Database Management System

EG2201CT

Year: II
Part: II

Total: 6 hours /week
Lecture: 3 hours/week
Tutorial: 1 hours/week
Practical: hours/week
Lab: 2 hours/week

Course description:

This course covers the core principles and techniques required in the design and implementation of database systems. It consists of relational database systems RDBMS - the predominant system for business, scientific and engineering applications at present, Entity-Relational model, Normalization, Relational model, Relational algebra, and data access queries as well as an introduction to SQL. It also covers essential DBMS concepts such as: Transaction Processing, Concurrency Control and Recovery.

Course objectives:

The main objectives of this course are:

1. Explain the concepts of database and database management system.
2. Provide knowledge of database design using entity relationship diagram.
3. Perform on SQL statements, normalization, transaction processing, and database recovery.

Course Contents:

Theory

Unit 1. Introduction **[5 Hrs.]**

1.1. History, Database and its applications
1.2. Characteristics
1.3. Architecture
1.4. Data abstraction and Independence
1.5. Schemas and Instances
1.6. Classifications of DBMS
1.7. Introduction to DDL, DML, DCL

Unit 2. Data Models **[8 Hrs.]**

2.1. Introduction to Entity Relationship Model
2.2. Entities type
2.3. Entities set
2.4. Attributes and keys
2.5. Relationship types and sets
2.6. E-R diagrams

Unit 3. Normalization **[6 Hrs.]**

3.1. Importance of Normalization
3.2. Functional Dependencies
3.3. Integrity and Domain constraints
3.4. Normal forms (1NF, 2NF, 3NF, BCNF)

Unit 4. Relational Language **[8 Hrs.]**

4.1. Introduction to SQL
4.2. Features of SQL

4.3. Basic Retrieval queries
4.4. INSERT, UPDATE, DELETE queries
4.5. Join, Semi join and Sub queries
4.6. Views
4.7. Relational Algebra
    4.7.1. Select, Project
    4.7.2. Set Operations
    4.7.3. Cartesian Product
    4.7.4. Join

Unit 5. Query Processing **[6 Hrs.]**

5.1. Introduction to Query Processing
5.2. Query Cost estimation
5.3. Query Operations, Operator TREE
5.4. Evaluation of Expressions
5.5. Query Optimization
5.6. Performance Tuning

Unit 6. Transaction and Concurrency Control **[6 Hrs.]**

6.1. Introduction to Transaction
6.2. Serializability concept
6.3. Concurrent execution
6.4. Lock based Concurrency Control
6.5. 2PL and Strict 2PL
6.6. Timestamp concept

Unit 7. Recovery **[6 Hrs.]**

7.1. Failure Classifications
7.2. Recovery and Atomicity
7.3. IN PLACE and Out of Place Update
7.4. Log based Recovery
7.5. Shadow Paging
7.6. Local Recovery Manager
7.7. UNDO and REDO protocol

Practical: **[30 Hrs.]**

1. SQL Queries on CREATE, INSERT, DELETE, and UPDATE operations.
2. SQL query for SELECT operation.
3. SQL query for ALTER operations.
4. SQL queries on JOIN
5. SQL query using aggregate functions.
6. Apply SQL for specifying constraints.

Final written exam evaluation scheme			
Unit	**Title**	**Hours**	**Marks Distribution***
1	Introduction	5	8
2	Data Model	8	14
3	Normalization	6	11
4	Relational Language	8	14
5	Query Processing	6	11

6	Transaction and Concurrency Control	6	11
7	Recovery	6	11
	Total	**45**	**80**

* There may be minor deviation in marks distribution.

References:

1. Silberschatz, H.F. Korth, and S. Sudarshan (2010), Database System Concepts, 6th Edition, McGraw Hill
2. Ramez Elmasri and Shamkant B. Navathe (2010), Fundamentals of Database Systems, 6 th Edition, Pearson Addison Wesley
3. Raghu Ramakrishnan, and Johannes Gehrke (2007), Database Management Systems, 3rd Edition, McGraw-Hill
4. Jaffrey D. Ullman, Jennifer Widom; A First Course in Database Systems; Third Edition; Pearson Education Limited

Name : Dilbar Yadav
Clz: Narayani Model Secondary School
Bharatpur 10 Chitwan

Unit 1. Introduction

1.1 History, Database and its applications

Introduction:

A database is a structured collection of data that is organized in a way that allows for efficient storage, retrieval, and management of information.

It serves as a central repository for storing and organizing data, making it easier to access and manipulate that data as needed. Databases are an essential component of modern information systems and play an importance role in various applications, including business, science, research, and more.

History:

The history of database:

1. Pre-Computer Era: Before the advent of computers, data was stored in physical records, such as paper documents, ledgers, and index cards. Retrieving and managing this data was a manual and time-consuming process.
2. 1960s-1970s: The Emergence of Computerized Databases: With the growth of computer technology in the 1960s, the concept of computerized databases emerged. Early database management systems (DBMS) like CODASYL and IMS were developed to organize and store data efficiently.
3. 1970s-1980s: The Relational Database Model: In the 1970s, Edgar F. Codd introduced the relational database model, which revolutionized data management. Relational databases, like IBM's DB2 and Oracle, used tables with rows and columns to store structured data, making it easier to manage and query information.

4. 1980s-1990s: Commercial Database Systems and SQL: The 1980s and 1990s saw the rise of commercial relational database management systems (RDBMS) like Microsoft SQL Server and MySQL. The Structured Query Language (SQL) became the standard language for interacting with relational databases.
5. 2000s-Present: The Era of Big Data and NoSQL: The 21st century brought about the era of Big Data, where organizations had to manage vast amounts of unstructured and semi-structured data. NoSQL databases like MongoDB and Cassandra emerged to handle these new data types and the need for scalable and flexible data storage.

Database and its applications

Databases are widely used in various applications across different industries due to their ability to efficiently store, manage, and retrieve data. Here are eight common applications of databases:

1. Business Data Management:
   Databases are central to businesses for managing customer information, sales data, inventory, and financial records. Enterprise Resource Planning (ERP) systems and Customer Relationship Management (CRM) systems rely on databases to streamline operations.
2. E-commerce:
   Online retailers use databases to store product catalogs, customer profiles, order histories, and transaction records. Databases enable personalized shopping experiences and efficient order processing.
3. Healthcare:
   Electronic Health Records (EHR) systems store patient medical histories, test results, and treatment plans in databases. Healthcare professionals rely on these systems to provide quality patient care.
4. Education:
   Educational institutions use databases to manage student enrollment, grades, course schedules, and faculty information. Learning Management Systems (LMS) also rely on databases to deliver course content and track student progress.
5. Finance:
   Banks and financial institutions use databases to manage customer accounts, process transactions, and detect fraudulent activities. Stock exchanges and trading platforms rely on high-performance databases for real-time trading.
6. Telecommunications:

Telecom providers use databases to store subscriber information, call records, and network configuration data. This data is crucial for billing, network optimization, and customer support.

7. Government and Public Services:
   Government agencies use databases for various purposes, such as managing tax records, issuing passports, and maintaining census data. Geographic Information Systems (GIS) databases are used for mapping and urban planning.
8. Research and Science:
   In scientific research, databases store experimental data, research findings, and genomic information. Researchers and scientists can query these databases to analyze and share their work.
9. Social Media:
   Social media platforms rely heavily on databases to store user profiles, posts, comments, and multimedia content. Databases enable rapid retrieval of personalized content and support real-time interactions.
10. Airlines and Transportation:
    Airlines and transportation companies use databases for booking reservations, managing flight schedules, and tracking cargo. Databases play a crucial role in ensuring efficient logistics and passenger management.

1.2 Characteristics of database

The characteristics of a database system help define its functionality and purpose.

- Data Organization:
  Database's structure and organize data for efficient storage and retrieval.
- Data Integrity:
  Databases maintain data accuracy and consistency through constraints.
- Data Retrieval:
  Databases support querying to extract specific information.
- Concurrency Control:
  They manage simultaneous data access to maintain consistency.
- Security:
  Databases control access to data to ensure confidentiality and integrity.
- Scalability:
  They can grow vertically or horizontally to handle increased data and users.
- Redundancy and Backup:
  Databases ensure data availability and durability through redundancy and backups.
- Indexing:
  Indexes are used for quick data retrieval, improving query performance.

1.3 Database Architecture

What is Database Architecture?

A Database Architecture is a representation of DBMS design. It helps to design, develop, implement, and maintain the database management system. A DBMS architecture allows dividing the database system into individual components that can be independently modified, changed, replaced, and altered. It also helps to understand the components of a database.

A Database stores critical information and helps access data quickly and securely. Therefore, selecting the correct Architecture of DBMS helps in easy and efficient data management.

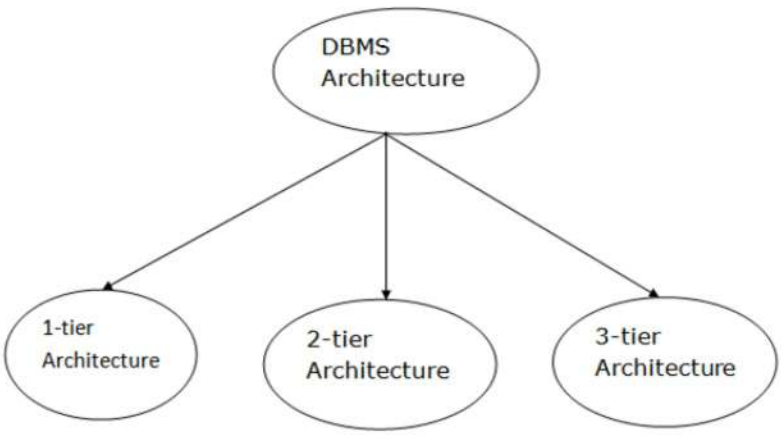
DBMS Architecture

The DBMS design depends upon its architecture. The basic client/server architecture is used to deal with a large number of PCs, web servers, database servers and other components that are connected with networks.

The client/server architecture consists of many PCs and a workstation which are connected via the network.

DBMS architecture depends upon how users are connected to the database to get their request done.

Types of DBMS Architecture



Database architecture can be seen as a single tier or multi-tier. But logically, database architecture is of two types like: 2-tier architecture and 3-tier architecture.

1-Tier Architecture

In this architecture, the database is directly available to the user. It means the user can directly sit on the DBMS and uses it.

Any changes done here will directly be done on the database itself. It doesn't provide a handy tool for end users.

The 1-Tier architecture is used for development of the local application, where programmers can directly communicate with the database for the quick response.

2-Tier Architecture

The 2-Tier architecture is same as basic client-server. In the two-tier architecture, applications on the client end can directly communicate with the database at the server side. For this interaction, API's like: ODBC, JDBC are used.

The user interfaces and application programs are run on the client-side.

The server side is responsible to provide the functionalities like: query processing and transaction management.

To communicate with the DBMS, client-side application establishes a connection with the server side.

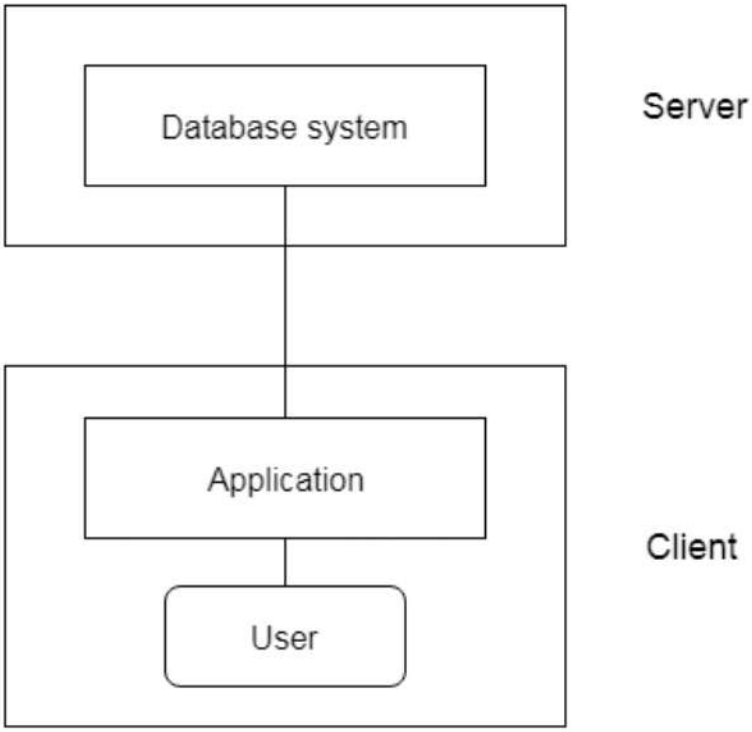


Fig: 2-tier Architecture

3-Tier Architecture

The 3-Tier architecture contains another layer between the client and server. In this architecture, client can't directly communicate with the server.

The application on the client-end interacts with an application server which further communicates with the database system.

End user has no idea about the existence of the database beyond the application server. The database also has no idea about any other user beyond the application.

The 3-Tier architecture is used in case of large web application.

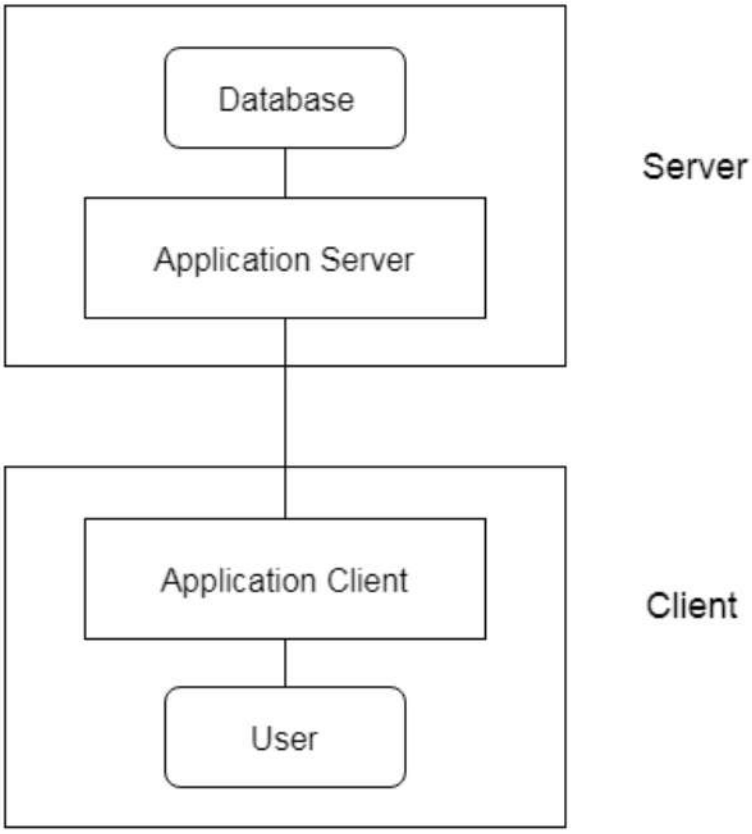
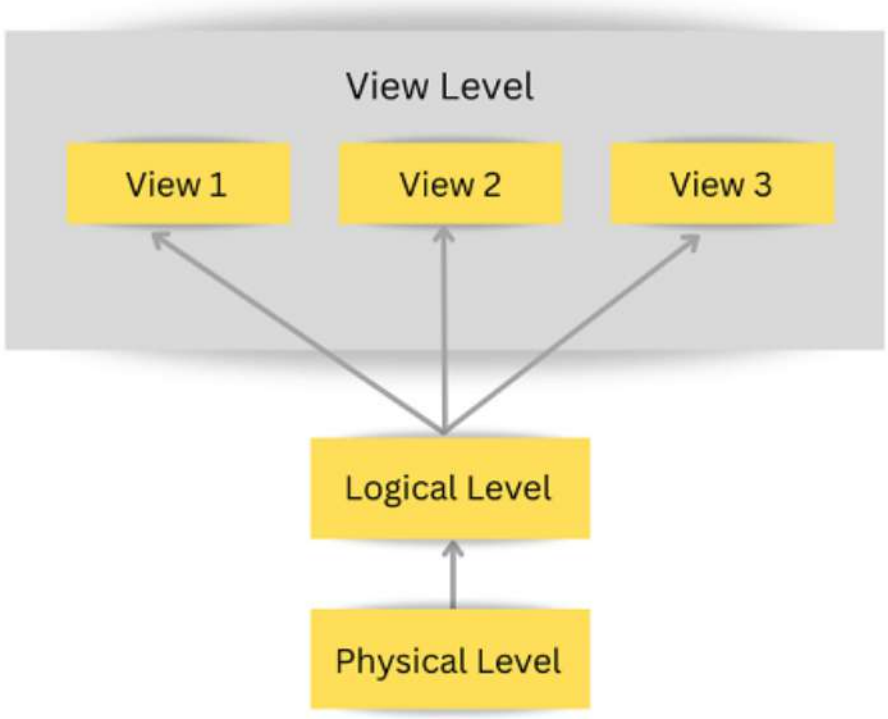


Fig: 3-tier Architecture

1.4 Data abstraction and Independence

- **Data Abstraction:**

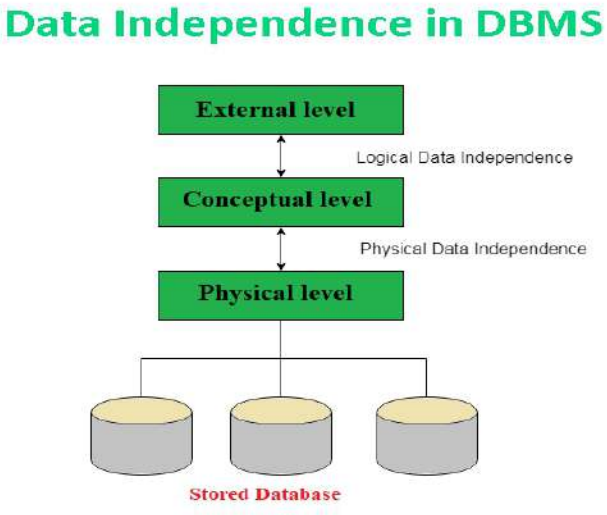
  Data abstraction simplifies complex data by presenting only essential details, allowing easier interaction with and understanding of data.



Levels of Data Abstraction in DBMS

- **Data Independence:**

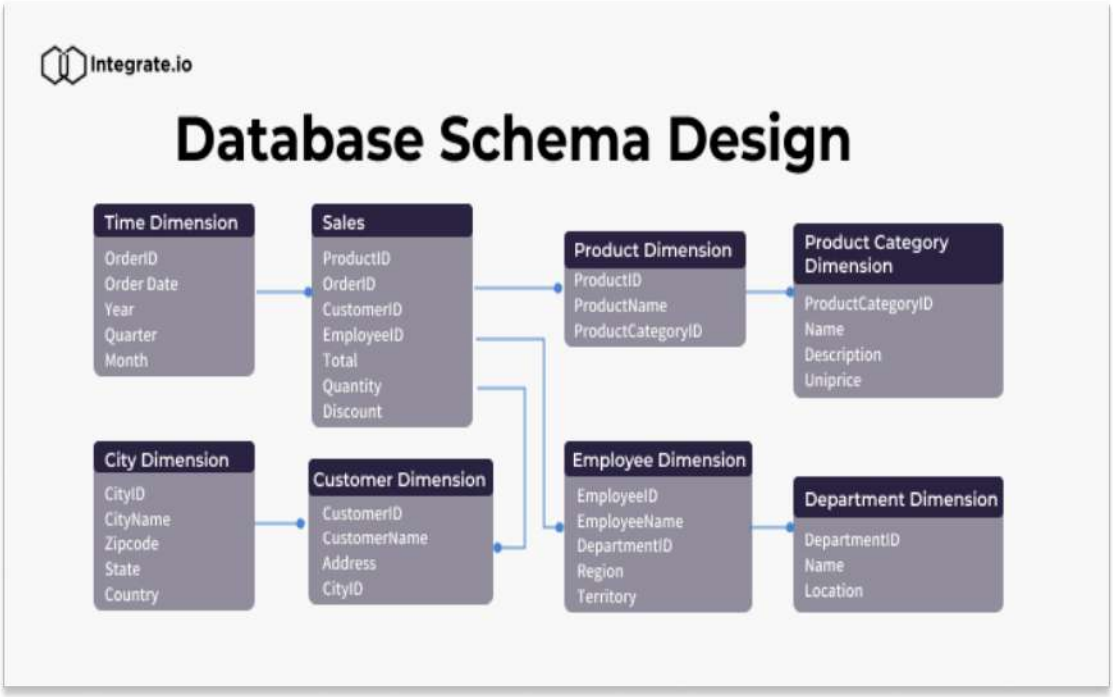
  Data independence separates the logical view of data from its physical storage, enabling changes to the physical structure without affecting applications or queries that use the data.



1.5 Schemas and Instances

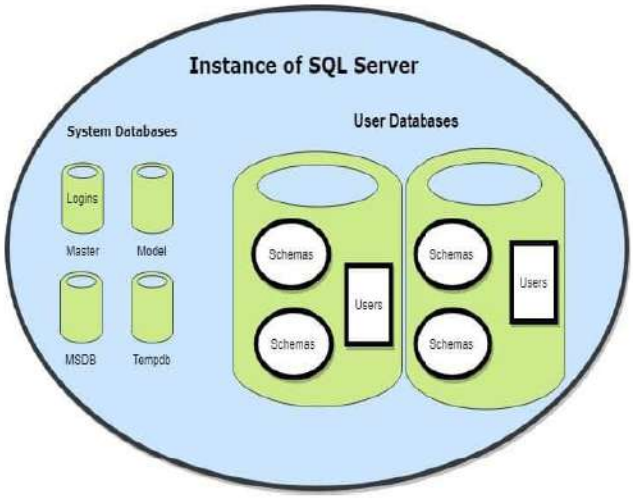
- **Database Schema:**
  Definition: A database schema is a blueprint or logical design that defines the structure, organization, and relationships of data within a database. It specifies the tables, columns, data types, constraints, and relationships between tables.

  Purpose: The schema defines the overall structure and rules governing the data stored in the database. It acts as a framework for organizing and maintaining data integrity.



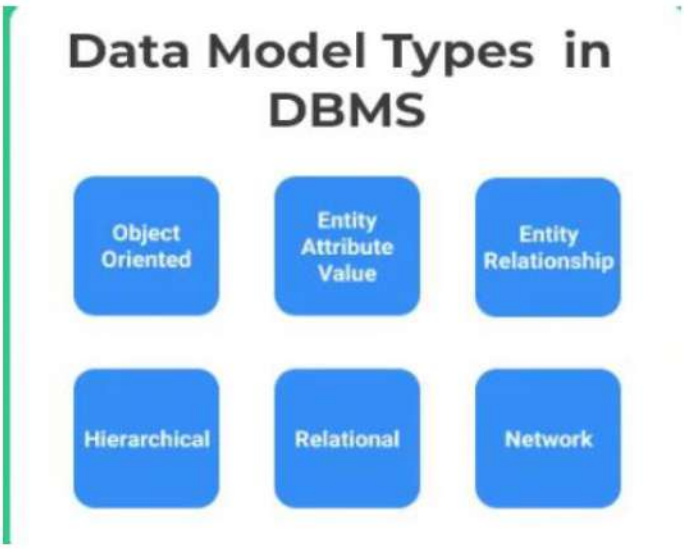
- **Database Instance:**
  Definition: A database instance is a specific snapshot or realization of a database schema at a particular point in time. It represents the actual data stored in the database, including the records, values, and relationships.

  Purpose: The instance contains the real data that users and applications interact with. It can change over time as data is added, modified, or deleted, but it adheres to the schema's structure and constraints.



1.6 Classifications and DBMS

- **Based on Data Model:**
  -Relational Databases: Store data in tables with rows and columns, using the relational model. Examples include Oracle, MySQL, and SQL Server.
  -NoSQL Databases: Designed for unstructured or semi-structured data and offer flexible schemas. Types include document stores (e.g., MongoDB), key-value stores (e.g., Redis), column-family stores (e.g., Cassandra), and graph databases (e.g., Neo4j).



- **Based on Deployment:**
  -On-Premises Databases: Installed and operated on local servers or data centers.
  -Cloud Databases: Hosted and managed in cloud platforms like AWS, Azure, or Google Cloud.
  -Hybrid Databases: Combine on-premises and cloud-based components for data storage and processing.
- **Based on Use Case:**
  -Operational Databases: Designed for day-to-day transactional activities, such as customer orders and inventory management.
  -Analytical Databases: Optimized for complex queries and data analysis, often used for business intelligence and reporting.
- **Based on Access Method:**
  -Centralized Databases: Single, centralized database server with multiple clients accessing it.
  -Distributed Databases: Data is distributed across multiple servers or locations, often for scalability or fault tolerance.
  -Peer-to-Peer Databases: Each node in the network can act as both a client and a server, sharing data and resources.
- **Based on Licensing and Cost:**
  -Open-Source Databases: Free and open-source database systems, such as PostgreSQL or MySQL.
  -Commercial Databases: Proprietary database systems that require licensing, like Oracle Database or Microsoft SQL Server.

1.7 Introduction to DDL, DML and DCL

- **DDL (Data Definition Language): (CREATE, ALTER, DROP)**

Purpose: Used for defining, modifying, and managing the structure of database objects, such as tables, indexes, and constraints.

Examples:

- CREATE TABLE to create a new table.
- ALTER TABLE to modify the structure of an existing table.
- DROP TABLE to delete a table.

Effect: Affects the database's structure or schema, not the data itself.

Scope: Typically operates at the database or schema level, defining or modifying the structure of multiple objects.

DDL (Data Definition Language): (CREATE, ALTER, DROP)

Example:

1. **Create Table**

```
CREATE TABLE employees (
employee_id INT PRIMARY KEY,
first_name VARCHAR(50),
last_name VARCHAR(50),
hire_date DATE
);
```

2. **Alter Table**

```
ALTER TABLE table_name
ADD  Column_name datatype;
Example :
ALTER TABLE ABC_COMPANY
ADD department_id INT;
```

3. Drop Table

```
DROP TABLE employees;
```

- **DML (Data Manipulation Language): (INSERT, SELECT, UPDATE, DELETE)**

Purpose: Used for manipulating data in the database, such as adding, retrieving, updating, or deleting data.

Examples:

- INSERT INTO to add data.
- SELECT to retrieve data.
- UPDATE to modify data.

- DELETE to remove data.

Effect: Affects the data stored in the database tables.

Scope: Operates at the row level, affecting specific data rows

DML (Data Manipulation Language): (SELECT, INSERT, UPDATE, DELETE)

Example:

1. SELECT

```
SELECT column1, column2
FROM table
WHERE condition;
```

2. INSERT

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

3. UPDATE

```
UPDATE table_name
SET column1 = value1
WHERE condition;
```

```
update nmss
set id=56
where id=34;
```

4. Delete

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

- **DCL (Data Control Language): (GRANT, REVOKE)**

Purpose: Used for controlling access and permissions on database objects, as well as managing security.

Examples:

- GRANT to give specific permissions to a user.
- REVOKE to remove permissions.

Effect: Modifies access permissions and security settings.

Scope: Operates at the database or object level, affecting multiple tables or permissions.

DCL (Data Control Language):

Example:
Grant

```
GRANT SELECT, INSERT ON table_name TO user_name;
```

Revoke

REVOKE DELETE ON table_name FROM user_name;

==========

Unit 2. Data Models

2.1 Introduction to entity relationship model (E-R Model)
2.2 Entities types
2.3 Entities Set
2.4 Attributes and Keys
2.5 Relationship types and sets
2.6 E-R Diagram

2.1 Introduction to entity relationship model (E-R Model)

The Entity Relational Model is a model for identifying entities to be represented in the database and representation of how those entities are related.
The ER data model specifies enterprise schema that represents the overall logical structure of a database graphically.
The Entity Relationship Diagram explains the relationship among the entities present in the database. ER models are used to model real-world objects like a person, a car, or a company and the relation between these real-world objects. In short, the ER Diagram is the structural format of the database.

Why Use ER Diagrams In DBMS?

- ER diagrams are used to represent the E-R model in a database, which makes them easy to be converted into relations (tables).
- ER diagrams provide the purpose of real-world modeling of objects which makes them intently useful.
- ER diagrams require no technical knowledge and no hardware support.
- These diagrams are very easy to understand and easy to create even for a naive user.
- It gives a standard solution for visualizing the data logically.

Symbols Used in ER Model

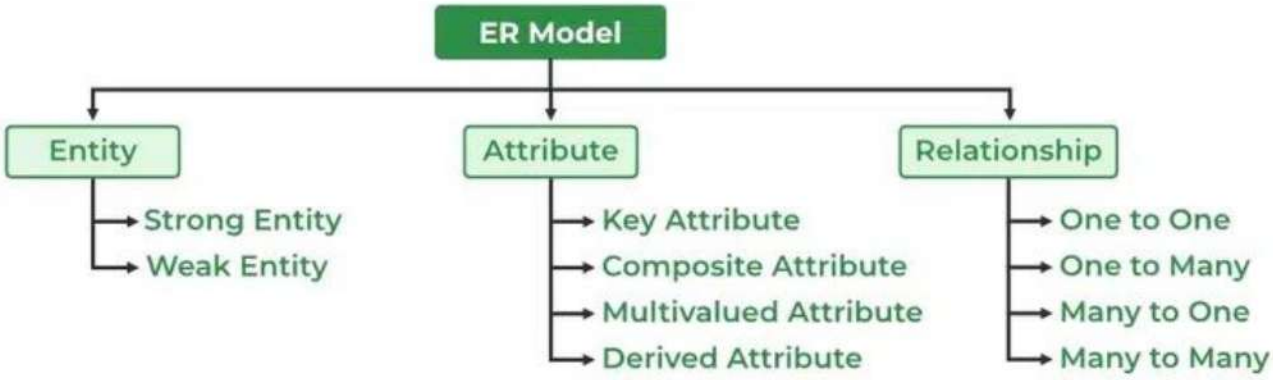
ER Model is used to model the logical view of the system from a data perspective which consists of these symbols:

- **Rectangles:** Rectangles represent Entities in the ER Model.
- **Ellipses:** Ellipses represent Attributes in the ER Model.
- **Diamond:** Diamonds represent Relationships among Entities.
- **Lines:** Lines represent attributes to entities and entity sets with other relationship types.
- **Double Ellipse:** Double Ellipses represent Multi-Valued Attributes.
- **Double Rectangle:** Double Rectangle represents a Weak Entity.

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

Components of ER Diagram

ER Model consists of Entities, Attributes, and Relationships among Entities in a Database System.



Components of ER Diagram

2.2 Entities types

Entity

An Entity may be an object with a physical existence, for example a particular person, car, house, or employee – or it may be an object with a conceptual existence – a company, a job, or a university course.

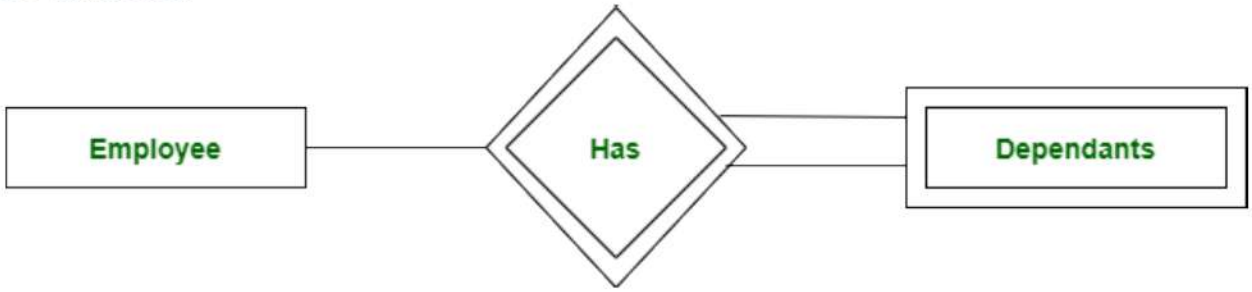
1.Strong Entity

A Strong Entity is a type of entity that has a key Attribute. Strong Entity does not depend on other Entity in the Schema. It has a primary key, that helps in identifying it uniquely, and it is represented by a rectangle. These are called Strong Entity Types.

2. Weak Entity

An Entity type has a key attribute that uniquely identifies each entity in the entity set. But some entity type exists for which key attributes can't be defined. These are called Weak Entity types.

For Example, A company may store the information of dependents (Parents, Children, Spouse) of an Employee. But the dependents don't have existed without the employee. So Dependent will be a **Weak Entity Type** and Employee will be Identifying Entity type for Dependent, which means it is **Strong Entity Type**.
A weak entity type is represented by a Double Rectangle. The participation of weak entity types is always total. The relationship between the weak entity type and its identifying strong entity type is called identifying relationship and it is represented by a double diamond.

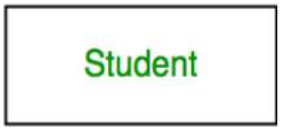


Strong Entity and Weak Entity

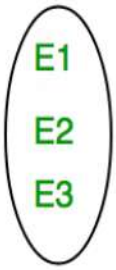
2.3 Entities Set

Entity Set:

An Entity is an object of Entity Type and a set of all entities is called an entity set. For Example, E1 is an entity having Entity Type Student and the set of all students is called Entity Set. In ER diagram, Entity Type is represented as:



Entity Type



Entity Set

2.4 Attributes and Keys

Attributes

Attributes are the properties that define the entity type.
For example, Roll_No, Name, DOB, Age, Address, and Mobile_No are the attributes that define entity type Student. In ER diagram, the attribute is represented by an oval.

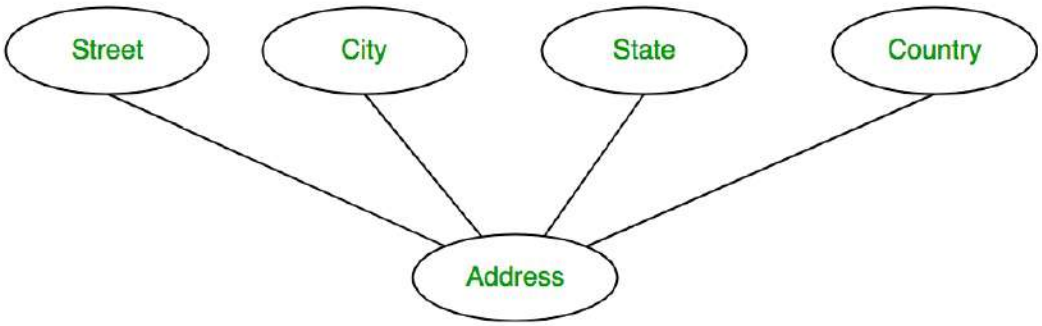
1. Primary Key Attribute

The attribute which **uniquely identifies each entity** in the entity set is called the key attribute. For example, Roll_No will be unique for each student. In ER diagram, the key attribute is represented by an oval with underlying lines.

Key Attribute

2. Composite key Attribute

An attribute **composed of many other attributes** is called a composite attribute. For example, the Address attribute of the student Entity type consists of Street, City, State, and Country. In ER diagram, the composite attribute is represented by an oval comprising of ovals.



Composite Attribute

3. **Multivalued Attribute**

An attribute consisting of more than one value for a given entity. For example, Phone_No (can be more than one for a given student). In ER diagram, a multivalued attribute is represented by a double oval.

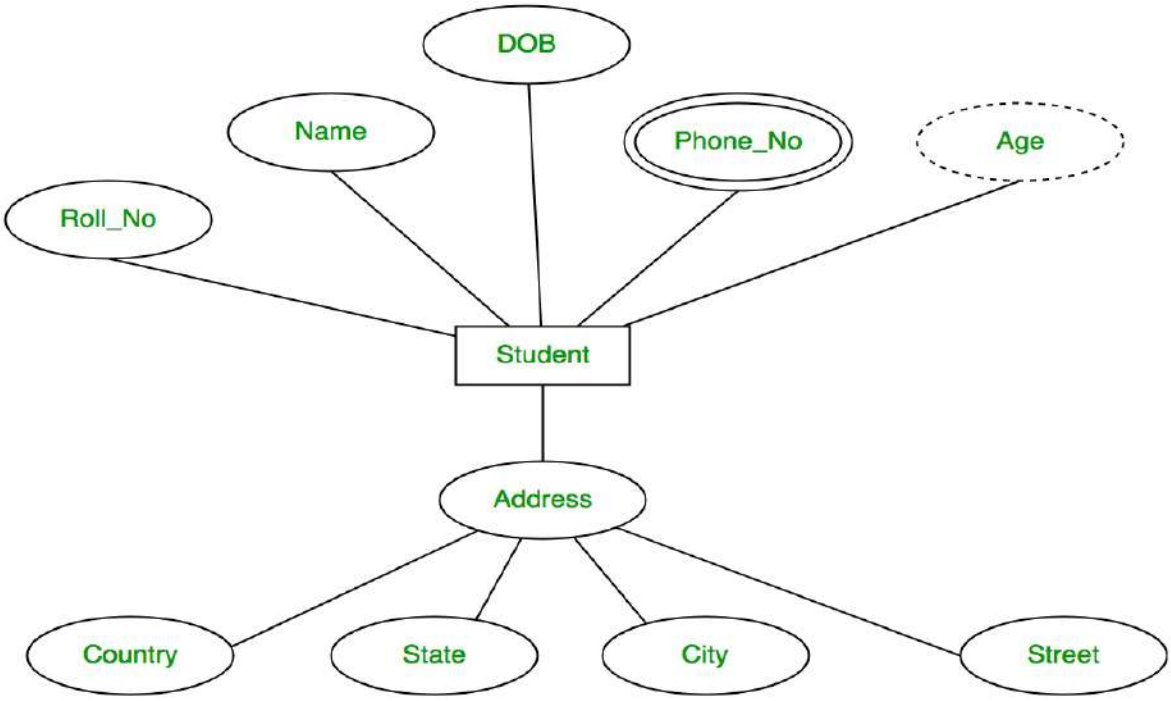
Multivalued Attribute

4. **Derived Attribute**

An attribute that can be derived from other attributes of the entity type is known as a derived attribute. e.g.; Age (can be derived from DOB). In ER diagram, the derived attribute is represented by a dashed oval.

Derived Attribute

The Complete Entity Type Student with its Attributes can be represented as:



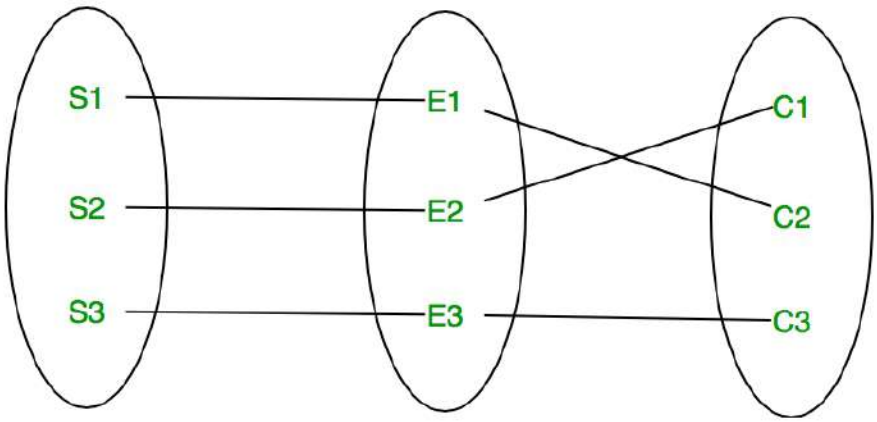
Entity and Attributes

2.5 Relationship types and Sets

A Relationship Type represents the association between entity types. For example, 'Enrolled in' is a relationship type that exists between entity type Student and Course. In ER diagram, the relationship type is represented by a diamond and connecting the entities with lines.

Entity-Relationship Set

A set of relationships of the same type is known as a relationship set. The following relationship set depicts S1 as enrolled in C2, S2 as enrolled in C1, and S3 as registered in C3.

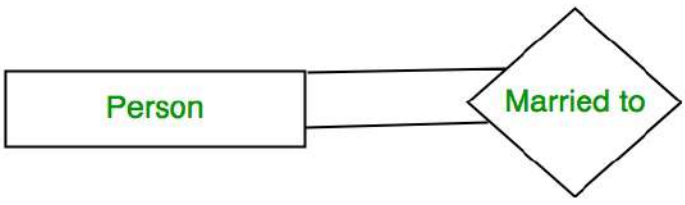


Relationship Set

Degree of a Relationship Set

The number of different entity sets participating in a relationship set is called the **degree of a relationship set.**

1. Unary Relationship: When there is only ONE entity set participating in a relation, the relationship is called a unary relationship. For example, one person is married to only one person.



Unary Relationship

2. Binary Relationship: When there are TWO entities set participating in a relationship, the relationship is called a binary relationship. For example, a Student is enrolled in a Course.

Binary Relationship

3. n-ary Relationship: When there are n entities set participating in a relation, the relationship is called an n-ary relationship.

Cardinality

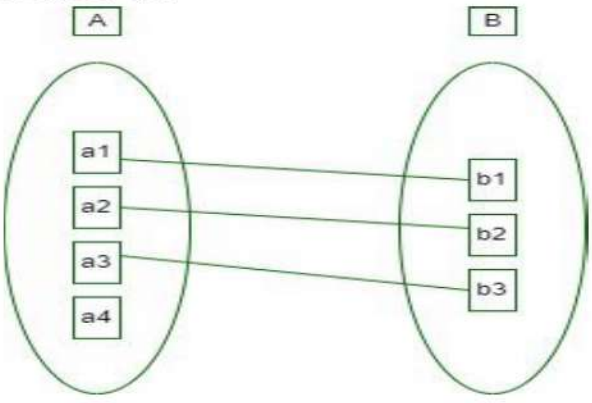
The number of times an entity of an entity set participates in a relationship set is known as cardinality. Cardinality can be of different types:

1. One-to-One: When each entity in each entity set can take part only once in the relationship, the cardinality is one-to-one. Let us assume that a male can marry one female and a female can marry one male. So the relationship will be one-to-one.

the total number of tables that can be used in this is 2.

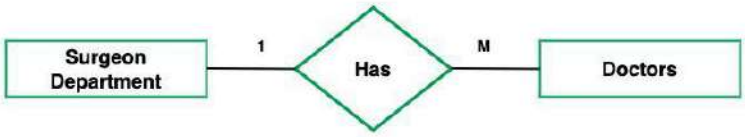
one to one cardinality

Using Sets, it can be represented as:



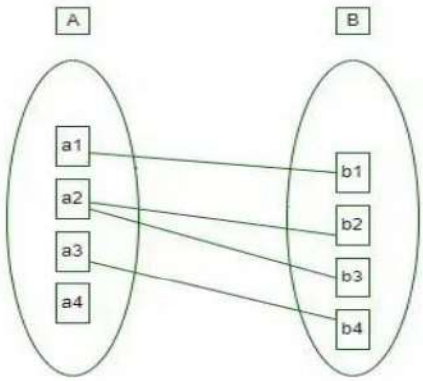
Set Representation of One-to-One

2. One-to-Many: In one-to-many mapping as well where each entity can be related to more than one relationship and the total number of tables that can be used in this is 2. Let us assume that one surgeon deparment can accomodate many doctors. So the Cardinality will be 1 to M. It means one deparment has many Doctors.



one to many cardinality

Using sets, one-to-many cardinality can be represented as:

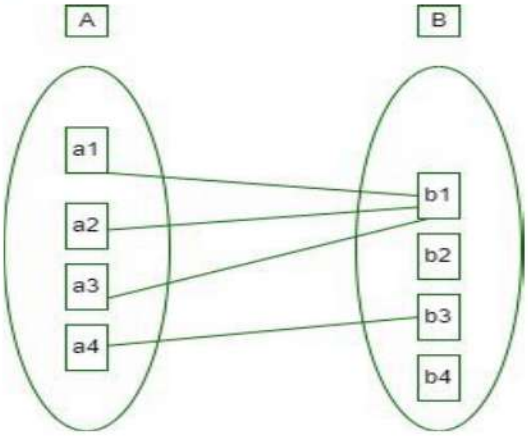


Set Representation of One-to-Many

3. Many-to-One: When entities in one entity set can take part only once in the relationship set and entities in other entity sets can take part more than once in the relationship set, cardinality is many to one. Let us assume that a student can take only one course but one course can be taken by many students. So the cardinality will be n to 1. It means that for one course there can be n students but for one student, there will be only one course.
The total number of tables that can be used in this is 3.

many to one cardinality

Using Sets, it can be represented as:



Set Representation of Many-to-One

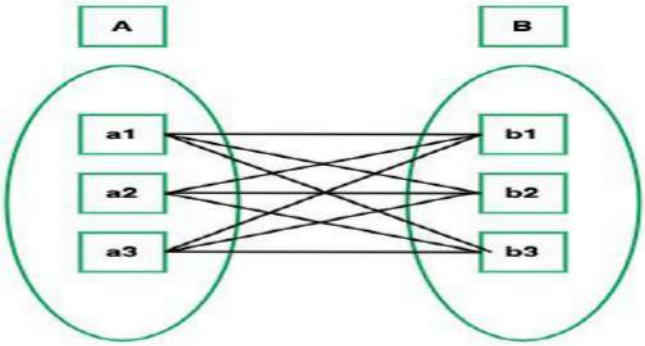
In this case, each student is taking only 1 course but 1 course has been taken by many students.

4. Many-to-Many: When entities in all entity sets can take part more than once in the relationship cardinality is many to many. Let us assume that a student can take more than one course and one course can be taken by many students. So the relationship will be many to many.

the total number of tables that can be used in this is 3.

many to many cardinality

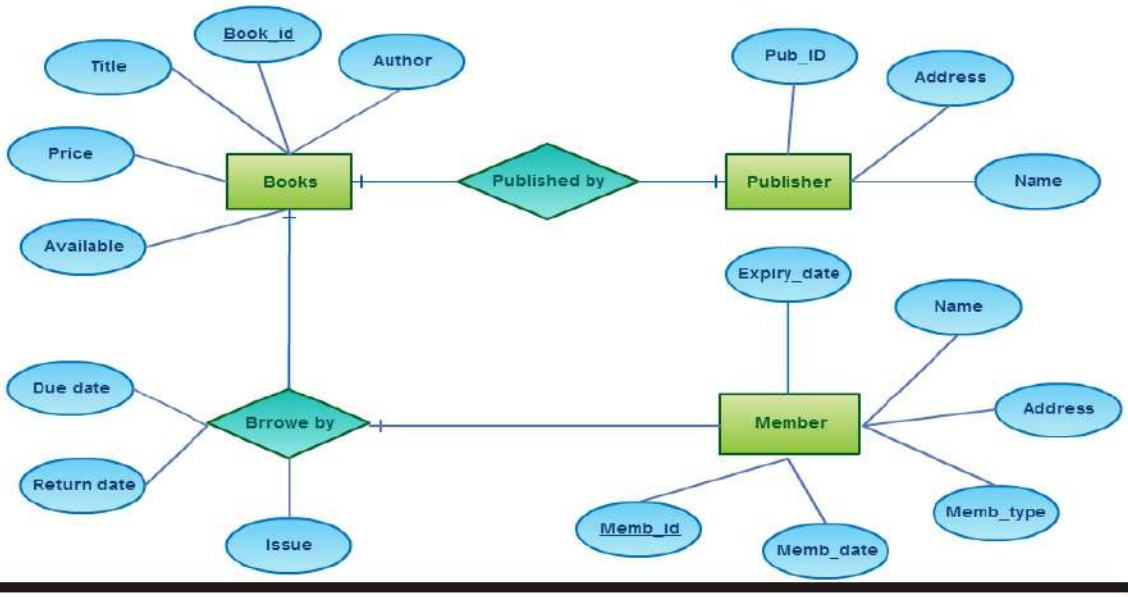
Using Sets, it can be represented as:

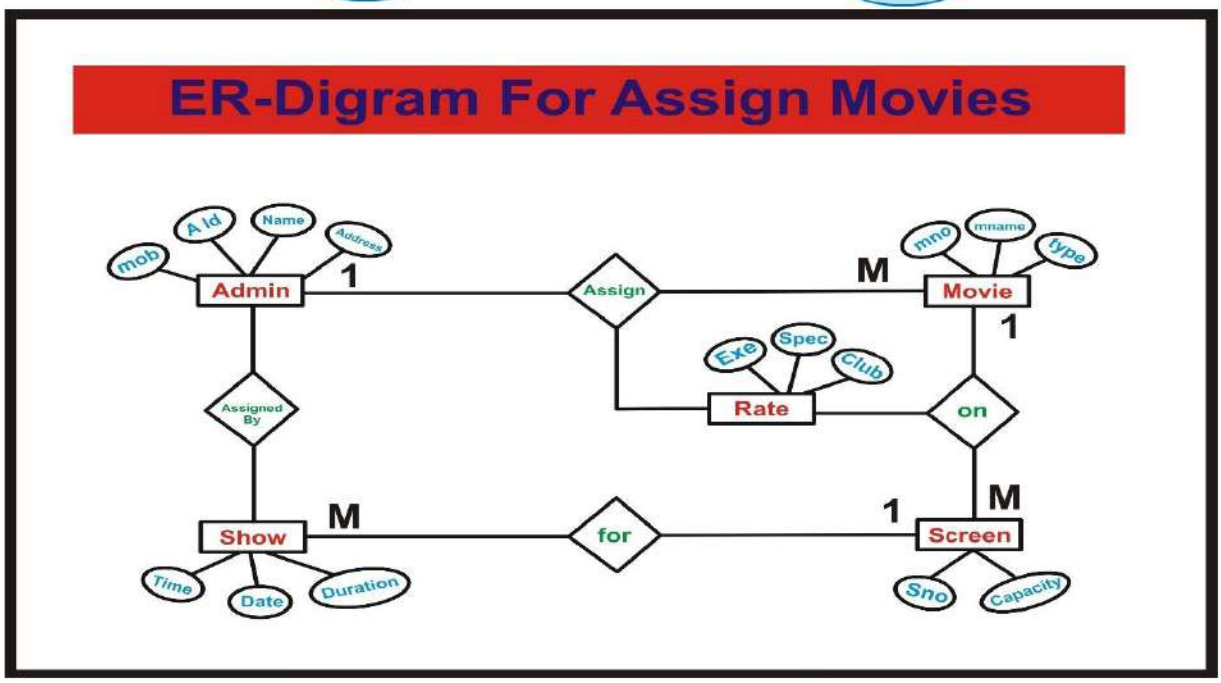


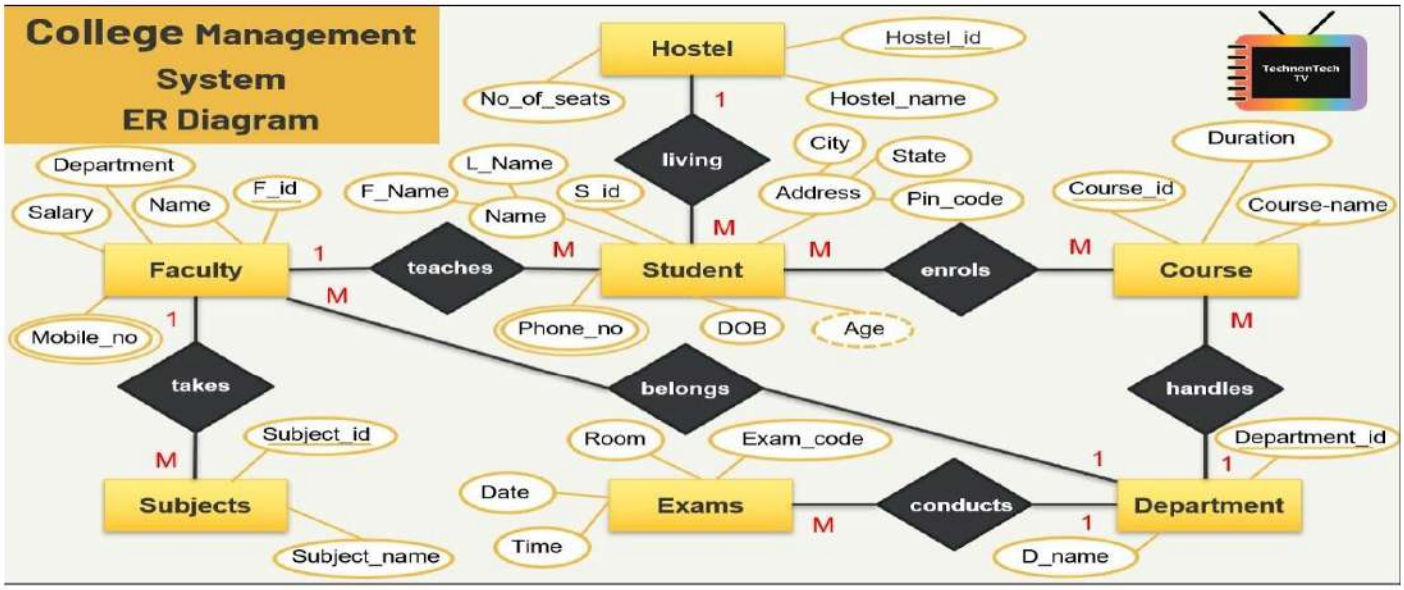
Many-to-Many Set Representation

2.6 E-R Diagrams

E-R Diagram of Library Management System







==========

Unit -3. Normalization

3.1 Importance of Normalizations
3.2 Functional Dependences
3.3 Integrity and Domain Constraints
3.4 Normal Forms (1 NF, 2NF, 3NF and BCNF)

Definition of Normalizations:

Normalization in a database is the process of efficiently organizing data to minimize redundancy and dependency. It involves structuring tables and relationships between them in a way that reduces data duplication and ensures that data is logically organized. It is a process of decomposing a large and complex table into simpler and separate form that's why it can be easy to create multiple connection in database. Which is more manageable form.

Normalization is typically achieved through a series of rules and forms, such as First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), and Boyce-Codd Normal Form (BCNF).

In simpler terms, normalization is like arranging information in a database so that it's organized, easy to understand, and efficient to manage, reducing the risk of errors and inconsistencies in the data.

3.1 Importance of Normalizations

Importance of Normalization are given below:

- **Reduces Data Redundancy:** Normalization eliminates duplicate data by organizing it efficiently. This reduces storage space and ensures that each piece of information is stored only once.
- **Data Efficiency:** By organizing data more efficiently, normalization reduces data redundancy. You store each piece of information only once, saving storage space.
- **Data Integrity:** Normalization ensures that data is accurate and consistent. It helps prevent errors and inconsistencies by structuring data in a way that minimizes the risk of conflicting information.
- **Data Consistency:** When data is organized following normalization principles, it becomes more consistent across the entire database. This consistency is essential for reliable data retrieval and analysis.

- **Update Anomalies Prevention**: Normalization helps prevent anomalies that may arise during data updates. It ensures that modifying or adding data won't lead to unexpected issues or errors.
- **Improves Query Performance:** Well-normalized databases often lead to better query performance. Retrieving and manipulating data is more efficient when the database is organized in a structured manner.

3.2 Functional Dependences

Functional dependencies are a fundamental concept in database management and normalization. They describe the relationships between attributes (columns) in a relational database. A functional dependency occurs when the value of one attribute uniquely determines the value of another attribute in the same table.

In a functional dependency A → B:

- Attribute B is functionally dependent on attribute A.
- For every unique value of A, there is exactly one corresponding value of B.

For example, consider a table with the attributes EmployeeID, FirstName, and Department:

EmployeeID	FirstName	Department
1	Alice	HR
2	Bob	IT
3	Carol	HR

In this table:

- EmployeeID → FirstName: Each employee ID uniquely determines the first name of the employee.
- EmployeeID → Department: Each employee ID uniquely determines the department in which the employee works.
- Department (alone) does not determine any other attribute uniquely because multiple employees can belong to the same department.

Functional dependencies play an importance role in the normalization process. By understanding these dependencies, one can organize the database tables to minimize redundancy and improve data integrity. This is achieved by breaking down tables into smaller, well-organized structures based on functional dependencies.

3.3 Integrity and Domain Constraints

Integrity Constraints and Domain Constraints are importance concepts in database management to ensure the accuracy, consistency, and reliability of data.

Integrity Constraints:

1. Entity Integrity Constraint:

- Ensures that the primary key of a table uniquely identifies each record, preventing duplicate or null entries.

Example:

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

2. Referential Integrity Constraint:

Maintains consistency between tables by ensuring that foreign key values in one table match primary key values in another.

Example:

```
CREATE TABLE Departments (
DepartmentID INT PRIMARY KEY,
DepartmentName VARCHAR(50)
);

CREATE TABLE Employees (
EmployeeID INT PRIMARY KEY,
FirstName VARCHAR(50),
LastName VARCHAR(50),
DepartmentID INT,
FOREIGN KEY (DepartmentID) REFERENCES
Departments(DepartmentID)
);
```

3. Check Constraint:

Imposes a condition on data entered into a column, ensuring it meets specific criteria.

Example:

```
CREATE TABLE Students (
StudentID INT PRIMARY KEY,
FirstName VARCHAR(50),
LastName VARCHAR(50),
Age INT CHECK (Age >= 18)
 );
```

Domain Constraints:

1. Data Type Constraint:

- Defines the type of data allowed in a column (e.g., INTEGER, VARCHAR, DATE).
  Example:

```
CREATE TABLE Products (
ProductID INT PRIMARY KEY,
ProductName VARCHAR(100),
Price DECIMAL(10, 2)
);
```

2. NOT NULL Constraint:

- Ensures that a column cannot have a NULL value, meaning it must contain some data.
  Example:

```
CREATE TABLE Customers (
CustomerID INT PRIMARY KEY,
FirstName VARCHAR(50) NOT NULL,
LastName VARCHAR(50) NOT NULL
);
```

3. Unique Constraint:

- Guarantees that each value in a column is unique across the table.
  Example:

```
CREATE TABLE Employees (
EmployeeID INT PRIMARY KEY,
SSN VARCHAR(9) UNIQUE,
FirstName VARCHAR(50),
LastName VARCHAR(50)
);
```

In summary,

Integrity constraints maintain the relationships between tables, ensuring data consistency,

while domain constraints define the acceptable data types and values for columns, contributing to data accuracy and reliability in a database.

3.4 Normal Forms (1 NF, 2NF, 3NF and BCNF)

Starting with normal form	Convert to normal form	Abbreviation	Use the rule
Un-normalized data	First	1NF	remove repeating groups of data within a single tuple into new tuples with only one data value for each attribute
First	Second	2NF	Extract non-key partial dependencies (items not fully functionally dependent on the key) into a separate relation
Second	Third	3NF	remove transitive dependencies into a separate relation
Third	Boyce-Codd	BCNF	remove overlapping keys by identifying a single primary key and holding other values in a separate relation
Third Or Boyce-Codd	Fourth	4NF	remove multi-valued dependencies by extracting independent data into a separate relation
Fourth	Fifth	5NF	remove join dependencies caused by interdependent data

1. First Normal Form (1NF):

Features:

- Atomic Values: All columns must contain atomic (indivisible) values.
- No Repeating Groups: Avoids storing lists or sets within a single cell.

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721, 9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 1

↓ Conversion to first normal form

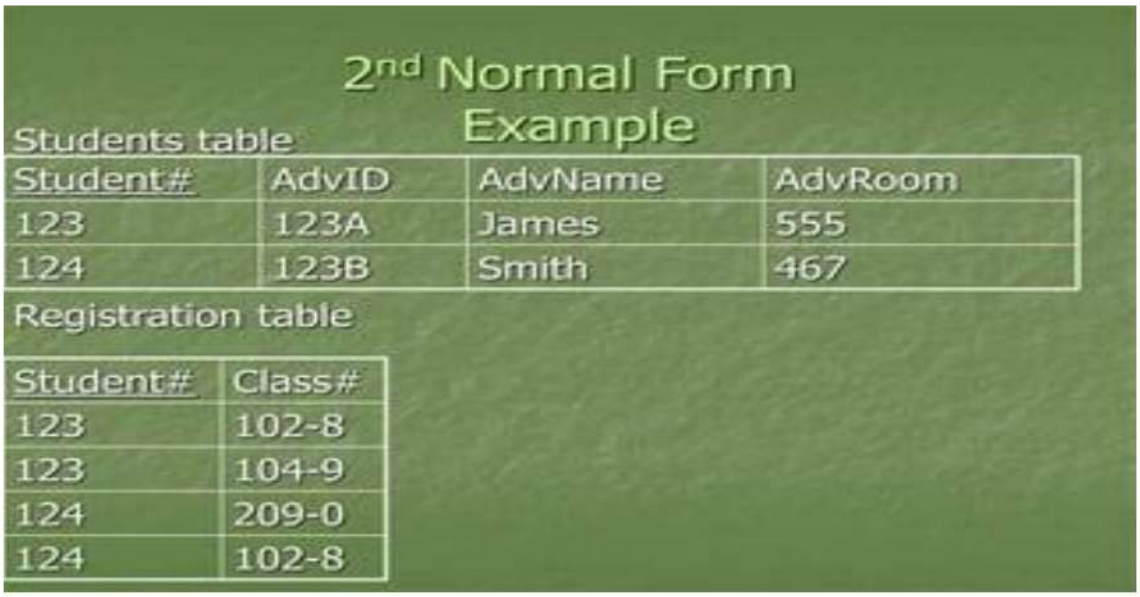
STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY
1	RAM	9716271721	HARYANA	
1	RAM	9871717178	HARYANA	INDIA
2	RAM	9898297281	PUNJAB	INDIA
3	SURESH		PUNJAB	INDIA

Table 2

2. Second Normal Form (2NF):

Features:

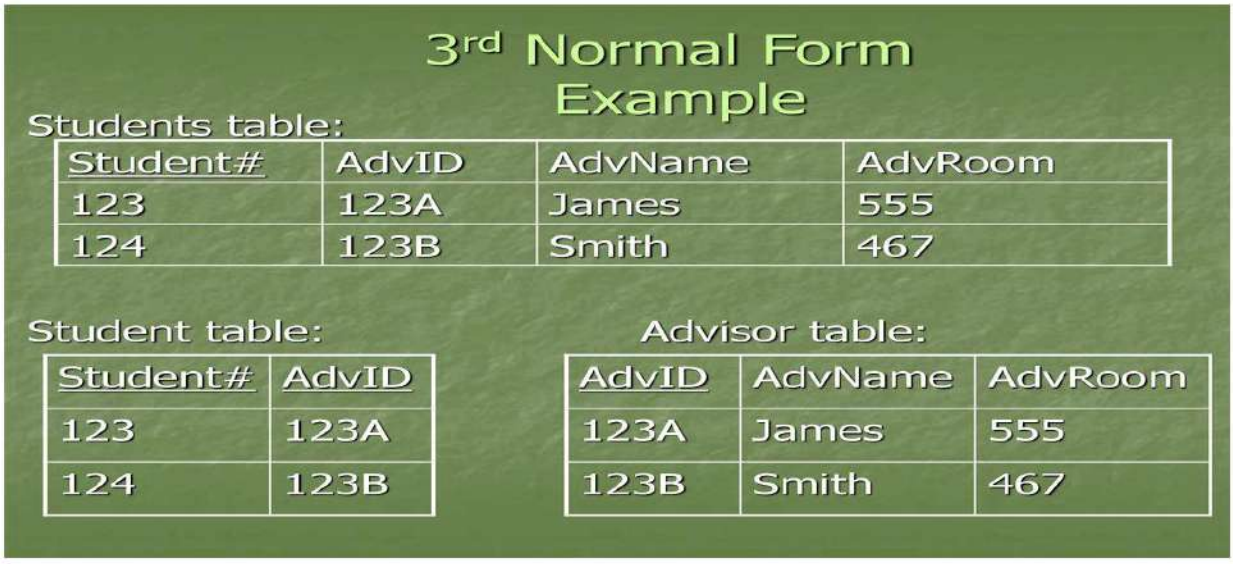
- Must be in 1NF: Ensures atomic values.
- No Partial Dependencies: Each non-prime attribute must be fully functionally dependent on the primary key.
- Eliminates Redundancy: Reduces redundancy by separating data into appropriate tables.



3. Third Normal Form (3NF):

Features:

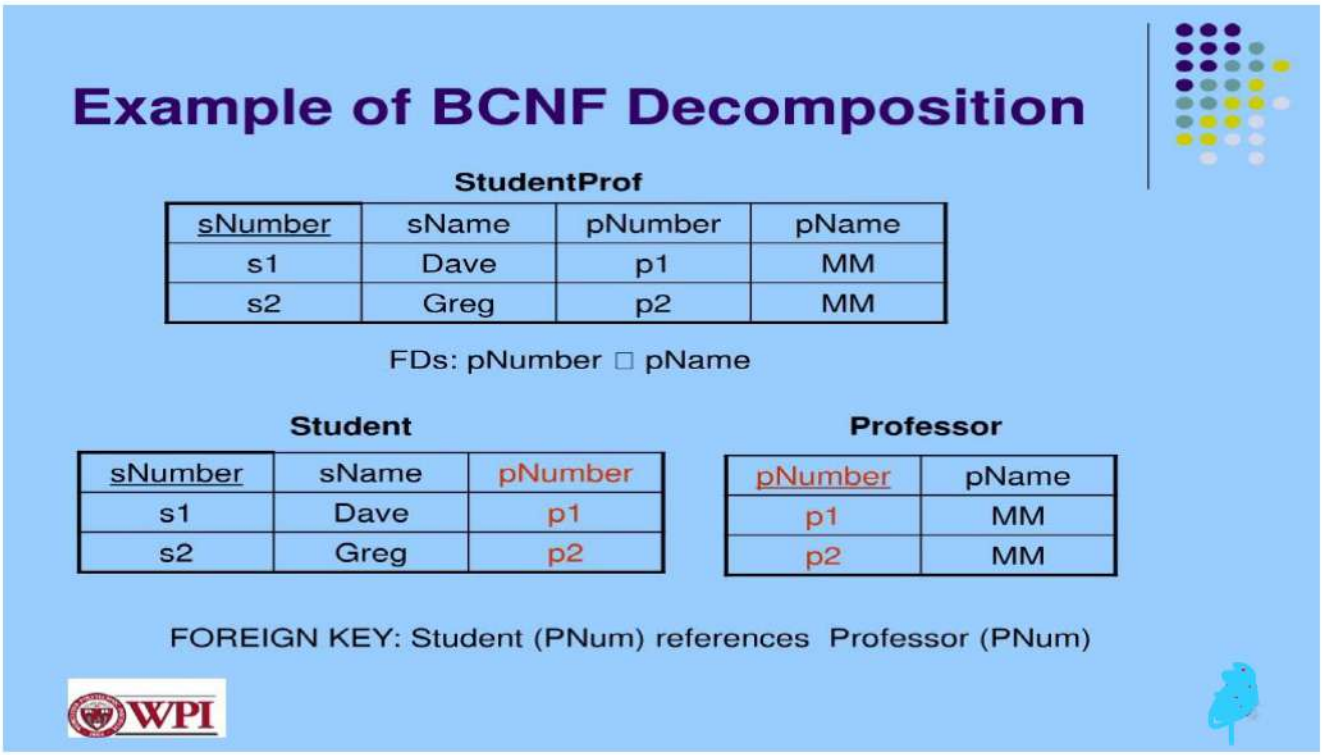
- Must be in 2NF: Ensures 1NF and removes partial dependencies.
- No Transitive Dependencies: Non-prime attributes must not depend on other non-prime attributes.
- Further Reduces Redundancy: Minimizes redundancy by organizing data based on dependencies.



4. Boyce-Codd Normal Form (BCNF):

Features:

- Must be in 3NF: Ensures 1NF and removes partial and transitive dependencies.
- Every Determinant is a Superkey: All non-prime attributes are functionally dependent on superkeys.
- Further Eliminates Redundancy: Achieves a higher level of normalization by eliminating additional forms of dependency.



==========

Unit -4. Relational Language

4.1 Introduction to SQL
4.2 Features of SQL
4.3 Basic retrieval Queries
4.4 INSERT, UPDATE, DELETE Queries
4.5 Join, Semi Join and Sub Queries
4.6 Views
4.7 Relational Algebra
- **4.7.1 Select, Project**
- **4.7.2 Set Operations**
- **4.7.3 Cartesian Product**
- **4.7.4 Join**

4.1 Introduction to SQL (Structured Query Language)

- SQL is the set of instructions used to interact with databases. It allows users to do **CRUD** Operation.

Where C=Create

R= Retrieve

U=Update

D= Delete.

- SQL is a special-purpose programming language designed for managing data stored in a relational database management system.
- SQL is the language with which a programmer communicates with a database to manipulate its data.

4.2 Features of SQL

- SQL is an English-like language . It uses words such as select , insert , delete as part of its commend set.
- SQL is an a non-procedural language :
- SQL processes sets of records rather than a single record at a time . The most common form of a set of records is a table.
- SQL can be used by a range of user including DBAs application programmers , management personal , and many other types of end users.
- SQL Provides command for a variety of tasks including:
  - querying data
  - Inserting, updating and deleting rows in a table
  - Creating, modifying and deleting database objects
  - controlling access to the database and database objects
  - guaranteeing database consistency.

4.3 Basic retrieval Queries

1. **SELECT All Columns from a Table:**

   Note: Retrieves all columns from a specified table.

   Example: SELECT * FROM employees;

2. **SELECT Specific Columns:**

   Note: Retrieves specific columns from a table.

   Example: SELECT first_name, last_name, salary FROM employees;

3. **Filtering with WHERE Clause:**

   Note: Retrieves rows that satisfy a specified condition.

   Example: SELECT product_name, price FROM products WHERE category = 'Electronics';

4. **Sorting with ORDER BY:**

   Note: Retrieves records and orders them based on a specified column.

   Example: SELECT product_name, price FROM products ORDER BY price DESC;

5. **Limiting Results with LIMIT:**

   Note: Retrieves a specific number of rows from the result set.

   Example: SELECT * FROM orders LIMIT 10;

6. **Aggregate Functions (SUM, AVG, COUNT):**

   Note: Performs aggregate calculations on a set of values.

   Example: SELECT AVG(salary) FROM employees;

4.4 INSERT, UPDATE, DELETE Queries

Here are examples of basic SQL queries for INSERT, UPDATE, and DELETE operations:

INSERT SQL Queries:

Note: Adds new records to a table.

Example: This example inserts a new employee with the specified first name, last name, and salary into the "employees" table.

```
INSERT INTO employees (first_name, last_name, salary)
VALUES ('John', 'Doe', 50000);
```

UPDATE SQL Queries:

Note: Modifies existing records in a table based on a condition.

Example: This example updates the price of all products in the "Electronics" category to $150.

```
UPDATE products
SET price = 150
WHERE category = 'Electronics';
```

DELETE SQL Queries:

Note: Removes records from a table based on a condition.

Example: This example deletes customers whose last purchase date is before January 1, 2023, from the "customers" table.

```
DELETE FROM customers
WHERE last_purchase_date < '2023-01-01';
```

4.5 Join, Semi Join and Sub Queries

1. **JOIN Operation:**

   Note: Combines rows from two or more tables based on a related column between them.

   Example: This example retrieves the order ID and customer name by joining the "orders" and "customers" tables on the common column "customer_id."

```
SELECT orders.order_id, customers.customer_name
FROM orders
INNER JOIN customers ON orders.customer_id = customers.customer_id;
```

2. **Semi Join Operation:**

   Note: Retrieves rows from the first table where a related row exists in the second table.

   Example: This example retrieves product names from the "products" table where the corresponding product ID exists in the "order_items" table.

```
SELECT product_name
FROM products
WHERE product_id IN (SELECT product_id FROM order_items);
```

3. **Subquery:**

   Note: A query nested inside another query, which can be used within SELECT, FROM, WHERE, and other clauses.

   Example: This example uses a subquery to retrieve customer names from the "customers" table based on those who placed orders after January 1, 2023.

```
SELECT customer_name
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders WHERE
 order_date > '2023-01-01');
```

4.6 Views

Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

ID	NAME	MARKS	AGE
1	Harsh	90	19
2	Suresh	50	20
3	Pratik	80	19
4	Dhanraj	95	21
5	Ram	85	18

CREATING VIEWS

We can create View using **CREATE VIEW** statement. A View can be created from a single table or multiple tables.

Syntax:

```
CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE condition;
```

view_name: Name for the View

table_name: Name of the table

condition: Condition to select rows

Examples:

Creating View from a single table:

In this example we will create a View named DetailsView from the table StudentDetails.

Query:

```
CREATE VIEW DetailsView AS
SELECT NAME, ADDRESS
FROM StudentDetails
WHERE S_ID < 5;
```

To see the data in the View, we can query the view in the same manner as we query a table.

```
SELECT * FROM DetailsView;
```

Output:

NAME	ADDRESS
Harsh	Kolkata
Ashish	Durgapur
Pratik	Delhi
Dhanraj	Bihar

Uses/Application /Characteristics of a View:

A good database should contain views due to the given reasons:

1. **Restricting data access –** Views provide an additional level of table security by restricting access to a predetermined set of rows and columns of a table.
2. **Hiding data complexity –** A view can hide the complexity that exists in multiple tables join.
3. **Simplify commands for the user –** Views allow the user to select information from multiple tables without requiring the users to actually know how to perform a join.
4. **Store complex queries –** Views can be used to store complex queries.
5. **Rename Columns –** Views can also be used to rename the columns without affecting the base tables provided the number of columns in view must match the number of columns specified in select statement. Thus, renaming helps to hide the names of the columns of the base tables.
6. **Multiple view facility –** Different views can be created on the same table for different users.

4.7 Relational Algebra

RELATIONAL ALGEBRA is a widely used procedural query language. It collects instances of relations as input and gives occurrences of relations as output. It uses various operations to perform this action.

4.7.1 Select, Project

Select (σ):

The select operation (σ) is used to retrieve a subset of tuples from a relation that satisfy a given condition.

Example: Suppose you have a relation Students with attributes Name, Age, and Grade. The select operation could be used to retrieve all students with an age greater than 20:

- σAge > 20(Students)

This expression would return a subset of the Students relation containing only those tuples where the age is greater than 20.

Project (π):

The project operation (π) is used to select specific columns from a relation, creating a new relation with only the specified attributes.

Example: Using the same Students relation, the project operation could be used to retrieve only the Name and Grade attributes:

- πName, Grade(Students)

This expression would result in a new relation containing only the Name and Grade columns from the original Students relation.

4.7.2 Set Operations

Relational algebra includes set operations such as union (U), intersection (∩), and difference (-), which can be applied to relations to combine or compare their tuples.

Example: If you have two relations, A and B, the union of these relations (AUB).

Would include all tuples that are in either A or B.

4.7.3 Cartesian Product

The Cartesian product operation (×) combines tuples from two relations, resulting in a new relation with all possible combinations of tuples.

Example: If you have two relations, A and B, the Cartesian product A×B would include all possible pairs of tuples where the first element is from A and the second element is from B.

4.7.4 Join

The join operation combines tuples from two relations based on a common attribute, creating a new relation.

Example: Suppose you have two relations, Students and Courses, with a common attribute StudentID. The join operation could be used to retrieve a relation that combines information about students and the courses they are enrolled in:

$$\text{Students} \bowtie_{\text{StudentID}} \text{Courses}$$

This expression would create a new relation by combining tuples from Students and Courses where the StudentID values match.

==========

==========

For Practical purpose only

Create any two table and perform various operations

```
create database equipment
use equipment
create table gadget(gid int not null,name varchar(25),gadget_type
varchar(20),price float)
insert into gadget values(1,'tv','electronic',4200);

select *from gadget

CREATE TABLE vehicle (vid int NOT NULL, name
varchar(25),vehicle_type varchar(20),price float);
insert into vehicle values(1,'cycle','two wheeler',800000);

select *from vehicle
```

joins (inner join)

```
SELECT gadget.gid, vehicle.name, gadget.price
FROM gadget
INNER JOIN vehicle ON gadget.gid=vehicle.vid;
```

joins (cross join)

```
SELECT gid
FROM gadget
CROSS JOIN vehicle
```

joins (left join)

```
SELECT gadget.name, vehicle.vid
FROM gadget
LEFT JOIN vehicle ON gadget.gid = vehicle.vid
ORDER BY gadget.name;
```

joins (right join)

```
SELECT gadget.name, vehicle.vid
FROM gadget
right JOIN vehicle ON gadget.gid = vehicle.vid
ORDER BY gadget.name;
```

joins (full join) (Note: FULL OUTER JOIN and FULL JOIN are the same.)

```
SELECT gadget.name, vehicle.vid, vehicle.price
FROM gadget
full outer join vehicle ON gadget.gid = vehicle.vid
ORDER BY gadget.name;
```

Join (self join)

```
SELECT e.name AS gadget, h.name AS stuff
FROM gadget e
JOIN gadget h ON e.gid = h.gid;
```

MIN and MAX:

```
SELECT MIN(Price)
FROM vehicle;
```

```
SELECT MAX(Price)
FROM vehicle;
```

Average:

```
SELECT AVG(Price)
FROM vehicle;
```

Sum:

```
SELECT SUM(price)
FROM vehicle;
```

Count:

```
SELECT COUNT(*)
FROM vehicle;
```

UNION ALL Syntax:

```
SELECT name FROM gadget
UNION ALL
SELECT name FROM vehicle;
```

SQL UNION With WHERE

```
SELECT gid FROM gadget
WHERE gid=1
UNION
SELECT vid FROM vehicle
WHERE vid=1
order by gid;
```

Alter with foreign key example:

alter table Employees add constraint fk_table1_team

foreign key (EmployeeID) REFERENCES student(sid);

ALTER TABLE Employees

ADD Price varchar(255)

==========

Unit 5. Query Processing

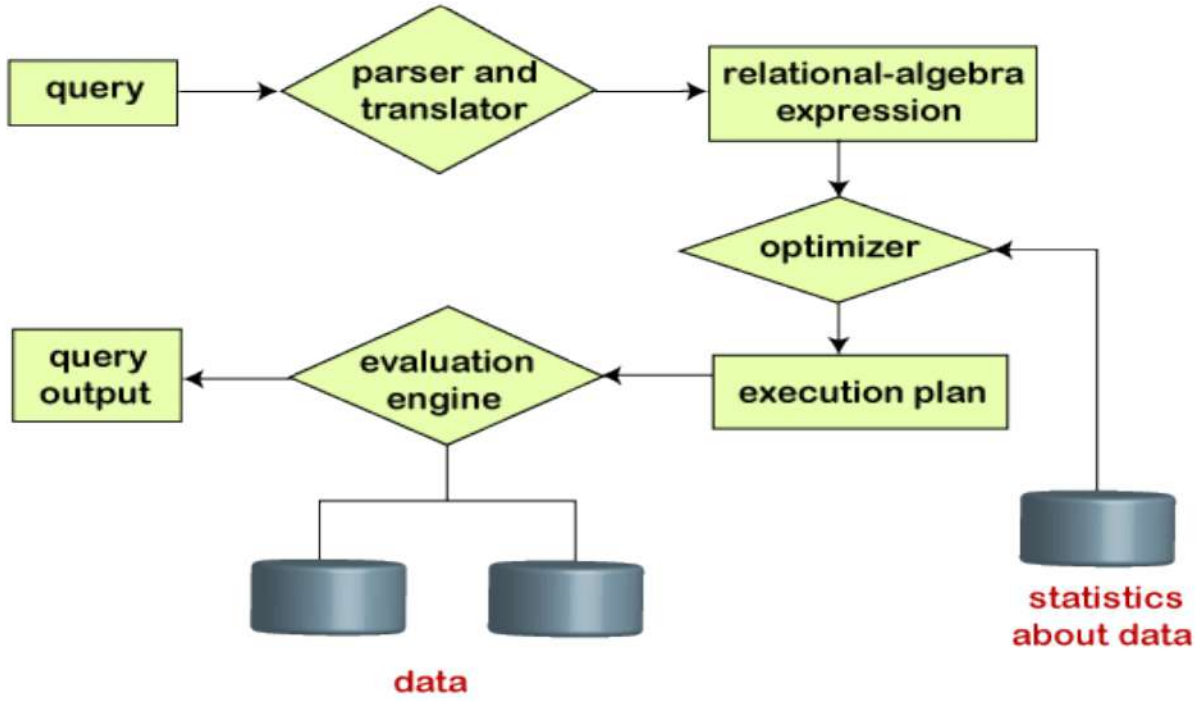
5.1 Introduction to Query Processing

Query Processing is a translation of high-level queries into low-level expression. It is a step wise process that can be used at the physical level of the file system, query optimization and actual execution of the query to get the result.

Query Processing is the activity performed in extracting data from the database.

In query processing, it takes various steps for fetching the data from the database. The steps involved are:

- Parsing and translation
- Optimization
- Evaluation



Steps in query processing

EXAMPLE:
BEFORE QUERY PROCESSING:

Products Table

ProductID	Name	Category	Price	Quantity
101	Laptop	Electronics	800	10
102	Smartphone	Electronics	500	20
103	Headphones	Electronics	100	30
104	Backpack	Fashion	50	15

User Query:

User enters the query: **"names and prices of electronics products with a price less than 600."**

Parsing:

The system parses the query and identifies the components: SELECT (names and prices), FROM (Products), and WHERE (electronics products with a price less than 600).

Query Optimization:

In this simple example, optimization may involve determining the most efficient way to access the required data, such as using an index on the "Category" column.

Execution:

The system executes the query, scanning the "Products" table, filtering rows where the category is "Electronics" and the price is less than 600, and retrieving the names and prices of matching products.

AFTER QUERY PROCESSING: (OUTPUT)

Name	Price
Smartphone	500
Headphones	100

5.2 Query Cost estimation

Query cost estimation in DBMS involves estimating the resources and time required to execute a given query. **The optimizer aims to minimize the overall cost of executing a query,** and this cost encompasses various factors.

The cost estimation of a query evaluation plan is calculated in terms of various resources that include:

- **Number of disk accesses**
- **Execution time taken by the CPU** to execute a query
- Communication costs in distributed or parallel database systems.

Number of Disk Accesses: The optimizer estimates the number of reads and writes to the disk, considering table and index scans.

CPU Time: The cost model calculates the expected CPU processing time for operations like filtering and joining.

Communication costs in distributed or parallel database systems: Network Latency, Data Transfer Volume etc....

Before cost estimation:	After cost estimation:
CREATE TABLE Scores (StudentID INT PRIMARY KEY, ExamSubject VARCHAR(50), Score INT);	SELECT AVG(Score) AS AverageScore FROM Scores WHERE ExamSubject = 'Math' AND Score > 80; Note: The cost model estimates the cost components: • CPU Cost: Minimal computation for averaging. • I/O Cost: Scanning the table to find rows where ExamSubject = 'Math' and Score > 80.

5.3 Query Operations, Operator TREE

Query Operations

A query operation refers to a specific action or computation that is performed on the data to retrieve, manipulate, or organize information. Query operations are often represented in the form of an operator tree or query execution plan, which outlines the sequence of operations needed to fulfill a given query.

Common Query Operations:

Projection (π):

Selecting specific columns from a table.

Example: SELECT Name, Age FROM Students;

Selection (σ):

Filtering rows based on a condition.

Example: SELECT * FROM Employees WHERE Department = 'IT';

Join (⋈):

Combining rows from two or more tables based on a related column.

Example: SELECT * FROM Orders JOIN Customers ON Orders.CustomerID = Customers.CustomerID;

Aggregation (Σ):

Performing operations like COUNT, SUM, AVG, MAX, or MIN on a set of rows.

Example: SELECT AVG(Salary) FROM Employees;

Sorting (τ):

Arranging rows based on one or more columns.

Example: SELECT * FROM Products ORDER BY Price DESC;

Operator Tree:

An operator tree (or query execution plan) visually represents the sequence of query operations needed to execute a given query. The nodes of the tree represent operations, and the edges indicate the flow of data between operations.

Example:

```
SELECT FirstName, LastName
FROM Employees
WHERE Department = 'IT'
ORDER BY Salary DESC;
```

In operator tree:

```
SORT (Salary DESC)
  |
PROJECT (FirstName, LastName)
  |
SELECT (Department = 'IT')
  |
SCAN (Employees)
```

5.4 Evaluation of Expressions

(Expression is the combination of operators and operands, suppose a+b^2=c, where a,b,c and 2 is the operands whereas +,^,= are the operators).

Query evaluation of expressions involves the process of computing and producing the results of expressions specified in a query. In the context of databases, this often includes arithmetic operations, comparisons, and other expressions used within SELECT statements.

I) Arithmetic Expressions:

SELECT ProductName, Quantity * UnitPrice AS TotalCost (**MULTIPLY (*)** SIGN IS USED HERE)
FROM Products;

II) Comparison Expressions:

SELECT * FROM Employees WHERE Salary > 50000; (**Greater THAN SIGN(>)** IS USED HERE)

III) Aggregate Expressions with GROUP BY:

SELECT Category, SUM(Revenue) AS TotalSales (**GROUP BY** PHRASE IS USED HERE)
FROM Sales
GROUP BY Category;

IV) Logical Expressions:

SELECT * FROM Employees WHERE Department = 'IT' **AND** Salary > 60000; (**AND** LOGICAL OPERATOR IS USED HERE)

5.5 Query Optimization

Query optimization is the process of choosing the most efficient execution plan for a database query to minimize resource usage.

Before query optimization example:

SELECT * FROM Products WHERE **Category = 'Electronics' AND Price > 500;**

After query optimization example:

SELECT ProductName, Price FROM Products WHERE **Category = 'Electronics' AND Price > 500;**

Note: Optimization includes selecting specific columns to reduce data retrieval and potentially utilizing indexes for faster access.

5.6 Performance Tuning

In SQL or databases, performance tuning specifically refers to activities to improve the speed and efficiency of database operations.

The goal is to ensure that queries and transactions are executed in the most optimal way to meet user expectations.

Some common types of performance tuning strategies often applied in the context of databases are:

1) **Indexing:**
   Create appropriate indexes on columns frequently used in WHERE clauses and JOIN conditions to speed up data retrieval.
2) **Query Optimization:**
   Analyze and optimize SQL queries by rewriting, simplifying, or restructuring them to improve execution efficiency.
3) **Caching:**
   Implement caching mechanisms to store frequently accessed data in memory, reducing the need for repeated database queries.
4) **Partitioning:**
   Partition large tables into smaller, more manageable pieces based on certain criteria (e.g., date ranges) to improve query performance.
5) **Concurrency Control:**
   Optimize transaction isolation levels to balance consistency and performance, minimizing contention in multi-user environments.

For example:

Before Performance Tuning:

```
SELECT * FROM Orders WHERE OrderDate BETWEEN '2022-01-01' AND '2022-12-31';
```

After Performance Tuning:

```
SELECT OrderID, CustomerID, OrderDate FROM Orders WHERE OrderDate BETWEEN '2022-01-01' AND '2022-12-31';
```

Note: The tuned query specifies only the necessary columns (OrderID, CustomerID, OrderDate), reducing the amount of data retrieved and potentially improving performance.

==========

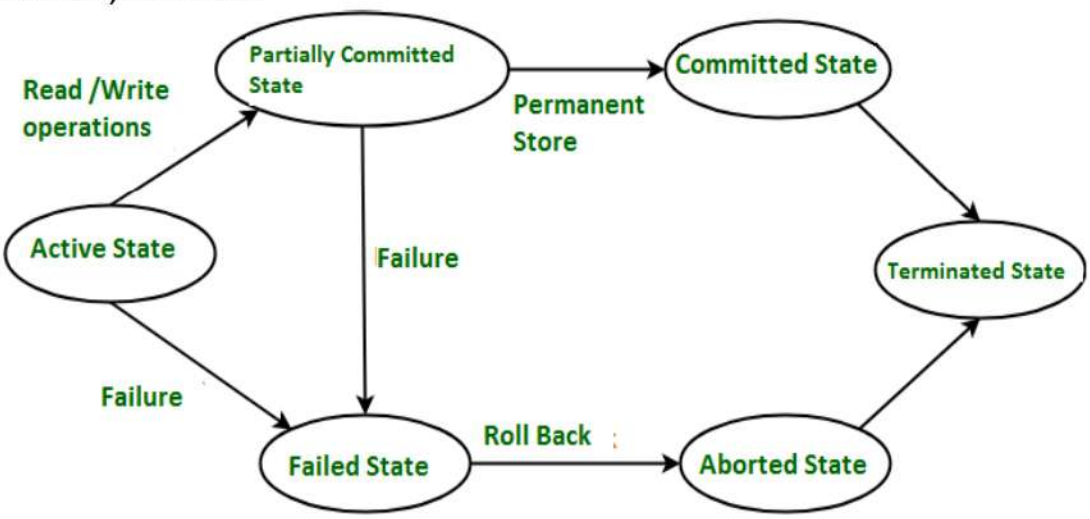
Unit 6. Transaction and Concurrency Control

6.1. Introduction to Transaction

A transaction is a set of one or more database operations that are executed as a single, indivisible unit of work.

Transaction States:

Transactions typically go through various states: Active, Partially Committed, Committed, Aborted.



Transaction States in DBMS

- **Active:** The initial state when the transaction is executing.
- **Partially Committed:** The point at which the transaction is successfully completed but not yet fully committed.
- **Committed:** The final state where changes made by the transaction are permanently saved.
- **Aborted:** The state reached if a transaction encounters an error or is intentionally rolled back.

Transaction Operations:

Common transaction operations include Read (R), Write (W), Commit (C), and Rollback (R).

1. **Read (R):** Retrieves data from the database.
2. **Write (W):** Modifies or inserts data into the database.
3. **Commit (C):** Permanently saves the changes made by the transaction.
4. **Rollback (R):** Reverses the changes made by the transaction and brings the database back to its previous state.

6.2. Serializability concept:

Serializability is a fundamental concept in database management systems (DBMS) that ensures the correctness and isolation of concurrent transactions.

It defines as a set of schedules (sequences of operations from multiple transactions) that produce the same result as if the transactions were executed in some sequential order.

Examples:

Let's consider two transactions in a banking system: T1 and T2.

- **Transaction T1:**
  - Reads the balance of account A.
  - Deducts $100 from the balance of account A.
  - Writes the updated balance back to account A.
- **Transaction T2:**
  - Reads the balance of account A.
  - Deposits $50 into the balance of account A.
  - Writes the updated balance back to account A.

Now, let's examine two different schedules to see if they are serializable:

- **Schedule S1:**

  T1 reads
  T2 reads
  T1 writes
  T2 writes

  In this schedule, the transactions execute in the order T1, T2, T1, T2. The final balance depends on the order of execution.
- **Schedule S2:**

  T2 reads
  T1 reads
  T2 writes
  T1 writes

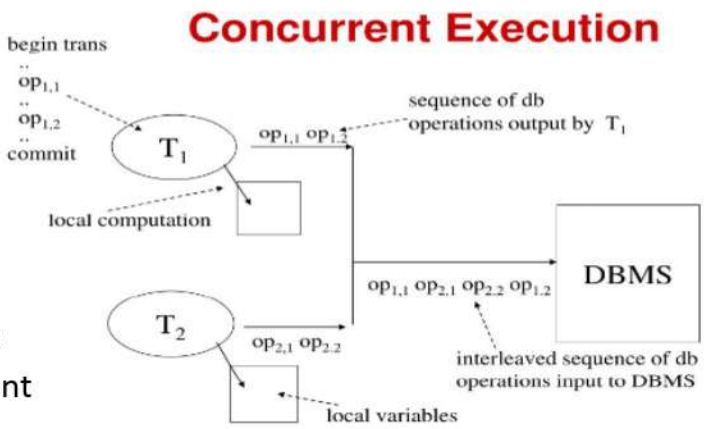
  In this schedule, the transactions execute in the order T2, T1, T2, T1. Again, the final balance depends on the order of execution.

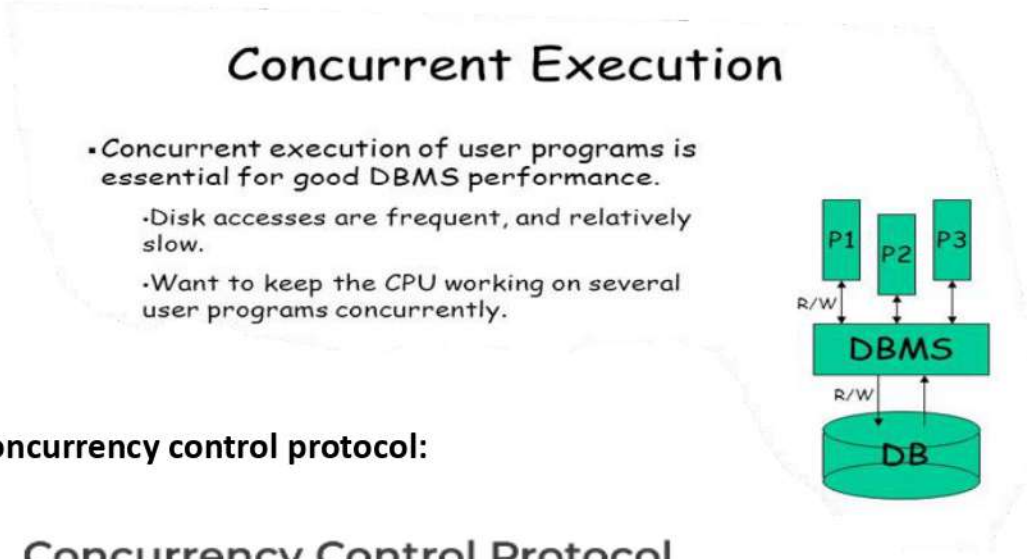
6.3. Concurrent execution

Concurrent execution in a database management system (DBMS) refers to the simultaneous execution of multiple transactions.

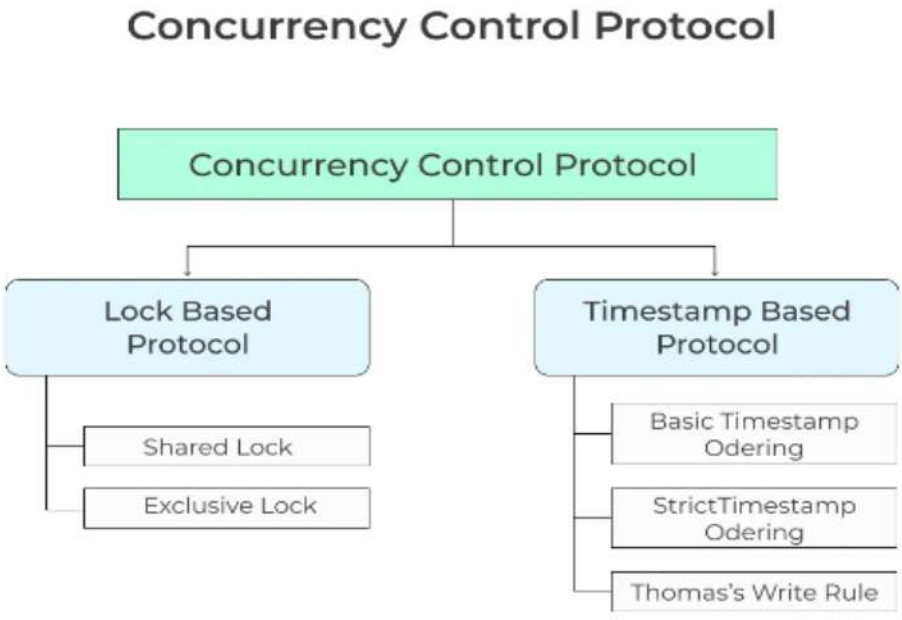
The goal is to improve system throughput and response time by allowing transactions to run concurrently.

In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.



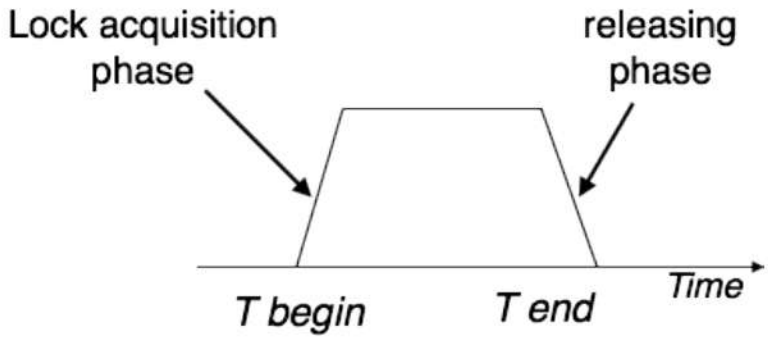


Types of concurrency control protocol:



6.4. Lock based Concurrency Control

Lock-based Concurrency Control uses locks to prevent conflicts between transactions, ensuring that only one transaction can modify data at a time, maintaining database consistency.



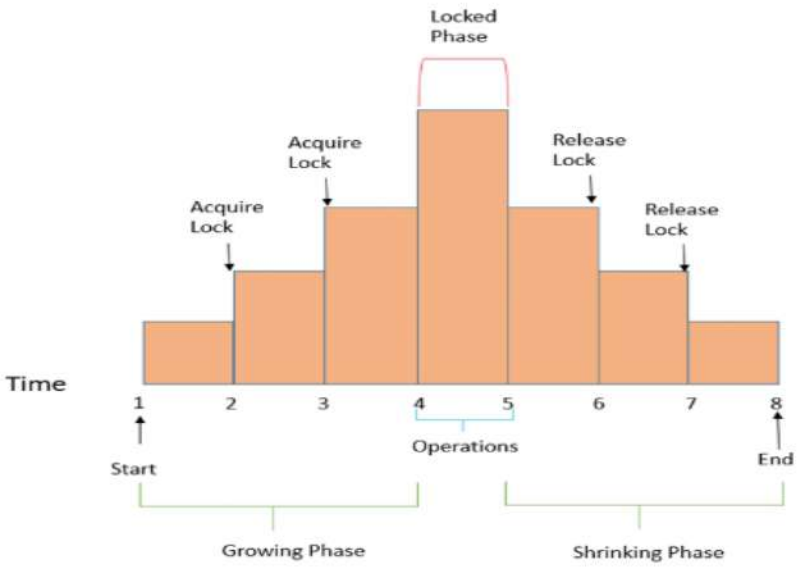
6.5. 2PL and Strict 2PL

2PL:

2PL stands for two phase locking whereas Transaction management approach allowing locks to be acquired and released in two phases, providing **flexibility in lock handling during a transaction.**

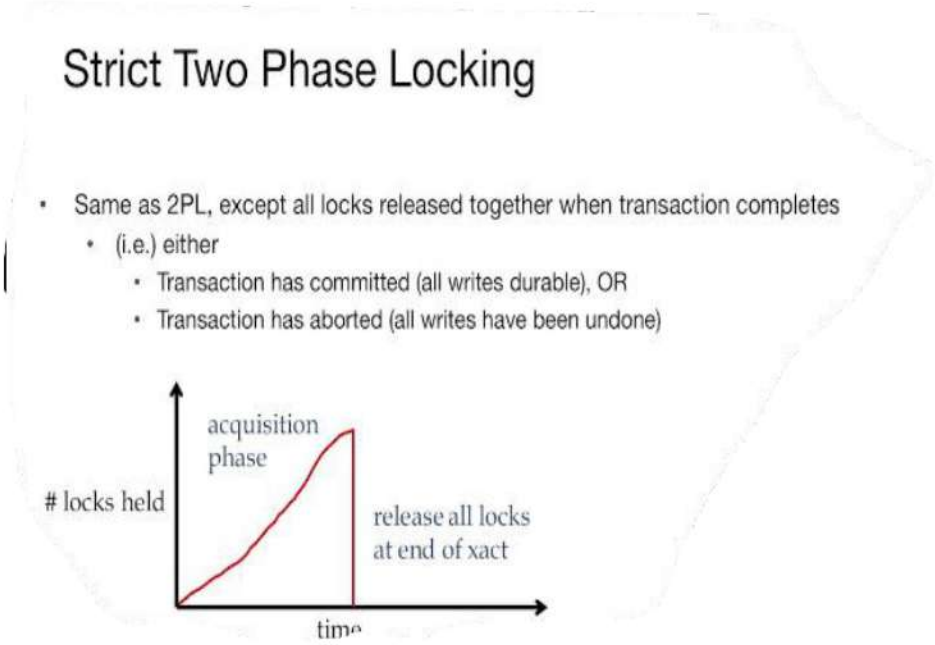
It has two phase

i. Growing phase and
ii. Shrinking phase



Strict 2PL:

A more rigid form of 2PL where all locks acquired during a transaction are held until its completion, reducing the risk of deadlocks.



Difference between 2 PL(Two Phase Locking)and Strict 2PL(Strict Two Phase locking)

Two-Phase Locking (2PL)	Strict Two-Phase Locking (Strict 2PL)
Allows releasing some locks early.	Holds all locks until the end of the transaction.
May face a risk of deadlocks.	Less likely to face deadlocks.
Can commit even if some locks are released early.	No commitment until all locks are held until the end.
Reflected in the way SQL transactions acquire and release locks.	Imposes stricter rules on when locks can be released based on SQL operations.
Locks might be held at a coarser level, providing more flexibility.	Requires holding locks at a finer level, limiting flexibility.
Allows variations in how transactions control locks, offering flexibility.	Enforces a stricter control on how locks are acquired and held in transactions.

6.6. Timestamp concept

Timestamp concept assigns unique time identifiers to transactions in a database. It helps order transactions, preventing conflicts and ensuring consistency by allowing the system to determine the chronological order of their execution.

Timestamp-Based Protocols

There are two simple methods for implementing this scheme:

1. Use the value of the **system clock** as the timestamp;
2. Use a **logical counter** that is incremented after a new timestamp has been assigned;

Thus, if $\mathbf{TS}(T_i) < \mathbf{TS}(T_j)$, then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j.

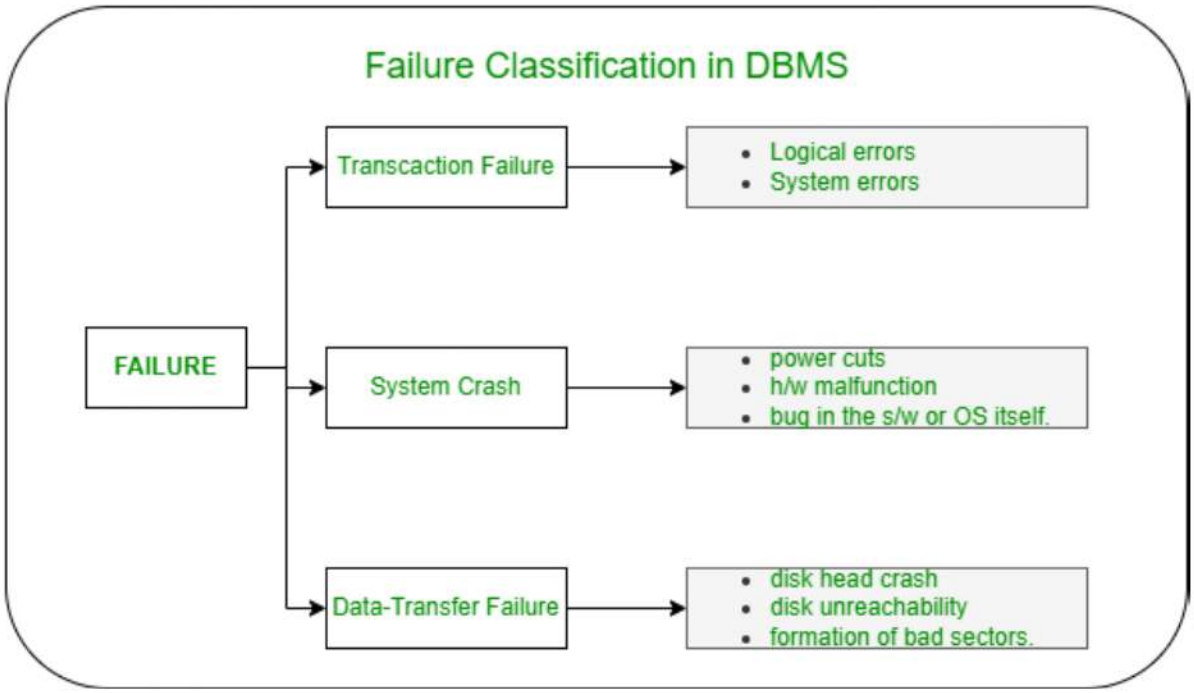
==========

Unit 7. Recovery

7.1. Failure Classifications

Failure of a database can be defined as its inability to execute the specified transaction or loss of data from the database.

A DBMS is vulnerable to several kinds of failures and each of these failures needs to be managed differently. There are many reasons that can cause database failures such as network failure, system crash, natural disasters, carelessness, sabotage (corrupting the data intentionally), software errors, etc.



7.2. Recovery and Atomicity

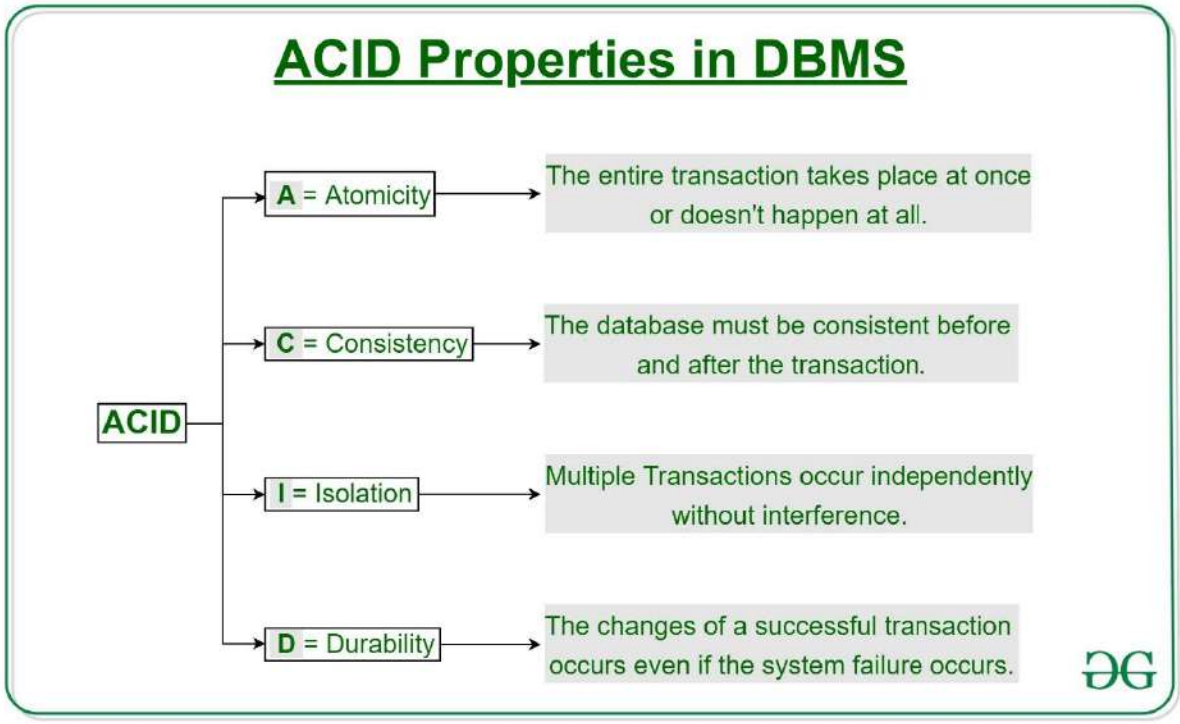
When a system crashes, it may have several transactions being executed and various files opened for them to modify the data items. Transactions are made of various operations, which are atomic in nature. But according to ACID properties of DBMS, atomicity of transactions as a whole must be maintained, that is, either all the operations are executed or none.

When a DBMS recovers from a crash, it should maintain the following −

- It should check the states of all the transactions, which were being executed.
- A transaction may be in the middle of some operation; the DBMS must ensure the atomicity of the transaction in this case.
- It should check whether the transaction can be completed now or it needs to be rolled back.
- No transactions would be allowed to leave the DBMS in an inconsistent state.

There are two types of techniques, which can help a DBMS in recovering as well as maintaining the atomicity of a transaction −

1) Maintaining the logs of each transaction, and writing them onto some stable storage before actually modifying the database.
2) Maintaining shadow paging, where the changes are done on a volatile memory, and later, the actual database is updated.



7.3. IN PLACE and Out of Place Update

"in-place update" and "out-of-place update" refer to different approaches for modifying data. These terms are commonly used in the context of updating records in a table.

Example of **in place update**:

```
UPDATE table_name
SET column1 = new_value1, column2 = new_value2
WHERE condition;
```

Example of **out of place update**:

```
CREATE TABLE new_table AS
SELECT column1, column2, ..., new_value1 AS column1, new_value2 AS column2, ...
FROM old_table
WHERE condition;
```

Difference between in place and out of place update :

Properties	In-Place Update	Out-of-Place Update
Data Modification	Updates existing data directly.	Creates a new version of the data.
Performance	Generally faster and uses fewer resources.	May be slower and requires more resources.
Data Consistency	Prone to leaving data inconsistent if update fails.	Safer as it preserves the original data until the new version is created.
Storage Usage	Requires less storage space.	May consume more storage space due to creating a new version.
Example SQL	UPDATE table_name SET column1 = new_value1 WHERE condition;	CREATE TABLE new_table AS SELECT column1, new_value1 FROM old_table WHERE condition;
Use Cases	Suitable for scenarios where efficiency is crucial.	Preferred when preserving the original data is essential.

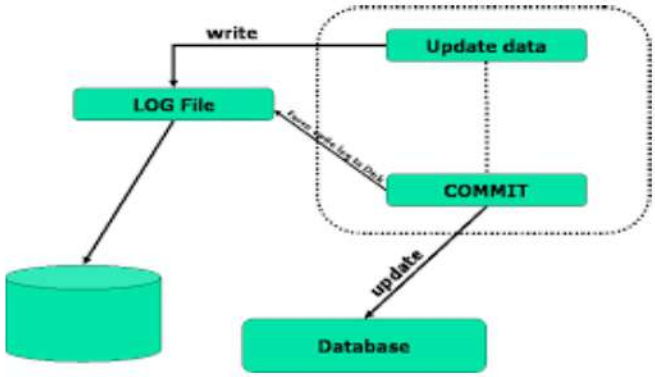
7.4. Log based Recovery

Log is a sequence of records, which maintains the records of actions performed by a transaction. It is important that the logs are written prior to the actual modification and stored on a stable storage media, which is failsafe.

Log-based recovery works as follows –

The log file is kept on a stable storage media.

When a transaction enters the system and starts execution, it writes a log about it.



<Tn, Start>

When the transaction modifies an item X, it write logs as follows –

<Tn, X, V1, V2>

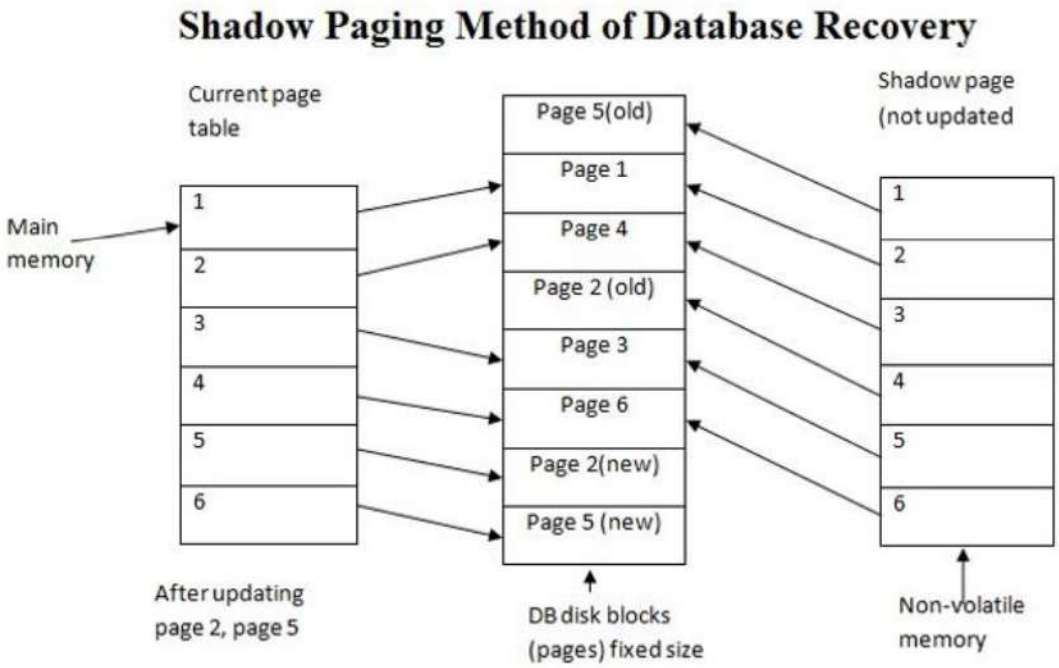
It reads Tn has changed the value of X, from V1 to V2.

When the transaction finishes, it logs –

<Tn, commit>

7.5. Shadow Paging

Shadow Paging is recovery technique that is used to recover database. In this recovery technique, database is considered as made up of fixed size of logical units of storage which are referred as pages. pages are mapped into physical blocks of storage, with help of the page table which allow one entry for each logical page of database. This method uses two page tables named current page table and shadow page table. i.e., Shadow page table is used when the transaction starts which is copying current page table. After this, shadow page table gets saved on disk and current page table is going to be used for transaction.



7.6. Local Recovery Manager

The "Local Recovery Manager" is a component within a database system responsible for recovering an individual database instance after a failure.

Key functions include managing transaction logs, performing redo and undo operations, and utilizing checkpoints for stability. It operates independently at the local level.

For example:

After a system crash, the recovery manager analyzes transaction logs, reapplies committed transactions using redo operations, rolls back uncommitted transactions

with undo operations, and utilizes checkpoints for stability. This ensures the local database's consistency and integrity.

7.7. UNDO and REDO protocol

UNDO Protocol:

Purpose: Reverses the effects of incomplete or uncommitted transactions during recovery.

Operation: Rolls back changes recorded in transaction logs to restore consistency.

Example:

If a transaction fails to complete (e.g., system crash), UNDO ensures any changes it made are rolled back, restoring the original state.

REDO Protocol:

Purpose: Reapplies changes made by committed transactions during recovery.

Operation: Re-executes committed transactions using information from transaction logs.

Example:

If a committed transaction's changes are in volatile memory (not yet saved), REDO ensures these changes are re-executed during recovery, preventing data loss.

==========**Best wishes**==========

Name : Dilbar Yadav
Clz : Narayani Model Secondary School Bharatpur 10 Chitwan