22

**Table: Teachers**

TeacherID	Name	Subject
1	Priya	Math
2	Raj	Science

**Output:**

Name
Radha
Shyam
Priya
Raj

## Grouping

**Purpose:** Group rows that have the same values into summary rows.

**Syntax:**

```
SELECT Column1, COUNT(*) FROM TableName GROUP BY Column1;
```

**Example:**



SELECT Subject, COUNT(\*) AS NumberOfStudents FROM Marks GROUP BY Subject;

**Table: Marks**

StudentID	Subject	Marks
1	Math	85
1	Science	90
2	Math	78
3	Science	88

**Output:**

Subject	NumberOfStudents
Math	2
Science	2

## Group By Clause

**Purpose:** Used with aggregate functions to group the result set by one or more columns.

**Syntax:**

SELECT Column1, AGG EUNC(Column2) FROM TableName GROUP BY Column1;

**Example:**

SELECT Subject, AVG(Marks) AS AverageMarks FROM Marks GROUP BY Subject;

**Output:**

Subject	AverageMarks
Math	81.5
Science	89

## Difference between WHERE and HAVING Clause:

Criteria	WHERE	HAVING
<b>Purpose</b>	Filters rows before grouping	Filters groups after grouping
<b>Usage</b>	Used with SELECT, UPDATE, DELETE	Used with SELECT and GROUP BY
<b>Syntax</b>	SELECT * FROM TableName WHERE Condition;	SELECT Column1, AGG FUNC(Column2) FROM TableName GROUP BY Column1 HAVING Condition;

### Example of WHERE Clause:

SELECT \* FROM Students WHERE Age > 21;

### Example of HAVING Clause:

SELECT Subject, AVG(Marks) AS AverageMarks FROM Marks GROUP BY Subject  
HAVING AVG(Marks) > 80;

# Database Modifications

## Commands to Modify Database:

- INSERT INTO TableName VALUES (...);
- UPDATE TableName SET Column = Value WHERE Condition;
- DELETE FROM TableName WHERE Condition;

### Example: Inserting Data

```
INSERT INTO Students (StudentID, Name, Age) VALUES (4, 'Sita', 24);
```

article amsmath booktabs enumitem graphicx

## Joins, Unions, Intersection, Minus

- **Joins:** Combine rows from two or more tables based on a related column.
- **Unions:** Combine result sets from multiple queries, eliminating duplicates.
- **Intersection:** Retrieves common rows present in both queries.
- **Minus:** Returns rows present in the first query but not in the second.

## Types of Join Operations

### Inner Join

**Purpose:** Retrieve rows with matching values in both tables.

### Syntax:

```
SELECT * FROM Table1 INNER JOIN Table2 ON Table1.Column = Table2.Column;
```

### Example Tables:

EmployeeID	Name
1	Alice
2	Bob
3	Carol

Table 1: Employee Table

EmployeeID	Department
1	HR
2	IT
4	Finance

Table 2: Department Table

### Example Query:

```
SELECT Employees.Name, Departments.Department FROM Employees INNER  
JOIN Departments ON Employees.EmployeeID = Departments.EmployeeID;
```

**Output:**

Name	Department
Alice	HR
Bob	IT

Table 3: Inner Join Result

**Outer Join**

**Purpose:** Retrieve all rows from one table and matched rows from another table.

**Left Outer Join Syntax:**

```
SELECT * FROM Table1 LEFT OUTER JOIN Table2 ON Table1.Column = Table2.Column;
```

**Right Outer Join Syntax:**

```
SELECT * FROM Table1 RIGHT OUTER JOIN Table2 ON Table1.Column = Table2.Column;
```

**Example Query:**

```
SELECT Employees.Name, Departments.Department FROM Employees LEFT  
OUTER JOIN Departments ON Employees.EmployeeID = Departments.EmployeeID;
```

**Output:**

Name	Department
Alice	HR
Bob	IT
Carol	NULL

Table 4: Left Outer Join Result

article amsmath booktabs enumitem graphicx  
SQL Operations and Management Your Name September 17, 2024

## Joins, Unions, Intersection, Minus

- **Joins:** Combine rows from two or more tables based on a related column.
- **Unions:** Combine result sets from multiple queries, eliminating duplicates.
- **Intersection:** Retrieves common rows present in both queries.
- **Minus:** Returns rows present in the first query but not in the second.

## Types of Join Operations

### Inner Join

**Purpose:** Retrieve rows with matching values in both tables.

**Syntax:**

```
SELECT * FROM Table1 INNER JOIN Table2 ON Table1.Column = Table2.Column;
```

#### Example Tables:

EmployeeID	Name
1	Alice
2	Bob
3	Carol

Table 5: Employee Table

EmployeeID	Department
1	HR
2	IT
4	Finance

Table 6: Department Table

#### Example Query:

```
SELECT Employees.Name, Departments.Department FROM Employees INNER JOIN Departments ON Employees.EmployeeID = Departments.EmployeeID;
```

**Output:**

Name	Department
Alice	HR
Bob	IT

Table 7: Inner Join Result

### Outer Join

**Purpose:** Retrieve all rows from one table and matched rows from another table.

**Left Outer Join Syntax:**

```
SELECT * FROM Table1 LEFT OUTER JOIN Table2 ON Table1.Column = Table2.Column;
```

**Right Outer Join Syntax:**

```
SELECT * FROM Table1 RIGHT OUTER JOIN Table2 ON Table1.Column = Table2.Column;
```

#### Example Query:

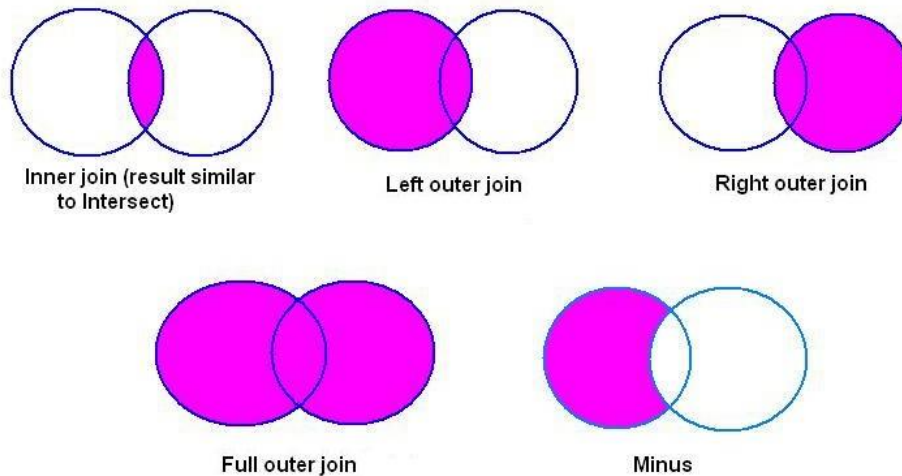
```
SELECT Employees.Name, Departments.Department FROM Employees LEFT OUTER JOIN Departments ON Employees.EmployeeID = Departments.EmployeeID;
```

**Output:**

Name	Department
Alice	HR
Bob	IT
Carol	NULL

Table 8: Left Outer Join Result

#### JOINS AND SET OPERATIONS IN RELATIONAL DATABASES



## SQL Set Operations

### Union

**Purpose:** Combines results from two or more queries, removing duplicates.

#### Syntax:

```
SELECT Column1 FROM Table1 UNION SELECT Column1 FROM Table2;
```

#### Example:

```
SELECT Name FROM Employees UNION SELECT Name FROM Contractors;
```

**Intersection Purpose:** Retrieves common rows from two queries.

#### Syntax:

```
SELECT Column1 FROM Table1 INTERSECT SELECT Column1 FROM Table2;
```

#### Example:

```
SELECT Name FROM Employees INTERSECT SELECT Name FROM Contractors;
```

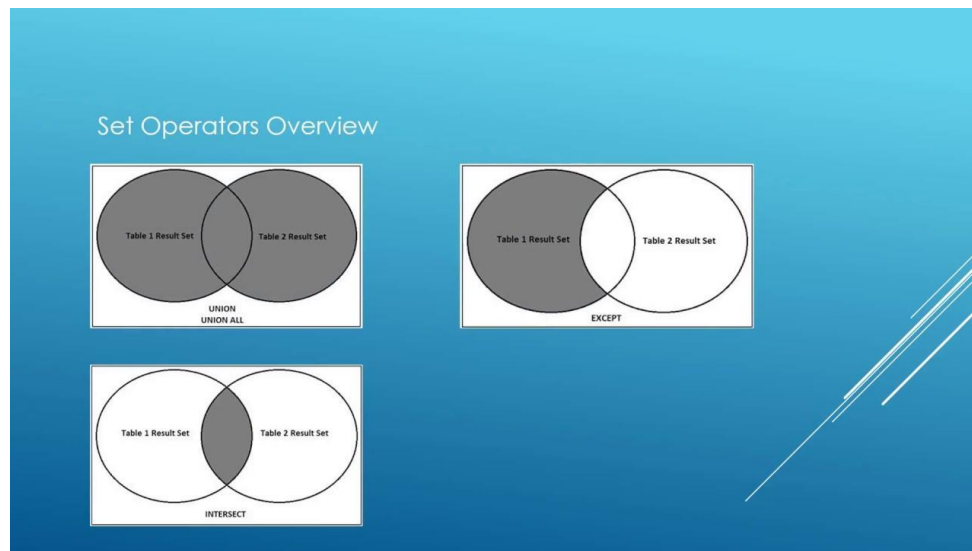
**Minus Purpose:** Returns rows present in the first query but not in the second.

#### Syntax:

```
SELECT Column1 FROM Table1 MINUS SELECT Column1 FROM Table2;
```

#### Example:

```
SELECT Name FROM Employees MINUS SELECT Name FROM Contractors;
```



## Cursors, Triggers, and Procedures in SQL/PL-SQL

### Cursors

**Purpose:** Cursors provide a mechanism to handle a set of rows returned by a query. They are useful for processing each row individually in a set of rows.

#### Syntax:

```
DECLARE cursor_name CURSOR FOR SELECT statement;
```

#### Example:

```
DECLARE emp_cursor CURSOR FOR SELECT Name FROM Employees;
```

#### Steps to Use a Cursor:

1. **Declare Cursor:** Define the cursor with the query.
2. **Open Cursor:** Execute the query and populate the cursor.
3. **Fetch Data:** Retrieve data from the cursor row by row.
4. **Close Cursor:** Release the cursor resources.

#### Example Procedure with Cursor:

```
DECLARE
emp_cursor CURSOR FOR SELECT Name FROM Employees;
BEGIN
OPEN emp_cursor;
LOOP
FETCH emp_cursor INTO emp_name;
EXIT WHEN emp_cursor%NOTFOUND;
-- Process emp_name
END LOOP;
CLOSE emp_cursor;
END;
```

## Triggers

**Purpose:** Triggers are special types of stored procedures that automatically execute SQL code in response to certain events on a table.

**Data Manipulation Triggers:** These triggers execute in response to 'INSERT', 'UPDATE', or 'DELETE' operations on a table.

### Types of Data Manipulation Triggers:

- **BEFORE Trigger:** Executes before the data modification.
- **AFTER Trigger:** Executes after the data modification.
- **INSTEAD OF Trigger:** Executes in place of the data modification.

### Syntax for Data Manipulation Trigger:

```
CREATE TRIGGER trigger_name
BEFORE/AFTER/INSTEAD OF INSERT/UPDATE/DELETE ON TableName
FOR EACH ROW
BEGIN
-- SQL statements
END;
```

### Example of AFTER Trigger:

```
CREATE TRIGGER after_insert_emp
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
INSERT INTO AuditLog (Action, Date) VALUES ('INSERT', SYSDATE);
END;
```

**DDL Triggers:** These triggers respond to Data Definition Language events such as 'CREATE', 'ALTER', and 'DROP'.

### Syntax for DDL Trigger:

```
CREATE OR REPLACE TRIGGER trigger_name
AFTER/BEFORE EVENT ON SCHEMA
BEGIN
-- SQL statements
END;
```

### Example of DDL Trigger:

```
CREATE OR REPLACE TRIGGER log_ddl_events
AFTER CREATE ON SCHEMA
BEGIN
INSERT INTO DDLLog (Event, Date) VALUES ('CREATE TABLE', SYSDATE);
END;
```

**Logon Trigger:** These triggers execute when a user connects to the database.

### Syntax for Logon Trigger:

```
CREATE OR REPLACE TRIGGER logon trigger
AFTER LOGON ON DATABASE
BEGIN
-- SQL statements
END;
```

**CLR (Common Language Runtime) Trigger:** Used for managing SQL Server triggers written in .NET languages.

## Procedures

**Purpose:** Procedures are reusable SQL code blocks that can be executed with a single call. They encapsulate logic and can perform operations like data manipulation and complex calculations.

### Syntax:

```
CREATE PROCEDURE procedure name
AS
BEGIN
-- SQL statements
END;
```

### Example Procedure:

```
CREATE PROCEDURE IncreaseSalary (emp.id INT, amount DECIMAL)
AS
BEGIN
UPDATE Employees SET Salary = Salary + amount WHERE EmployeeID =
emp id;
END;
```

## Embedded SQL

**Purpose:** Embedded SQL allows the integration of SQL statements within a programming language. It provides a way to interact with a database from within a host language.

### Syntax:

```
EXEC SQL SQL_STATEMENT;
```

### Example in C:

```
EXEC SQL SELECT Name INTO :name FROM Employees WHERE EmployeeID =
:empID;
```

## Dynamic SQL

**Purpose:** Dynamic SQL allows the execution of SQL statements that are constructed at runtime. It is useful for scenarios where SQL statements are not known until execution time.

### Syntax:



```
EXECUTE IMMEDIATE 'SQL_STATEMENT';
```

**Example:**

```
EXECUTE IMMEDIATE 'INSERT INTO Employees (Name, Salary) VALUES (:name, :salary)';
```

## 8 Procedures in PL/SQL

**Definition:** A procedure in PL/SQL is a named block of code that performs a specific task and can be executed whenever needed. It is a reusable piece of logic that can be invoked using a call statement. Unlike functions, procedures do not need to return a value, although they can return data using output parameters.

**Syntax:**

```
CREATE OR REPLACE PROCEDURE procedure_name IS
BEGIN
    -- SQL and PL/SQL statements
END;
```

**Example:**

```
CREATE OR REPLACE PROCEDURE IncreaseSalary (emp_id NUMBER, amount NUMBER) IS
BEGIN
    UPDATE Employees SET Salary = Salary + amount WHERE EmployeeID = emp_id;
END;
```

**Advantages of Using Procedures:**

- **Modularity:** Procedures allow for code to be divided into smaller, manageable pieces, promoting code reuse and better organization.
- **Reusability:** Procedures can be called multiple times from different parts of an application, reducing code duplication.
- **Performance:** Since procedures are stored in the database, they can be compiled once and executed many times, resulting in faster execution.
- **Security:** Procedures can control user access by limiting the scope of database operations they can perform.

**Disadvantages of Using Procedures:**

- **Complex Debugging:** Debugging procedures can be more complex than debugging regular code, especially when procedures call other procedures or involve dynamic SQL.
- **Maintenance:** If the procedure logic is not well-documented, maintaining large procedures can become difficult.
- **Limited Error Handling:** Error handling within procedures needs to be carefully managed, as unhandled exceptions can terminate the procedure unexpectedly.



---

## Constraints on Tuples in Functional Dependencies

Let  $R$  be a relation and consider two tuples  $t_1$  and  $t_2$  in  $R$ . If a functional dependency  $\alpha \rightarrow \gamma$  holds, then:

- If  $t_1[\alpha] = t_2[\alpha]$ , then it must be true that  $t_1[\gamma] = t_2[\gamma]$ .

we will compute the canonical cover step by step:

### Step 1: Decompose Functional Dependencies

In this case, the dependencies  $A \rightarrow B$  and  $B \rightarrow C$  are already decomposed, so no further decomposition is needed.

### Step 2: Remove Redundant Dependencies

We now check for any redundant dependencies. Here, we can derive  $A \rightarrow C$  using the transitivity rule:

$$A \rightarrow B \text{ and } B \rightarrow C \Rightarrow A \rightarrow C$$

Thus, the canonical cover is:

2.  $E \rightarrow H$  is already given in  $F$ , and using the *union rule*, we get:

$$E \rightarrow AH$$

This matches with one of the functional dependencies in  $G$ .

3. From  $A \rightarrow C$  and  $AC \rightarrow D$ , using the *pseudotransitivity rule*, we get:

- The right-hand side of each dependency has only a single attribute.
- There are no extraneous attributes on the left-hand side of any dependency.
- There are no redundant dependencies.

---

ASAP

---

ASAP

---

**Problem: Convert Functional Dependency Set into Minimal Cover**

Given a relation  $R(A, B, C)$  and a functional dependency set  $F = \{A \rightarrow B, B \rightarrow C, AB \rightarrow B, AB \rightarrow C, AC \rightarrow B\}$ , convert this into a minimal cover step by step:

**Step 1: Simplify Right-Hand Side**

We break down the functional dependencies with multiple attributes on the right-hand side:

ASAP

--	--	--	--

---

EmpID	EmpName	PhoneNumber
1	John	1234567890
1	John	9876543210

ASAP

---

## Second Normal Form (2NF)

A relation is in 2NF if it is in 1NF and every non-prime attribute is fully functionally dependent on the primary key.

**Example:**

Consider a relation with attributes (EmpID, ProjectID, HoursWorked). If *EmpID*      *ProjectID*

ASAP

possibility by ensuring that all functional dependencies have a superkey on the left-hand side.

**Example:**

Consider a relation with the following functional dependencies:

FD1: *StudentID*  $\rightarrow$  *Course*



---

FD2:  $Course \rightarrow Instructor$

Here,  $StudentID$  is a superkey, but  $Course$  is not. Since  $Course$  is determining  $Instructor$ , this violates BCNF because  $Course$  is not a superkey. To achieve BCNF, we split the relation into two tables:

Table 1:  $(StudentID, Course)$

ASAP

$$B \rightarrow F \Rightarrow AB = \{A, B, C, D, E, F\}$$

$$F \rightarrow GH \Rightarrow AB^+ = \{A, B, C, D, E, F, G, H\}$$

$$D \rightarrow IJ \Rightarrow AB^+ = \{A, B, C, D, E, F, G, H, I, J\}$$

Thus,  $AB^+ = \{A, B, C, D, E, F, G, H, I, J\} = R$ , so  $AB$  is a candidate key.

---

**Closure of A:**

$$A^+ = \{A\}$$

$$A \rightarrow DE \Rightarrow A^+ = \{A, D, E\}$$

$$D \rightarrow IJ \Rightarrow A^+ = \{A, D, E, I, J\}$$

ASAP

---

S.No.	3NF (Third Normal Form)	BCNF (Boyce-Codd Normal Form)
1.	A relation is in 3NF if it is in 2NF and no transitive de-	A relation is in BCNF if it is in 3NF and for every func-

- A relation is in **Boyce-Codd Normal Form (BCNF)** if, for every functional dependency  $X \rightarrow Y$ ,  $X$  is a superkey.

**Proof:**

Let us consider the conditions required by 3NF and BCNF:

- 
- In 3NF, a functional dependency  $X \rightarrow Y$  is allowed if  $Y$  is a prime attribute, even if  $X$  is not a superkey.
  - In BCNF, the only condition that is allowed for  $X \rightarrow Y$  is that  $X$  must be a superkey.
- Thus, BCNF is stricter because:

ASAP

$$X^+ = \{X, Z\}$$

Using  $Z \rightarrow W$ :

$$X^+ = \{X, Z, W\}$$

$X^+$  does not cover all attributes of  $R$ , so  $X$  is not a superkey.

---

### Closure of $Y$

Now, calculating the closure of  $Y$ :

$$Y^+ = \{Y\}$$

Using  $Y \rightarrow Z$ :

+

$$R_2 = \{X, Y\}$$

$$R_3 = \{Y, Q\}$$

$$R_4 = \{Z, W, Q\}$$

$$R_5 = \{X, Q\}$$

---

## Checking Pairs of Decomposed Relations

We check the pairwise intersection of the decomposed relations.

1.  $R_1(X, W)$  and  $R_2(X, Y)$ :

$$R_1 \cap R_2 = \{X\}$$

ASAP

---

## Multi-Valued Dependency (MVD)

A Multi-Valued Dependency occurs when one attribute determines a set of values for another attribute, independent of other attributes. In formal terms, if a relation  $R$  has a dependency  $X \twoheadrightarrow Y$ , then  $Y$  is multi-valued with respect to  $X$ , meaning for every value of  $X$ , there can

ASAP

FACULTY	COURSE
Prof. A	DBMS
Prof. A	DAA
Prof. B	OS
Prof. B	CN

---

FACULTY	COMMITTEE
Prof. A	Exam
Prof. A	Sports
Prof. B	Placement

ASAP



# DATABASE MANAGEMENT SYSTEM BCS501

## 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

## 1 Transaction Processing Concept

### 1.1 Transaction System

A **Transaction** is a sequence of operations performed as a single logical unit of work in a database management system (DBMS). For a transaction to be completed successfully, all its operations must be performed successfully. If any operation within the transaction fails, the entire transaction fails, and the system is restored to its previous consistent state. This property is known as **Atomicity**. Transactions are used to ensure data integrity and consistency in DBMS.

### 1.2 ACID Properties of Transaction

The **ACID** properties describe the key principles that guarantee that database transactions are processed reliably. These properties include:

- **Atomicity:** Ensures that all operations within a transaction are completed successfully, or none at all. If any part of the transaction fails, the entire transaction is aborted, and the database is restored to its previous state.
  - **Example:** Consider a bank transaction where \$500 is being transferred from Account A to Account B. This transaction involves two operations: (1) deducting \$500 from Account A and (2) adding \$500 to Account B. If the deduction from Account A succeeds but the addition to Account B fails, the entire transaction will be rolled back, and Account A will still have its \$500. Atomicity ensures that either both operations succeed or none at all.
- **Consistency:** Ensures that a transaction brings the database from one valid state to another, maintaining defined rules, constraints, and integrity.
  - **Example:** In a bank database, the rule is that the total balance across all accounts must remain the same after a transaction. If \$500 is transferred from Account A to Account B, the total balance before and after the transaction should remain the same. Consistency ensures that the sum of balances before and after the transaction is maintained correctly.
- **Isolation:** Ensures that the execution of a transaction is isolated from other transactions. This means that the intermediate states of a transaction are invisible to other transactions until the transaction is completed.
  - **Example:** Suppose two users are performing banking transactions simultaneously. One user transfers \$500 from Account A to Account B, while the other user transfers \$300 from Account C to Account D. Isolation ensures that the intermediate state of one transaction does not affect the other. Even though both transactions are running concurrently, the results will be as if the transactions were executed one after the other, without interference.
- **Durability:** Guarantees that once a transaction is committed, its results are permanent, even in the case of a system failure.
  - **Example:** After successfully transferring \$500 from Account A to Account B, the bank system commits the transaction. Even if the system crashes immediately after the commit, the changes (i.e., deduction from Account A and addition to Account B) will remain intact once the system is restored. Durability ensures that committed transactions are not lost.

### 1.3 How Does the Recovery Manager Ensure Atomicity of Transactions?

The recovery manager in a DBMS ensures **Atomicity** through the following techniques:

1. The **Write-Ahead Logging (WAL)** protocol is used to maintain a log of all database operations before they are actually applied to the database. This ensures that, in case of a failure, there is a record of the actions taken.
2. Before any changes are made to the database, the corresponding log entries are written to stable storage. This ensures that the system can trace the steps of the transaction and either complete it or roll it back if an error occurs.

3. In the event of a failure, the recovery manager checks the log to determine the status of the transaction. If the transaction was incomplete, it rolls back the changes using the log entries, ensuring that partial changes do not remain.

4. The system can use the log to perform a **redo** or **undo** operation. Redo ensures that committed changes are applied to the database, while undo ensures that changes from uncommitted transactions are reversed.

This process ensures that the transaction is either completed successfully or entirely rolled back, maintaining atomicity.

## 1.4 How Does the Recovery Manager Ensure Durability?

The recovery manager ensures **Durability** by writing committed transaction data to non-volatile storage. Once a transaction is committed, its changes are guaranteed to be saved, even if a system failure occurs. This is typically done using stable storage techniques such as checkpoints and maintaining transaction logs.

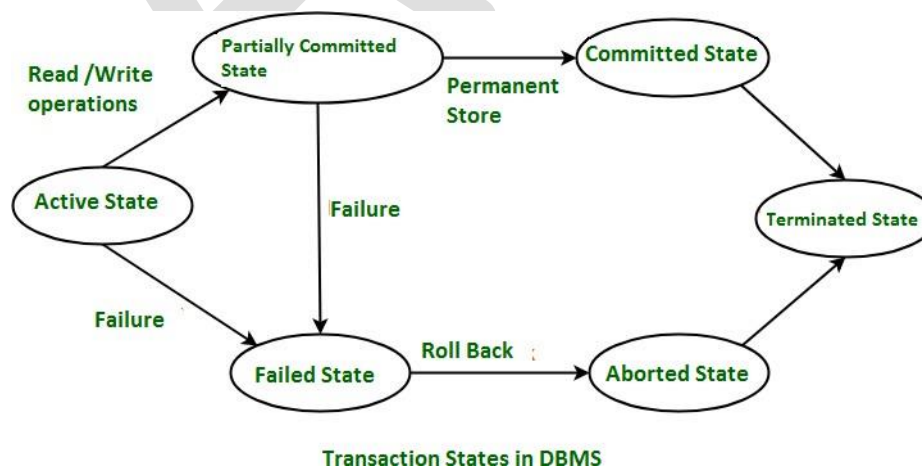
## 1.5 Usefulness of ACID Properties

The **ACID** properties are crucial to ensure the reliability and integrity of transactions in a DBMS. Without these properties:

- **Atomicity** ensures completeness and prevents partial updates.
- **Consistency** ensures that all transactions maintain the integrity of the database.
- **Isolation** ensures that transactions do not interfere with each other, providing a predictable environment for concurrent operations.
- **Durability** ensures that once a transaction is committed, it is not lost, even in the event of system failure.

## 1.6 State Diagram of Transaction

The state of a transaction can be represented in the following diagram:



## 1.7 States of Transaction

A transaction in a DBMS goes through several states during its execution. These states are:

- **Active:** This is the initial state of a transaction. The transaction remains in this state while its instructions are being executed. Any errors at this stage may cause the transaction to fail.
- **Partially Committed:** Once a transaction has executed its final operation but before it is committed, it enters the partially committed state. In this state, the changes made by the transaction are not yet permanent but are ready to be made permanent if no failure occurs.
- **Failed:** If a transaction encounters an error or failure during its execution, it moves to the failed state. In this state, the transaction cannot proceed further.
- **Aborted:** After a transaction has failed, the system moves it to the aborted state. In this state, the system must rollback the transaction, undoing any changes it made to the database, and restore the database to its previous consistent state.
- **Committed:** If a transaction completes successfully and all its operations are executed without error, it moves to the committed state. In this state, the changes made by the transaction are permanently saved to the database and are visible to other transactions.

## 1.8 How Can You Implement Atomicity in Transactions?

Atomicity in transactions can be implemented using several techniques that ensure the transaction is either fully completed or not executed at all. Key methods include:

### 1. **Completeness:**

- Ensures that all operations within a transaction are successfully completed.
- If any operation fails, the entire transaction is rolled back to maintain consistency.
- Partial execution is not allowed, meaning no changes will be saved unless the entire transaction succeeds.
- This is essential in systems where multiple steps are interconnected, such as transferring funds in banking transactions.

### 2. **Mutual Exclusion (Locking Mechanism):**

- Locking mechanisms are used to ensure mutual exclusion, where only one transaction can access and modify a data item at a time.
- By applying locks, a transaction ensures that no other transaction can make changes to the same data until it completes, maintaining atomicity.
- For example, a write-lock prevents any other transactions from reading or writing to the locked data until the lock is released.
- Common types of locks include exclusive locks for writes and shared locks for reads.

## 1.9 Serializability in Transactions

**Serializability** is a fundamental concept in ensuring the correctness of concurrently executing transactions. It ensures that the result of executing transactions concurrently is equivalent to some serial execution of the same transactions. This property is required to avoid concurrency issues such as lost updates, dirty reads, or accessing uncommitted data. In DBMS, serializability of a schedule is crucial for maintaining data integrity and consistency during concurrent transaction execution.

- A schedule is said to be **serializable** if the result of the concurrent execution of transactions is equivalent to a serial execution of the same transactions.
- **Conflict Serializability** occurs when a schedule can be transformed into a serial schedule by swapping non-conflicting operations. Two operations conflict if they access the same data item, and at least one of them is a write operation.
- **View Serializability** ensures that two schedules are equivalent if they produce the same result, even if their execution order is different. This form of serializability is more relaxed than conflict serializability.
- Serializability is critical to prevent problems like lost updates (where one transaction overwrites another's changes), dirty reads (where a transaction reads uncommitted data), and inconsistent data.

## 1.10 Conflict Serializability

**Conflict Serializability** checks whether a schedule can be transformed into a serial schedule by swapping non-conflicting operations. Two operations are said to be in conflict if they:

- Belong to different transactions.
- Access the same data item.
- At least one of them is a write operation.

For a schedule to be conflict serializable, it must be equivalent to some serial schedule by swapping non-conflicting operations. Below is an example of two transactions  $T_i$  and  $T_j$  with conflicting operations:

Step	Transaction $T_i$	Transaction $T_j$
1	$r(A)$	
2		$w(A)$
3	$w(B)$	
4		$r(B)$

Table 1: Conflict Serializability Example: Conflicting operations between  $T_i$  and  $T_j$

In this example, the operations  $r(A)$  in  $T_i$  and  $w(A)$  in  $T_j$  conflict because both access the same data item  $A$ , and one is a write operation. Similarly,  $w(B)$  in  $T_i$  and  $r(B)$  in  $T_j$  conflict for the same reason. These conflicts prevent the schedule from being conflict serializable without further modifications.

### 1.11 View Serializability

**View Serializability** is another way to check the equivalence of schedules. A schedule is view serializable if the following three conditions hold:

- **Initial Reads:** If a transaction reads a data item in a serial schedule, the same transaction must read the same data item in the non-serial schedule.
- **Final Writes:** If a transaction writes the final value of a data item in a serial schedule, the same transaction must perform the final write in the non-serial schedule.
- **Intermediate Writes:** Any data item written by one transaction and later read by another in a serial schedule must also have the same write-read relationship in the non-serial schedule.

Example of two schedules:

Step	Transaction $T_i$	Transaction $T_j$
1	$r(A)$	
2	$w(A)$	
3		$r(A)$
4		$w(B)$

Table 2: View Serializability Example:  $T_i$  and  $T_j$

In this example,  $T_i$  writes to  $A$  and  $T_j$  reads  $A$  after. As long as these read-write dependencies are maintained, the schedule can be view serializable.

### 1.12 Schedules and Types of Schedulers

A **schedule** is an arrangement of operations from different transactions. There are different types of schedules that control the recovery and consistency of transactions:

- **Recoverable Schedule:** A schedule is recoverable if, for any transaction  $T_j$  that reads a data item written by another transaction  $T_i$ ,  $T_j$  commits only after  $T_i$  commits. This prevents inconsistencies during rollbacks.
- **Cascadeless Schedule:** In a cascadeless schedule, a transaction can read a data item only if the transaction that wrote the data item has already committed. This avoids cascading rollbacks.
- **Strict Schedule:** A strict schedule ensures that a transaction can neither read nor write a data item until the previous transaction that wrote the data item has committed. This prevents dirty reads and ensures easier recovery.

In this example,  $T_j$  reads  $A$  only after  $T_i$  commits, making the schedule recoverable.

Step	Transaction $T_i$	Transaction $T_j$
1	$w(A)$	
2		$r(A)$
3	$c(T_i)$	
4		$c(T_j)$

Table 3: Recoverable Schedule Example

### 1.13 Precedence Graph and Conflict Serializability Testing

A **Precedence Graph**, also known as a **serialization graph**, is used to test the conflict serializability of a schedule. The graph is constructed with the following steps:

- Create a node for each transaction in the schedule.
- Draw a directed edge from transaction  $T_i$  to transaction  $T_j$  if  $T_i$  performs a conflicting operation before  $T_j$ .
- The schedule is conflict serializable if the precedence graph is acyclic.

Example of testing conflict serializability:

Step	Transaction $T_i$	Transaction $T_j$
1	$r(A)$	
2		$w(A)$
3	$w(B)$	
4		$r(B)$

Table 4: Conflict Serializability Testing Example

In this example, we would draw an edge from  $T_i$  to  $T_j$  for the conflicting operations on  $A$  and  $B$ . If the resulting graph contains no cycles, the schedule is conflict serializable.

### 1.14 Cascading Rollback

**Cascading Rollback** occurs when a transaction fails, causing other dependent transactions to be rolled back as well. This happens when a transaction reads data that was written by another uncommitted transaction. If the first transaction is rolled back, all dependent transactions must also be rolled back, leading to a cascade of rollbacks.

Step	Transaction $T_1$	Transaction $T_2$
1	$w(A)$	
2		$r(A)$
3	<b>failure</b>	
4		rollback

Table 5: Cascading Rollback Example:  $T_2$  reads data from  $T_1$  and is forced to rollback after  $T_1$  fails.

In this example,  $T_2$  reads data  $A$  written by  $T_1$ . When  $T_1$  fails and rolls back,  $T_2$  must also roll back to ensure consistency, causing a cascading rollback.

### 1.14.1 Why Is a Cascadeless Schedule Desirable?

A **Cascadeless Schedule** prevents cascading rollbacks by ensuring that a transaction can only read data from committed transactions. This is desirable because:

- It simplifies the recovery process by eliminating the need for cascading rollbacks.
- Ensures better performance by reducing the number of transactions that need to be rolled back in the event of a failure.
- Helps maintain data consistency by avoiding the propagation of uncommitted changes.

## 1.15 Rules for Preparing a Serializable Schedule

To prepare a **Serializable Schedule**, the following rules must be followed:

- Ensure that conflicting operations (e.g., read/write on the same data item) are properly ordered to maintain serializability.
- Use locks or other concurrency control mechanisms to prevent overlapping operations that would violate serializability.
- Transactions should not read uncommitted data from other transactions to avoid inconsistent states (ensure strict schedules).
- Commit a transaction only after ensuring that all preceding dependent transactions have committed.

**Why Prefer Serializable Schedules Over Serial Schedules?** Serializable schedules allow transactions to execute concurrently while maintaining the same outcome as serial execution, leading to:

- **Increased Throughput:** More transactions can be processed in parallel, improving system efficiency.
- **Reduced Waiting Time:** Transactions do not need to wait for others to complete as in a serial schedule.
- **Better Resource Utilization:** Concurrent execution makes better use of system resources.

## 1.16 Conflict Serializability vs View Serializability



Aspect	Conflict Serializability	View Serializability
<b>Definition</b>	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 produces the same result as a serial schedule, even if the order of non-conflicting operations differs.
<b>Checking Method</b>	Easier to check using <b>precedence graphs</b> .	Harder to check, requires verifying final writes, initial reads, and intermediate writes.
<b>Strictness</b>	More strict, as it only considers direct conflicts between operations.	Less strict, allowing more schedules to be considered serializable.
<b>Example</b>	If $T_i$ reads data item $A$ before $T_j$ writes $A$ , the order must be maintained for conflict serializability.	$T_i$ and $T_j$ may swap operations as long as the final output is the same, even if the intermediate steps differ.
<b>Application</b>	Preferred for simpler concurrency control mechanisms like locking protocols.	Useful when more complex transaction models are involved.

Table 6: Difference Between Conflict Serializability and View Serializability

**Problem-01:**

Check whether the given schedule  $S$  is conflict serializable or not-

$S : R_1(A) , R_2(A) , R_1(B) , R_2(B) , R_3(B) , W_1(A) , W_2(B)$

List all the conflicting operations and determine the dependency between the transactions-

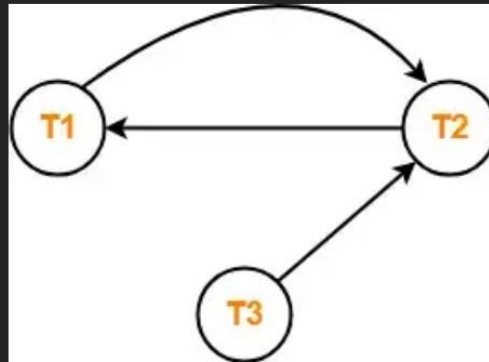
- $R_2(A) , W_1(A)$        $(T_2 \rightarrow T_1)$
- $R_1(B) , W_2(B)$        $(T_1 \rightarrow T_2)$
- $R_3(B) , W_2(B)$        $(T_3 \rightarrow T_2)$

**1.17 Precedence Graph for Conflict Serializability**

A **Precedence Graph** is used to check whether a schedule is conflict serializable. It represents transactions as nodes and conflicting operations as directed edges between them. The schedule is conflict serializable if the precedence graph contains no cycles.

Steps to create a precedence graph:

Draw the precedence graph-



- Clearly, there exists a cycle in the precedence graph.
- Therefore, the given schedule S is not conflict serializable.

- Create a node for each transaction.
- Draw a directed edge from transaction  $T_i$  to  $T_j$  if  $T_i$  performs a conflicting operation before  $T_j$ .
- Check for cycles: If the graph is acyclic, the schedule is conflict serializable.

Example:

Step	Transaction $T_1$	Transaction $T_2$
1	$r(A)$	
2		$w(A)$
3	$w(B)$	
4		$r(B)$

Table 7: Conflict Serializability Example: Precedence Graph Construction

In this example, an edge is drawn from  $T_1$  to  $T_2$  for the conflict on data item  $A$ , and another edge is drawn for the conflict on  $B$ . If the graph contains no cycles, the schedule is conflict serializable.

## 1.18 Logging in Database Systems

A **log** is a sequential record of all the operations performed on a database. It is maintained to ensure that the system can recover from failures by applying or undoing changes. The log contains entries such as:

- $\langle T_i \text{ start} \rangle$
- $\langle T_i, X_j, V1, V2 \rangle$  (indicating a transaction  $T_i$  updated data item  $X_j$  from value  $V1$  to  $V2$ )

- $T_i$  commit
- $T_i$  abort

### **1.18.1 Log-Based Recovery**

Log-based recovery techniques include: 1. **Deferred Database Modification:**

- Changes made by a transaction are not written to the database until the transaction is committed.
- This ensures that if a transaction fails, no changes are made to the database.

**Features:**

- Simplicity in rollback as no changes are applied until commit.
- Lower I/O overhead, as updates are batched and written at once.

2. **Immediate Database Modification:**

- Changes are written to the database immediately as they occur.
- This allows for quick access to the most current data but can lead to inconsistencies if a transaction fails before it commits.

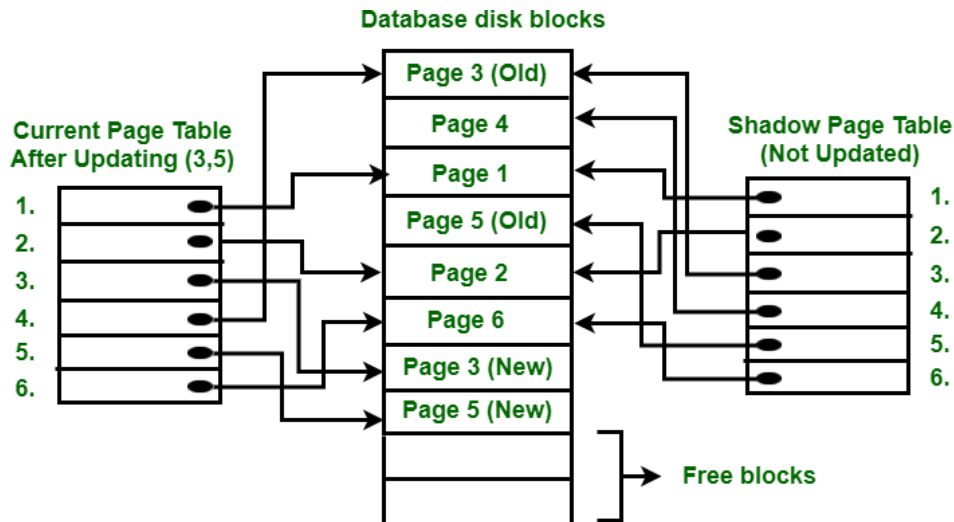
**Features:**

- Allows for more immediate access to data changes.
- Increases the risk of losing uncommitted changes in case of failure.

### **1.19 Shadow Paging Recovery Technique**

**Shadow Paging** is a recovery technique that maintains two page tables: the current page table and a shadow page table.

- Changes are made in a new set of pages rather than overwriting the original pages.
- If a transaction fails, the system can revert to the shadow page table.
- Only committed transactions update the current page table.
- This technique is efficient for read operations.
- Reduces the need for log maintenance, as changes are done in place.
- Shadow paging does not require a log if the system uses only the shadow page table for recovery.



## 1.20 Checkpointing

**Checkpointing** is the process of saving the state of a database at a specific point in time to facilitate recovery in case of failure.

### – Consistent Checkpointing:

- \* Involves creating a checkpoint when the database is in a consistent state.
- \* All transactions that were active at the time of the checkpoint are either committed or aborted to ensure consistency.
- \* Reduces the recovery time by limiting the number of transactions that need to be rolled back.
- \* Requires synchronization among transactions to maintain consistency.

### – Fuzzy Checkpointing:

- \* Allows for checkpoints to be created during transaction execution, not requiring a consistent state.
- \* Provides flexibility in managing large transactions.
- \* Easier to implement as it doesn't require strict synchronization.
- \* May lead to a longer recovery time since some transactions might be partially completed.

## 1.21 Log File

A **log file** records all changes made to the database. It is essential for log-based recovery.

### – Steps for log-based recovery of a system:

1. Identify the last checkpoint in the log.
2. Roll back any uncommitted transactions by reading the log backward.
3. Apply changes from committed transactions by reading the log forward from the last checkpoint.

## 1.22 Recovery Techniques

Various recovery techniques can be implemented in a database system:

- **Log-Based Recovery:**

- \* Advantages: Simple to implement, allows for quick recovery.
- \* Disadvantages: Log management can become complex, leading to performance overhead.

- **Shadow Paging:**

- \* Advantages: Efficient recovery without extensive logging, quick access to uncommitted data.
- \* Disadvantages: More memory usage due to duplicate pages, can be inefficient for large transactions.

- **Checkpoints:**

- \* Advantages: Reduces recovery time, simplifies log management.
- \* Disadvantages: May impact performance during checkpoint creation, requires careful management of transaction states.

## 2 Deadlock

A **deadlock** occurs in a database or concurrent system when two or more transactions are unable to proceed because each is waiting for the other to release a resource. This situation results in a standstill, where none of the transactions can continue executing.

### 2.1 Methods to Handle a Deadlock

There are several methods for handling deadlocks, which include prevention, detection, avoidance, and recovery.

#### 2.1.1 1. Deadlock Prevention

Deadlock prevention involves ensuring that the system will never enter a deadlock state by following certain protocols:

- **Pre-declaration Method:** Transactions must declare all the resources they will need before execution. If the resources are not available, the transaction must wait until all required resources are free.
- **Partial Ordering Method:** Establishes a global order of resource acquisition. Transactions must acquire resources in a predefined order, preventing circular wait conditions.
- **Timestamp Ordering:** Each transaction is given a unique timestamp. Resources can only be allocated if the requesting transaction's timestamp is greater than that of the holding transaction.

### 2.1.2 2. Deadlock Detection

In deadlock detection, the system allows deadlocks to occur and then detects them using various algorithms. The steps typically involve:

- Constructing a wait-for graph to represent transactions and their waiting conditions.
- Periodically checking the wait-for graph for cycles.
- If a cycle is detected, a deadlock exists.
- Implementing appropriate recovery strategies to resolve the deadlock.

### 2.1.3 3. Recovery from Deadlock

Once a deadlock is detected, the system must recover from it. Methods include:

- **Selection of a Victim:** Choosing one of the transactions involved in the deadlock to abort, freeing its resources for others.
- **Rollback:** The chosen transaction is rolled back to a previous state, allowing other transactions to proceed.
- **Starvation:** Continuous prevention of a transaction from proceeding if it is repeatedly chosen as a victim; it may need to be monitored to prevent indefinite blocking.

### 2.1.4 4. Deadlock Avoidance

Deadlock avoidance ensures that the system never enters a deadlock state by managing how resources are allocated:

- **Serial Access:** Transactions access resources in a serial order.
- **Auto-commit Transactions:** Transactions are automatically committed after their execution, reducing the potential for deadlocks.
- **Ordered Updates:** Ensuring that all transactions update resources in a predefined order.

## 2.2 Deadlock Prevention Schemes

Deadlock prevention can also be managed through specific schemes:

### 2.2.1 1. Wait-Die Scheme

In the wait-die scheme, if a transaction  $T_s$  requests a resource held by a younger transaction  $T_r$ ,  $T_s$  is aborted (dies). If  $T_s$  is older, it waits.

### 2.2.2 2. Wound-Wait Scheme

In the wound-wait scheme, if a transaction  $T_s$  requests a resource held by a younger transaction  $T_r$ ,  $T_r$  is aborted (wounded). If  $T_s$  is younger, it waits.

## 2.3 Necessary Conditions for Deadlock

For a deadlock to occur, the following four conditions must hold simultaneously:

- **Mutual Exclusion:** At least one resource must be held in a non-shareable mode.
- **Hold and Wait:** Transactions holding resources are allowed to request additional resources.
- **No Preemption:** Resources cannot be forcibly taken from transactions holding them.
- **Circular Wait:** There exists a circular chain of transactions where each transaction is waiting for a resource held by the next transaction in the chain.

## 3 Distributed Database

### 3.1 Distributed Data Storage

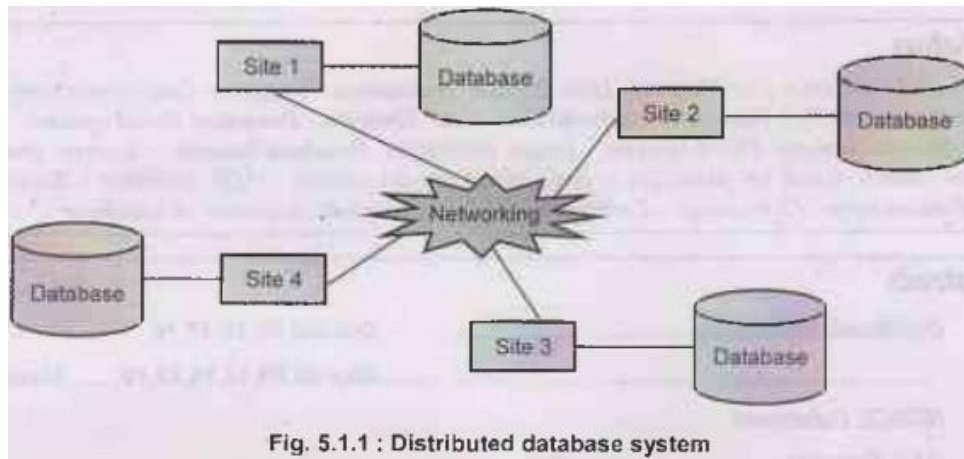
Data is stored across multiple locations to ensure better availability, reliability, and performance.

**Example:** A company's data could be stored in multiple branches located in different cities.

- Enhances data availability and fault tolerance.
- Enables load balancing by distributing the workload across multiple sites.
- Increases performance by reducing latency, as data can be accessed from the nearest location.
- Provides data redundancy, reducing the risk of data loss.
- Allows for scalability, accommodating growth in data volume.
- Supports localized control over data, adhering to legal and regulatory requirements.
- Facilitates disaster recovery by distributing data backups across various locations.

### 3.2 What is a Distributed Database?

A distributed database is a collection of interconnected databases spread across different physical locations but functioning as a unified system. The system allows for the management of data across various sites while providing transparent access to the users as if it were a single entity.



### 3.2.1 Advantages of Distributed Databases

1. Improved reliability and availability due to data replication and redundancy.
2. Enhanced performance by allowing faster data access from local databases.
3. Scalability to handle growing data and user demand.
4. Flexibility in data management across different locations.
5. Local autonomy, allowing each site to control its data.
6. Reduced communication overhead for localized transactions.
7. Fault tolerance with automatic failover mechanisms.

### 3.2.2 Disadvantages of Distributed Databases

1. Complex system management, requiring more maintenance and coordination.
2. High communication costs when synchronizing data between different locations.
3. Difficulties in ensuring global consistency of data.
4. Increased security risks due to data being spread across multiple locations.
5. Network failures can disrupt access to remote data.
6. Complex query optimization over multiple sites.
7. Data integrity challenges across distributed environments.

## 3.3 Atomic Commit Protocols

**Atomic commit protocols** ensure that a distributed transaction is either committed at all sites or aborted at all sites to maintain data consistency.

- **AC1:** One-phase commit protocol.
- **AC2:** Two-phase commit protocol.
- **AC3:** Three-phase commit protocol.

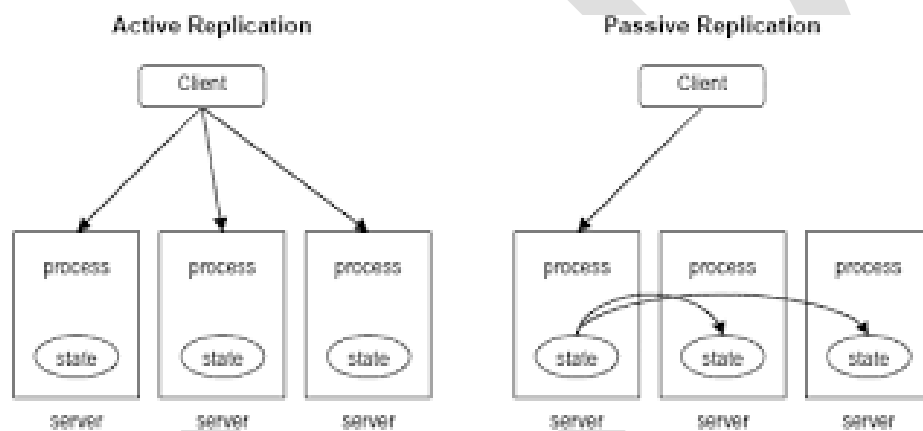


### 3.4 Replication in Distributed Systems

**Replication** is the process of storing copies of data across multiple locations to ensure availability, fault tolerance, and improved access speed.

#### 3.4.1 Types of Replication

- **Active Replication:** Involves all replicas processing the same operations simultaneously to ensure consistency.
- **Passive Replication:** Only one replica processes requests, and others are updated after the changes are committed.



### 3.5 Data Fragmentation

**Data Fragmentation** refers to dividing a database into smaller, logical pieces to improve performance and manageability.

#### 3.5.1 Types of Data Fragmentation

- **Vertical Fragmentation:** Dividing a table by its columns, grouping attributes that are frequently accessed together.
- **Horizontal Fragmentation:** Dividing a table into rows based on specific conditions.
  - \* **Primary Fragmentation:** Fragmenting data based on one condition.
  - \* **Derived Fragmentation:** Fragmenting based on a previously fragmented table.
- **Hybrid Fragmentation:** A combination of vertical and horizontal fragmentation.

### 3.5.2 Advantages of Data Fragmentation

1. Improved query performance by localizing data access.
2. Enhanced security by limiting access to specific fragments.
3. Better data distribution, reducing network congestion.
4. Increased manageability by splitting data logically.

### 3.5.3 Disadvantages of Data Fragmentation

1. Complex query reconstruction from fragments.
2. Data redundancy in hybrid fragmentation may increase storage costs.
3. Difficulty in maintaining consistency across fragments.
4. Increased complexity in fragment allocation and management.

## 3.6 Replication Transparency vs Fragmentation Transparency

Replication Transparency	Fragmentation Transparency
Ensures users are unaware of data replication	Ensures users are unaware of data fragmentation
Handles data availability by replicating data across multiple sites	Handles logical division of data into fragments for efficient access
Improves fault tolerance through data redundancy	Improves query performance by accessing data from relevant fragments
Maintains consistency across all replicas	Maintains consistency across all fragments
Requires synchronization to ensure that all replicas have the latest data	Requires reconstruction when combining fragments to retrieve complete data
Deals with multiple copies of data distributed across sites	Deals with data partitioned into distinct segments across sites
Affects system scalability due to the overhead of maintaining multiple copies	Affects query efficiency as data needs to be reassembled from fragments

## 3.7 Types of Distributed Databases

- **Homogeneous Distributed Database:** All sites use the same database management system (DBMS).
- **Heterogeneous Distributed Database:** Different sites may use different DBMS, but they are connected to function as a unified system.

## 3.8 Concurrency Control

Concurrency control ensures that multiple transactions can occur concurrently without leading to data inconsistency. Techniques include:

- **Locking Protocols:** Ensures that multiple transactions do not interfere with each other by locking resources during a transaction.
- **Timestamp Ordering:** Conflicting operations are executed based on the timestamps of transactions, ensuring order and consistency.
- **Optimistic Concurrency Control:** Transactions execute without locking resources and check for conflicts only at the end of the transaction.
- **Multiversion Concurrency Control (MVCC):** Maintains multiple versions of data to allow concurrent transactions without locking.
- **Validation-Based Protocols:** Transactions are validated before they commit to ensure they don't conflict with other transactions.

### 3.9 Frequency Control

Frequency control refers to the regulation of transaction frequency and access patterns to ensure data consistency and system performance in a database system. It is needed because:

1. To ensure **consistency** in data when multiple transactions are accessing and updating the same data simultaneously.
2. Prevents **lost updates**, where changes made by one transaction are overwritten by another without proper synchronization.
3. Avoids **dirty reads**, where a transaction reads uncommitted data from another transaction, leading to inconsistent results.
4. Prevents **non-repeatable reads**, where a transaction reads the same data multiple times and gets different results due to concurrent updates.
5. Ensures **serializability**, where the execution of concurrent transactions results in a state that is equivalent to executing them sequentially.
6. Avoids **phantom reads**, where a transaction reads a set of records that may be modified or deleted by another transaction during execution.
7. Helps in regulating **transaction throughput** and system performance, reducing conflicts and improving data access speed.

### 3.10 Concurrency Control Mechanisms in Distributed Databases

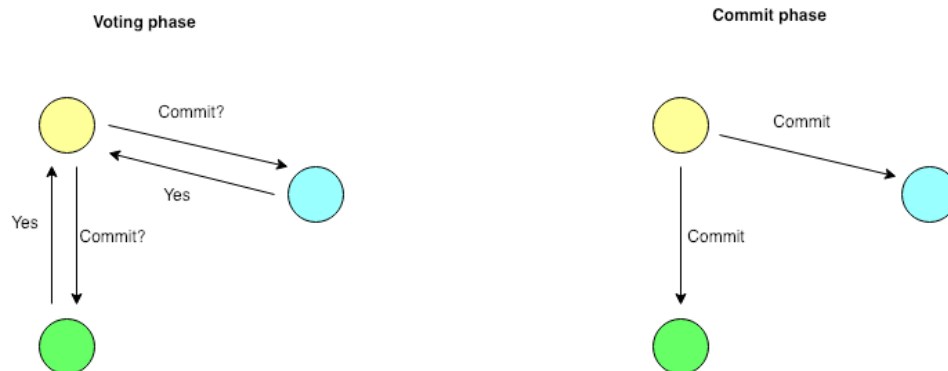
Concurrency control mechanisms in distributed databases ensure that transactions in different locations occur without inconsistencies.

#### 3.10.1 Two-Phase Commit Protocol (2PC)

The **two-phase commit protocol** is used to ensure atomicity in distributed transactions. It has two phases:

- **Phase 1 - Prepare Phase:** The coordinator sends a request to all participants to prepare the transaction. Each participant responds with a 'ready' or 'abort' message.

- **Phase 2 - Commit Phase:** If all participants are 'ready', the coordinator sends a commit request. If any participant sends an 'abort' message, the coordinator sends an abort message to all participants, and the transaction is rolled back.



### 3.11 Directory System

A directory system is an essential component in distributed databases that provides information about the location and structure of data across the distributed system. It acts as a map or index that helps locate data distributed over multiple sites.

#### Detailed Explanation of Directory System:

- The directory contains metadata about the distributed data, such as data location, partitioning, replication details, and access paths.
- It provides transparency to users, hiding the complexities of data distribution and allowing users to access data as if it were stored in a single location.
- The directory manages mapping between logical data and physical storage, making data retrieval efficient.
- It plays a crucial role in query optimization by guiding the system to the appropriate location of data fragments or replicas.
- The directory system helps in load balancing by distributing query loads across different sites.
- It supports fault tolerance by allowing the system to locate alternative replicas if the primary data source fails.
- The directory can either be centralized (with one global directory) or distributed, with each site maintaining its own directory but interconnected.

---

# Database Management System (BCS501)

## UNIT 5: Concurrency Control Techniques

### Syllabus:

- 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

### 1 Concurrency Control

Concurrency control in DBMS refers to the techniques used to manage simultaneous operations on the database, ensuring that the database remains consistent. Below are four key points about concurrency control:

- It helps manage the execution of transactions in parallel.
- Ensures that no conflicting operations occur during simultaneous transactions.
- Prevents the violation of database integrity.
- Maintains serializability, ensuring that the outcome of concurrent transactions is the same as if they were executed sequentially.

---

## Why Concurrency Control is Needed

Concurrency control is necessary in DBMS for the following reasons:

- To ensure consistency in the database during simultaneous transactions.
- To prevent issues such as:
  1. **Lost Update:**
    - Occurs when two or more transactions read the same data and update it simultaneously.
    - The final value depends on the last update, leading to a lost update by the earlier transaction.
  2. **Dirty Read:**
    - Occurs when a transaction reads data that has been updated by another uncommitted transaction.
    - If the second transaction is rolled back, the first transaction would have used invalid or dirty data.

**Example:** Consider two transactions,  $T_1$  and  $T_2$ , both trying to update the same record in a database. Without proper concurrency control,  $T_1$  could overwrite changes made by  $T_2$ , leading to inconsistent data.

## 2 Locking Techniques for Concurrency Control

Locking is one of the most common techniques for concurrency control in DBMS. A lock is a mechanism that restricts access to a database item while it is being used by a transaction, ensuring data integrity during concurrent access.

### Different Types of Locks

The different types of locks are:

1. **Binary Lock:**
  - A binary lock has two states:
    - **Locked (1):** The data item is currently in use.
    - **Unlocked (0):** The data item is available for access.
  - It can lock or unlock a database item.
2. **Shared Lock (S-lock):**
  - Allows multiple transactions to read the data item but prevents any updates until all shared locks are released.
3. **Exclusive Lock (X-lock):**
  - Prevents any other transaction from accessing or modifying the data item.

---

If the simple binary locking scheme is used, every transaction must obey the following rules:

1. A transaction can lock a data item only if it is currently unlocked.
2. A transaction cannot unlock a data item until it has completed its operations.
3. No transaction can lock a data item that is currently locked by another transaction.
4. A transaction must unlock all data items before it can terminate.

## Shared and Exclusive Locks

Shared and exclusive locks are more complex types of locks used in transaction management:

### 1. Shared Lock (S-lock):

- Allows concurrent transactions to read a data item but not modify it.

### 2. Exclusive Lock (X-lock):

- Only one transaction can hold the exclusive lock, preventing others from reading or writing.

If the shared/exclusive locking scheme is used, every transaction must obey the following rules:

1. A transaction can hold multiple shared locks for reading data.
2. A transaction can upgrade a shared lock to an exclusive lock when it needs to modify data.
3. Once an exclusive lock is acquired, no other transaction can hold any lock on that data item until the exclusive lock is released.
4. A transaction must request a lock before accessing a data item.
5. Locks must be released only after the transaction completes to ensure data integrity.
6. Deadlock detection and resolution mechanisms should be in place to handle potential deadlocks.

**Example:** In a bank database, if one transaction is updating an account balance (using an exclusive lock), another transaction cannot read or modify that balance until the update is complete and the lock is released.

## Lock Compatibility

Lock compatibility refers to the conditions under which multiple transactions can acquire locks on the same data item without causing conflicts. The compatibility of shared and exclusive locks can be illustrated as follows:

Lock Mode	Shared Lock (S-lock)	Exclusive Lock (X-lock)
Shared Lock (S-lock)	Yes	No
Exclusive Lock (X-lock)	No	Yes

Table 1: Lock Compatibility Table

## How is Lock Implemented

Lock implementation involves the following steps:

### 1. Lock Request:

- A transaction requests a lock on a data item before accessing it.
- The request is made to the lock manager, which checks the lock compatibility with existing locks.

### 2. Lock Granting:

- If the lock is compatible, the lock manager grants the lock to the transaction.
- If not compatible, the transaction may be put into a wait state until the lock becomes available.

### 3. Lock Types:

- Depending on the operation, a transaction may request either a shared lock (for read operations) or an exclusive lock (for write operations).

### 4. Locking Protocols:

- Transactions follow specific locking protocols (like Two-Phase Locking) to ensure data consistency and avoid deadlocks.

### 5. Unlock Request:

- After completing its operations, the transaction releases the lock on the data item.
- This is also communicated to the lock manager, which updates the lock status.

### 6. Lock Timeout:

- In some implementations, if a transaction waits too long for a lock, it may timeout and roll back, allowing other transactions to proceed.

### 7. Deadlock Detection:

- The system periodically checks for deadlocks, where two or more transactions are waiting indefinitely for locks held by each other.
- If detected, the system resolves the deadlock by aborting one of the transactions.

### 8. Lock Upgrade/Downgrade:



- A transaction holding a shared lock may request to upgrade it to an exclusive lock when it needs to write to the data item.
- Similarly, a transaction may release an exclusive lock and downgrade it to a shared lock if it only needs to read.

## Typical Lock Manager Implementation

A lock manager is responsible for managing lock requests and ensuring that transactions follow the locking protocols. It typically uses atomic operations to grant and release locks.

## Reason for Lock and Unlock

Locks are implemented to maintain data integrity and consistency during concurrent transactions. Unlocking is necessary to allow other transactions to access the data once the current transaction has completed.

## Difference Table: Lock vs. Latch

Feature	Lock	Latch
Scope	Higher level (transactions)	Lower level (memory structures)
Purpose	Ensure data consistency in transactions	Protect data structures during modification
Duration	Held until transaction commits or aborts	Held for a short duration (typically during a single operation)
Compatibility	Supports multiple modes (shared/exclusive)	Typically binary (locked/unlocked)
Overhead	Higher overhead due to managing multiple modes and transaction states	Lower overhead, generally faster due to simpler mechanism
Usage	Commonly used in databases to ensure transaction isolation	Used in low-level programming to synchronize access to shared resources

Table 2: Difference between Lock and Latch

## Convoy

A convoy occurs when multiple transactions are delayed while waiting for locks, causing reduced system performance. When one transaction holds a lock for an extended period, other transactions may form a convoy, leading to inefficiencies.

---

## Short Notes on Lock-Based Protocol

Lock-based protocols manage concurrent access to database resources using locks. These protocols ensure that transactions follow strict rules for acquiring and releasing locks, preventing conflicts and ensuring consistency.

### Two-Phase Locking Technique for Concurrency Control (2PL)

Two-phase locking (2PL) is a concurrency control mechanism used in database management systems to ensure that transactions are executed in a serializable manner. This technique consists of two distinct phases: the Growing Phase and the Shrinking Phase.

- **Growing Phase:**

- In this phase, a transaction can acquire locks on data items but is not allowed to release any locks.
- The transaction continues to obtain the necessary locks to ensure that it can read or write the data it requires for its operations.
- This phase continues until the transaction has acquired all the locks it needs or until it has been forced to stop due to waiting on another transaction.
- **Example:** If Transaction *T1* needs to read data items *A*, *B*, and *C*, it will request and acquire locks on these items during the Growing Phase.

- **Shrinking Phase:**

- In this phase, the transaction can release the locks it has acquired but cannot acquire any new locks.
- This phase allows for the release of locks, thereby making data items available for other transactions to access.
- Once a transaction releases a lock, it cannot obtain any additional locks, which helps prevent deadlocks.
- **Example:** After completing its operations, Transaction *T1* releases the locks on data items *A*, *B*, and *C*, allowing other transactions to access these items.

#### Salient Features of Two-Phase Locking:

- **Serializability:** 2PL guarantees that the schedule of transactions will be serializable, meaning the transactions can be executed in a sequence that does not alter the final outcome compared to executing them in isolation.
- **Locking Order:** To avoid deadlocks, it is often recommended to enforce a strict locking order when acquiring locks on multiple data items.
- **Types of 2PL:**
  - **Strict 2PL:** A more restrictive version where a transaction cannot release any locks until it has completed its execution. This guarantees strict serializability.
  - **Conservative 2PL:** Allows releasing locks during the Growing Phase, but ensures that once a lock is released, no new locks can be acquired. This can lead to less contention but may not guarantee serializability.

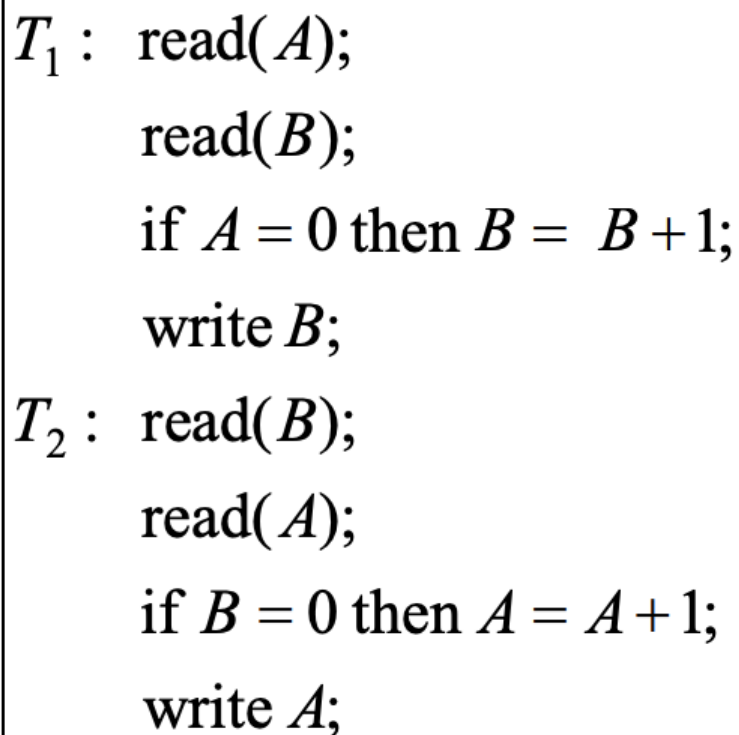
---

**Advantages of 2PL:**

- Simple to implement and understand.
- Effective in preventing lost updates, dirty reads, and other concurrency-related issues.

**Disadvantages of 2PL:**

- Potential for deadlocks if transactions are not managed properly.
- May lead to reduced concurrency since transactions may need to wait for locks.



```
 $T_1$  : read( $A$ );  
      read( $B$ );  
      if  $A = 0$  then  $B = B + 1$ ;  
      write  $B$ ;  
 $T_2$  : read( $B$ );  
      read( $A$ );  
      if  $B = 0$  then  $A = A + 1$ ;  
      write  $A$ ;
```

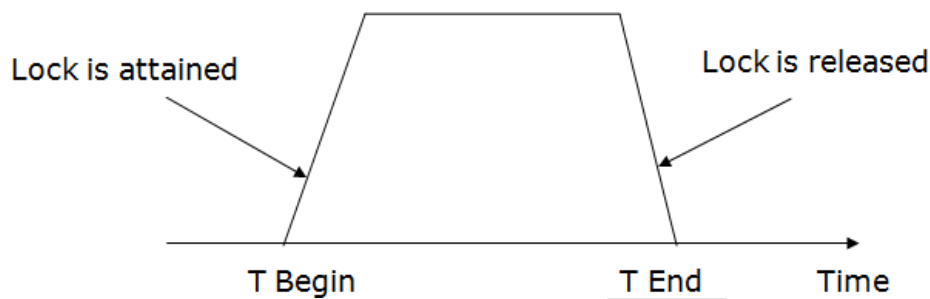
Figure 1. Two transactions  $T_1$  and  $T_2$

**Example:**

- Transaction  $T_1$ :
  - Acquire a read lock on item  $Y$ .
  - Read item  $y$ .

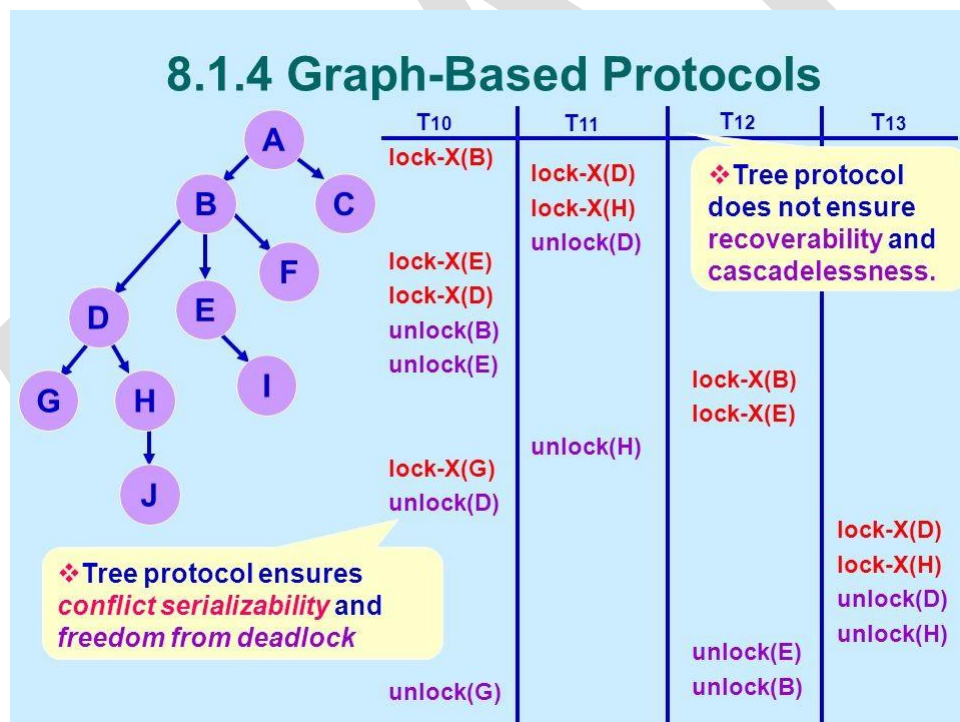
**Discuss Strict 2PL**

Strict Two-Phase Locking (Strict 2PL) requires that transactions hold all their locks until they commit or abort. This ensures that no transaction can release a lock until its completion, thereby preventing any other transaction from accessing the data item.



## Salient Features of Graph-Based Locking Protocol

Graph-based locking protocols use a directed graph to represent transactions and the locks they hold. Each node represents a transaction, and edges represent the locks. If a cycle is detected in the graph, it indicates a deadlock situation.



## Two-Phase Locking Technique Guarantees Serializability

The Two-Phase Locking technique guarantees serializability because it ensures that once a transaction releases a lock, it cannot obtain any more locks. This mimics a serial execution of transactions, maintaining data consistency.

## Problems Faced When Concurrent Transactions Execute Uncontrolled

When concurrent transactions execute without control, various problems can arise:

## 1. The Lost Update Problem:

- (a) Transaction  $T_1$  reads a value.
- (b) Transaction  $T_2$  also reads the same value and updates it, causing  $T_1$ 's changes to be lost.

## Lost update problem

**Example:** Bank account, two parallel withdrawals

Time	Transaction 1: Withdraw 200 from account 123	Transaction 2: Withdraw 100 from account 123
↓	$\text{bal} = \text{read\_balance}(123) = 1000 \text{ €}$ $\text{bal} = \text{bal} - 200 \text{ €}$  $\text{write\_balance}(123, \text{bal}) = 800 \text{ €}$	$\text{bal} = \text{read\_balance}(123) = 1000 \text{ €}$ $\text{bal} = \text{bal} - 100 \text{ €}$  $\text{write\_balance}(123, \text{bal}) = 900 \text{ €}$

- The first update is lost. Note that the transactions are not aware of each other's internal program variables (bal).

AdvDB-3 J. Teuhola 2015

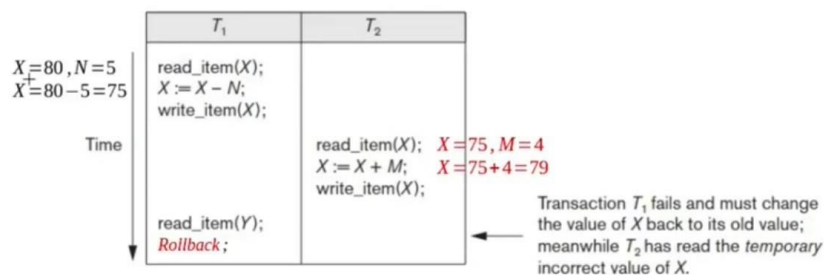
68

## 2. The Dirty Read Problem:

- (a) Transaction  $T_1$  reads uncommitted changes from  $T_2$ .
- (b) If  $T_2$  rolls back,  $T_1$  has read invalid data.
- (c) This leads to inconsistencies and incorrect results.

## The Dirty Read problem


- This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value.



### 3. The Incorrect Summary Problem:

- (a) Transaction  $T_1$  calculates a summary based on intermediate results.
- (b) If  $T_2$  modifies the underlying data,  $T_1$ 's summary becomes incorrect.

#### Incorrect summary problem

Time	Transaction 1 <i>Sum of balances</i>	Transaction 2 <i>Transfer 100 € from 789 to 123</i>
	<pre>sum = 0 bal = read_balance(123) = 1000 € sum = sum + bal = 1000 € bal = read_balance(456) = 2000 € sum = sum + bal = 3000€  bal = read_balance(789) = 2900 € sum = sum + 2900 € = 5900 € // <b>Should be 6000 €</b></pre>	<pre>bal = read_balance(789) = 3000 € bal = bal - 100 € write_balance(789, bal) = 2900 € bal = read_balance(123) = 1000 € bal = bal + 100€ write_balance(123, bal) = 1100 €</pre>

AdvDB-3 J. Teuhola 2015

70

### Role of Locks

Locks play a crucial role in managing concurrent access to data in a database. They ensure that only one transaction can modify a data item at a time, thereby preventing conflicts and maintaining data integrity during concurrent operations.

## 3 Time Stamping Protocols for Concurrency Control

Time-stamping protocols are essential in database management systems to handle concurrency control by ensuring that transactions are executed in a serializable order. Each transaction is assigned a unique timestamp based on its arrival time, which helps maintain a consistent state of the database by preventing conflicts.

### 3.1 Example

Consider two transactions,  $T_1$  and  $T_2$ , arriving at times  $t_1$  and  $t_2$ , respectively. If  $t_1 < t_2$ ,  $T_1$  will be executed first, followed by  $T_2$ . This establishes a clear order of execution based on the timestamps.

---

## 3.2 Timestamp-based Protocols

### 3.2.1 Timestamps

- **W-timestamp(Q)**: The latest write timestamp for data item  $Q$ .
- **R-timestamp(Q)**: The latest read timestamp for data item  $Q$ .

## 3.3 Timestamp Ordering Protocol

1. **Read Operation**: If transaction  $T_i$  issues a read operation for data item  $Q$ :
  - It checks the R-timestamp of  $Q$ . If the R-timestamp is greater than the timestamp of  $T_i$ , the read operation is rejected (conflict). Otherwise,  $T_i$  can proceed with the read, and the R-timestamp of  $Q$  is updated to  $T_i$ 's timestamp.
2. **Write Operation**: If transaction  $T_i$  issues a write operation for data item  $Q$ :
  - It checks the W-timestamp and R-timestamp of  $Q$ . If either of these timestamps is greater than the timestamp of  $T_i$ , the write operation is rejected (conflict). Otherwise,  $T_i$  can proceed with the write, and the W-timestamp of  $Q$  is updated to  $T_i$ 's timestamp.

## 3.4 Advantages and Disadvantages

- **Advantages**:
  - *Serializability*: Ensures a consistent execution order for transactions.
  - *Conflict Resolution*: Automatically handles conflicts through timestamp comparison.
- **Disadvantages**:
  - *Starvation*: Older transactions may starve if younger transactions keep arriving.
  - *Overhead*: Maintaining timestamps adds some overhead to transaction management.

## 4 Short Notes on

### 4.1 Thomas's Write Rule

**Thomas's Write Rule** is a refinement of the basic timestamp ordering protocol. It allows a transaction to write an item even if its timestamp is older than the current W-timestamp of that item, provided that:

- The read timestamp (R-timestamp) is older than the writing transaction's timestamp, meaning it has been read by a transaction that has already committed.

This rule helps reduce unnecessary aborts and improves concurrency by allowing more writes.

---

## 4.2 Strict Timestamp Ordering Protocol

**Strict Timestamp Ordering Protocol** requires that:

- Transactions must be executed in the order of their timestamps without any exceptions. This ensures that no transaction can be aborted due to conflicts arising from timestamp comparisons.
- In this protocol, once a transaction is assigned a timestamp, it must either be completed or aborted.

The strictness of this protocol guarantees serializability but may lead to increased transaction wait times and decreased concurrency.

## 5 Validation Protocol in Concurrency Control

The **Validation Protocol** consists of three phases: Read Phase, Validation Phase, and Write Phase.

### 5.1 1. Read Phase

1. **Step A:** A transaction  $T_i$  reads data item  $Q$ .
2. **Step B:** It keeps a record of the read operation.

### 5.2 2. Validation Phase

During this phase, the system checks whether the transaction can be committed based on its read operations and the states of other transactions.

### 5.3 3. Write Phase

1. **Step 1:** If the validation is successful,  $T_i$  can write its changes to the database.
2. **Step 2:** Update the timestamps accordingly.
3. **Step 3:** If the validation fails,  $T_i$  is aborted.

## 6 Phantom Phenomenon

The **phantom phenomenon** occurs when a transaction reads a set of rows that match a certain condition, and another concurrent transaction inserts or deletes rows that affect the result set of the first transaction, leading to inconsistent results.



---

## 6.1 Timestamp-Based Protocol to Avoid Phantom Phenomenon

To devise a timestamp-based protocol that avoids the phantom phenomenon, we can implement the following steps:

1. Each transaction will acquire locks on the range of data items it intends to read or write.
2. Before committing, a transaction will check whether any new transactions have modified the range it accessed since it started.
3. If modifications are detected, the transaction will be aborted and must restart.

This approach ensures that transactions maintain consistency and avoid reading inconsistent or invalid data.

## 7 Validation Based Protocol

In the validation-based protocol, transactions execute in three phases:

- **Read Phase:** Transaction reads data items and performs operations.
- **Validation Phase:** Checks whether transactions can be committed without violating serializability.
- **Write Phase:** Updates the database.

**Example:** During the validation phase, if two transactions  $T_1$  and  $T_2$  conflict, one is rolled back.

## 8 Multiple Granularity

Multiple granularity allows locking at different levels of granularity, such as tuples, pages, or entire tables. It provides more flexibility by allowing transactions to lock only the required data granularity, reducing the possibility of conflicts.

**Example:** If a transaction only needs to update a single record, it can lock that record without locking the entire table.

### 8.1 What Do You Mean by Multiple Granularities?

Multiple granularity refers to the ability of a transaction management system to lock data at different levels of detail or granularity. Instead of only locking entire tables or individual records, multiple granularity allows for locking at various levels, such as pages, tuples, or entire tables, based on the needs of the transaction.

---

## 8.2 How It Is Implemented in Transaction System

The implementation of multiple granularity in transaction systems is typically organized in a tree structure, where:

- The root node represents the entire database.
- Intermediate nodes represent tables or pages.
- Leaf nodes represent individual records or tuples.

This tree structure allows transactions to lock only the required nodes instead of the entire hierarchy, improving concurrency and reducing conflicts.

## 8.3 Multiple Granularity Protocol of Concurrency Control

The multiple granularity protocol enables various types of locks at different granularity levels, such as:

- **Intension Shared (IS):** Indicates a transaction intends to acquire shared locks on lower-level resources.
- **Intension Exclusive (IX):** Indicates a transaction intends to acquire exclusive locks on lower-level resources.
- **Shared and Intension Exclusive (SIX):** Allows a transaction to hold shared locks at one level while having the intent to acquire exclusive locks at a lower level.

Table 3: Multiple Granularity Protocol Types

Lock Type	Description
Intension Shared (IS)	Allows shared locks on lower-level resources.
Intension Exclusive (IX)	Allows exclusive locks on lower-level resources.
Shared and Intension Exclusive (SIX)	Holds shared locks while intending to acquire exclusive locks at lower levels.

## 8.4 What is Granularity Locking?

Granularity locking refers to the locking mechanism that determines the size of data items that can be locked (e.g., table, page, record).

---

#### **8.4.1 How Does Granularity of Data Item Affect the Performance of Concurrency Control?**

The granularity of data items significantly affects concurrency control performance:

- Fine Granularity (e.g., locking individual records): Increases concurrency but may result in higher overhead due to frequent locking and unlocking.
- Coarse Granularity (e.g., locking entire tables): Reduces overhead but can lead to conflicts and decreased concurrency, as multiple transactions cannot access different records in the same table simultaneously.

#### **8.4.2 Factors Affecting the Selection of Granularity Size of Data Item**

Several factors influence the choice of granularity size:

- Transaction Characteristics: If transactions typically access multiple records, a coarser granularity may be more efficient.
- System Load: Higher system load may benefit from finer granularity to improve concurrency.
- Data Access Patterns: The nature of data access (read-heavy vs. write-heavy) can guide granularity selection.
- Overhead Considerations: The cost of managing locks at different granularities must be balanced against the benefits of reduced contention.

## **9 Multi-Version Concurrency Control**

Multi-version concurrency control allows multiple versions of data items. This approach enables transactions to read the latest committed version of the data, ensuring that no transaction waits for others to finish.

### **9.1 Explain Multi-Version Timestamping Protocol**

The multi-version timestamping protocol maintains several versions of data items, allowing transactions to access different versions based on their timestamps. The key components of this protocol include:

- Each data item has multiple versions, each associated with a timestamp indicating when it was created.
- When a transaction wants to read a data item, it retrieves the version whose timestamp is closest but less than or equal to its own timestamp.
- When a transaction writes to a data item, it creates a new version with the current timestamp.

---

This protocol enhances concurrency by allowing transactions to read old versions while other transactions update data.

## 10 Multi-Version Two-Phase Locking

Multi-version two-phase locking combines multi-versioning with the two-phase locking protocol. In this system, transactions operate under a locking protocol while allowing multiple versions of data.

Table 4: Multi-Version Two-Phase Locking

Lock Type	Operation	Certify
Shared	Read from a version	Yes
Exclusive	Write a new version	No
Certify	Validate transaction	No

## 11 Methods to Avoid Problems

Several methods can be employed to avoid concurrency control problems:

- **Deadlock Prevention:** Implementing protocols that ensure transactions do not enter a deadlock state by acquiring locks in a defined order.
- **Wait-Die and Wound-Wait Schemes:** These schemes help manage transaction priorities and prevent deadlocks by assigning timestamps and deciding which transaction waits or aborts.
- **Timeouts:** Setting time limits for transactions to acquire locks, after which they are aborted to prevent indefinite waiting.
- **Locking Granularity Control:** Adjusting the granularity of locks dynamically based on system load and transaction behavior.
- **Optimistic Concurrency Control:** Allowing transactions to proceed without locking but validating them before committing to ensure conflicts do not occur.

## 12 Recovery with Concurrent Transactions

Recovery in the presence of concurrent transactions ensures that the database can be restored to a consistent state after a crash, even when multiple transactions were being executed.

**Example:** If a system crashes while  $T_1$  and  $T_2$  are running, recovery protocols ensure that all committed transactions remain intact, while incomplete transactions are rolled back.

---

## 12.1 Recovery Mechanisms

### 12.1.1 1) Interaction with Concurrency Control

- Recovery mechanisms work in tandem with concurrency control to maintain database consistency. They ensure that transactions are either fully completed or completely rolled back, preventing partial updates that could lead to inconsistencies.
- Concurrency control ensures that concurrent transactions do not interfere with each other, while recovery protocols guarantee that the effects of committed transactions persist despite failures.

### 12.1.2 2) Transaction Rollback

- Transaction rollback is the process of reverting the database to its previous state in case of a failure or error. This ensures that any changes made by a transaction that did not complete successfully are undone.
- Rollback can be initiated automatically by the database system during recovery, ensuring that incomplete transactions do not leave the database in an inconsistent state.

### 12.1.3 3) Checkpoints

- Checkpoints are predetermined points in time when the database system saves a snapshot of its current state. This allows the recovery process to start from the last checkpoint rather than from the very beginning, reducing recovery time.
- During recovery, the system can quickly determine which transactions were committed or rolled back since the last checkpoint, minimizing the amount of work needed to restore the database to a consistent state.

### 12.1.4 4) Restart Recovery

- Restart recovery involves analyzing the log files and the checkpoint to identify which transactions were committed and which need to be rolled back after a crash.
- The process typically includes applying the effects of committed transactions and undoing the effects of transactions that were not completed at the time of the crash, ensuring that the database is restored to a consistent state.

## 13 Case Study of Oracle

Oracle Database uses sophisticated concurrency control mechanisms, including multi-version concurrency control and locking protocols. Oracle's automatic mechanisms ensure efficient transaction management and recovery.

**Example:** Oracle's automatic undo records maintain a history of changes, allowing read consistency and recovery in case of failures.

---

## 13.1 Data Storage in Oracle RDBMS

Oracle RDBMS stores data in a structured format, ensuring efficient data retrieval and management.

1. **Data Files:** These are the physical files on the disk that store the actual data. Each tablespace in Oracle is associated with one or more data files.
2. **Control Files:** These files contain metadata about the database, such as the database name, the names and locations of data files, and the current state of the database.
3. **Redo Log Files:** These files record all changes made to the database, ensuring that data can be recovered in the event of a failure.

## 13.2 Disk Files Used in Oracle

### 13.2.1 1) Data Files

- These files store user data and are organized into tablespaces.
- Each data file belongs to only one tablespace and cannot be shared among multiple tablespaces.

### 13.2.2 2) Control Files

- Control files store critical information about the database's structure and state.
- They are required for the database to start up and to ensure data integrity.

## 13.3 Database Schema

A database schema defines the organization of data as a blueprint of how the database is constructed.

1. **Tables:** Defined structures for storing data.
2. **Views:** Virtual tables representing a subset of the database.
3. **Indexes:** Data structures that improve the speed of data retrieval operations.
4. **Constraints:** Rules that ensure data integrity.
5. **Relationships:** Associations between tables.
6. **Procedures:** Stored programs for executing business logic.

---

## **13.4 Definitions of Key Terms**

### **13.4.1 Tablespace**

1. A logical storage unit in Oracle that groups related data files.
2. It can contain multiple data files, allowing for flexible data management.
3. Tablespaces help in managing disk space and performance.
4. Each tablespace can be defined as permanent or temporary.
5. Used for allocating space for database objects.

### **13.4.2 Package**

1. A package is a collection of related PL/SQL types, variables, and subprograms.
2. It consists of a specification (interface) and a body (implementation).
3. Packages allow for better organization of code.
4. They provide encapsulation and improved performance by reducing the need for recompilation.

### **13.4.3 Schema**

1. A schema is the structure that represents the logical view of the database.
2. It contains definitions of database objects like tables, views, and indexes.
3. Each schema is associated with a specific database user.
4. Schemas help in organizing database objects to avoid naming conflicts.

## **13.5 SQL\*Plus, SQL\*Net, and SQL Loader**

### **13.5.1 SQL\*Plus**

1. SQL\*Plus is an interactive and batch query tool for Oracle Database.
2. It allows users to execute SQL commands and PL/SQL blocks.
3. It provides a command-line interface to manage database objects and execute scripts.

### **13.5.2 SQL\*Net**

1. SQL\*Net is a networking protocol that allows communication between Oracle database clients and servers.
2. It facilitates remote access to the database over networks.

---

### 13.5.3 SQL Loader

1. SQL Loader is a tool for loading data from external files into Oracle tables.
2. It can handle various file formats and allows for data transformation during loading.

## 13.6 Locking Techniques

Various locking techniques are used in database management systems to ensure data consistency and integrity. Some common techniques include:

- Binary Locking: A simple locking mechanism that allows only two states: locked or unlocked. It can lead to deadlocks if not managed properly.
- Exclusive Locking: A lock that prevents other transactions from accessing the locked resource. It is useful for write operations but can lead to contention.
- Shared Locking: Allows multiple transactions to read the same resource simultaneously, promoting concurrency for read operations.
- Deadlock Prevention: Techniques to prevent deadlocks by ensuring that transactions acquire locks in a consistent order.
- Two-Phase Locking: A concurrency control method that requires all locks to be acquired before any are released, ensuring serializability.