Module 1

1. Explain the characteristics of database approach.

Self-Describing Nature of a Database System

A **database management system (DBMS)** is self-describing because it not only stores the actual data but also maintains a description of the data, known as **meta-data**. This meta-data is stored in the **DBMS catalog** and includes details such as:

- **Data Structures:** Information about tables, columns, indexes, and relationships between tables.
- **Data Types:** Definitions of the types of data that can be stored (e.g., integer, string, date).
- **Constraints:** Rules governing the data, such as primary keys, foreign keys, uniqueness, and check constraints.

The self-describing nature of a DBMS allows the system to be flexible and adaptable, as it can work with different database applications without needing to be hard coded for specific structures.

Insulation Between Programs and Data (Program-Data Independence)

Program-data independence refers to the separation of the data and the programs that use the data. This concept ensures that changes to the data structure (such as modifying a table or changing the way data is stored) do not require changes to the application programs that access the data.

For example:

- If you add a new column to a table, you don't need to rewrite all the applications that use that table unless they need to access the new column.
- If the storage organization of the data is changed, the DBMS ensures that applications can continue to retrieve and manipulate data without needing to know about the underlying changes.

Data Abstraction

Data abstraction is the process of hiding the complexities of data storage and presenting users with a simplified, conceptual view of the database. The DBMS uses a **data model** to define this conceptual view, which includes:

- **Entities and Relationships:** Representing real-world objects and the connections between them.
- **Attributes:** Describing the properties of entities.
- **Constraints:** Defining rules and relationships between data elements.

Support of Multiple Views of the Data

A DBMS supports **multiple views** of the same database, allowing different users or applications to see different subsets or representations of the data. Each view can be tailored to meet the specific needs of a user or a group of users.

For example:

- A sales department might see a view that includes customer orders and payment information.
- The finance department might see a view that includes customer balances and financial transactions but not order details.

Sharing of Data and Multi-User Transaction Processing

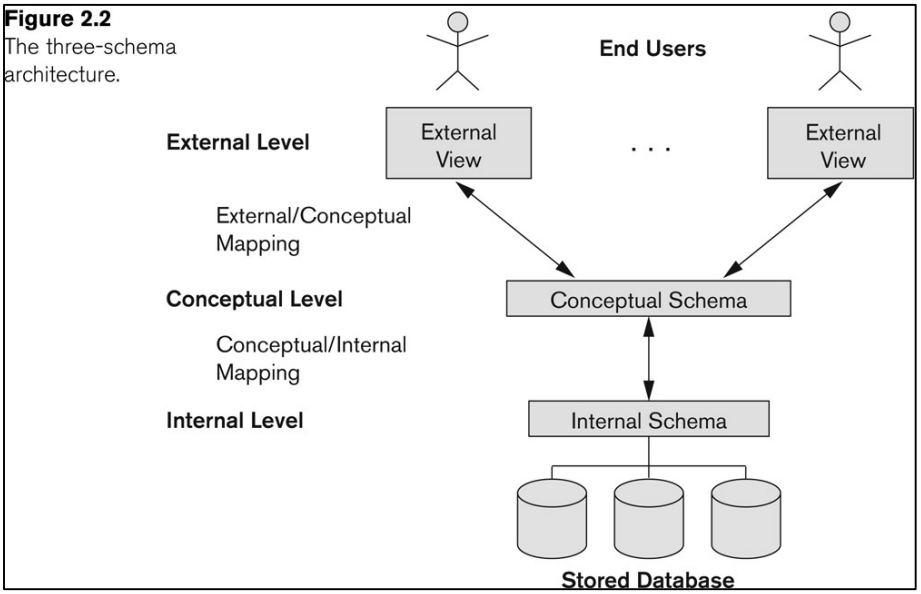
A DBMS is designed to support the **sharing of data** among multiple users and to handle **multi-user transaction processing** efficiently. In a multi-user environment, the DBMS must manage concurrent access to ensure that data remains consistent and accurate. Key components of this functionality include:

- **Concurrency Control:** The DBMS uses various techniques to ensure that transactions are executed in a way that prevents conflicts, such as two users trying to update the same record simultaneously. This control guarantees that each transaction is isolated from others, maintaining data integrity.
- **Recovery Subsystem:** The DBMS includes mechanisms to recover from failures. If a transaction is completed successfully, the changes it made are permanently recorded in the database. If a transaction fails, the DBMS ensures that no partial or incorrect changes are saved.
- **Online Transaction Processing (OLTP):** OLTP systems are optimized for handling a large number of short, fast transactions that involve updating, inserting, or deleting small amounts of data. These systems are essential for businesses that require real-time processing.

2. Explain the advantages of using DBMS Approach.

- **Controlling Redundancy in Data Storage**:
  - Databases reduce data redundancy through normalization, minimizing duplication and ensuring consistency.
- **Sharing Data Among Multiple Users**:
  - A DBMS supports concurrent access, allowing multiple users to work with the same data simultaneously while maintaining consistency through concurrency control.
- **Restricting Unauthorized Access**:
  - Databases enforce security through user authentication, roles, and permissions, ensuring that only authorized users can access or modify data.
- **Persistent Storage for Program Objects**:
  - DBMSs support the persistent storage of objects, enabling data to retain its state between sessions.
- **Efficient Query Processing**:
  - Databases use indexes to speed up data retrieval, making queries more efficient, especially with large datasets.
- **Backup and Recovery**:
  - DBMSs provide tools for regular backups and recovery mechanisms to restore data after failures, ensuring minimal data loss.
- **Multiple User Interfaces**:
  - Databases offer various interfaces like SQL, GUIs, APIs, and web interfaces, catering to different user needs.
- **Complex Relationships**:
  - Databases can model and manage complex data relationships, such as one-to-many and many-to-many, using keys and relational structures.
- **Enforcing Integrity Constraints**:
  - Integrity constraints (e.g., primary keys, foreign keys, check constraints) ensure data accuracy and consistency within the database.
- **Automation and Rules**:
  - Deductive rules and triggers allow the database to automate actions and infer new data, supporting complex decision-making processes.

3. Explain three schema Architecture.



The **Three-Schema Architecture** was proposed to enhance the flexibility and usability of a DBMS by providing clear separation between different levels of data abstraction, supporting key characteristics like **program-data independence** and **multiple user views**. It defines three levels of schemas:

1. **Internal Schema (Internal Level)**:
    - Describes how data is physically stored and accessed, including details like file structures, indexes, and access paths.
    - Uses a **physical data model** to define these storage details.
2. **Conceptual Schema (Conceptual Level)**:
    - Represents the entire database's structure and constraints as understood by the community of users.
    - Uses a **conceptual or implementation data model** (e.g., entity-relationship model) to define entities, relationships, and constraints.
    - Provides a unified view of the database without concern for physical storage details, ensuring data consistency across the organization.
3. **External Schemas (External Level)**:
    - Defines different user views, showing only the relevant data for each user or user group.
    - Typically uses the same data model as the conceptual schema but tailored to specific user needs.
    - Supports **multiple views** of the data, allowing different users to interact with the database in a way that suits their roles.

Mappings Between Schema Levels

Mappings are crucial to the Three-Schema Architecture:

- **External to Conceptual Mapping**: Translates user-specific views into the conceptual schema's terms.
- **Conceptual to Internal Mapping**: Converts conceptual-level requests into physical operations on the data.

These mappings allow programs to interact with the database at the external level while the DBMS manages the complexities of physical data storage. This separation ensures that changes in storage structure or data organization do not impact user applications, supporting program-data independence.

4. Define data independence. Explain types of data independence.

Data Independence refers to the ability to modify a database schema at one level without affecting the schema at the next higher level. This is a critical feature of database management systems, as it allows for flexibility and adaptability in database design and maintenance without disrupting existing applications or user views.

Types of Data Independence

- **Logical Data Independence**:
    - The ability to change the **conceptual schema** without requiring changes to the **external schemas** or application programs.
    - If you add a new attribute to a table or modify a relationship between tables in the conceptual schema, the external views and associated programs remain unaffected, as long as the changes don't alter the external schema.
    - It allows for the evolution of the database structure (e.g., adding new data, changing relationships) without impacting how users interact with the data or requiring changes to application code.
- **Physical Data Independence**:
    - The ability to change the **internal schema** (i.e., how data is physically stored) without altering the **conceptual schema** or higher levels.
    - Reorganizing file structures, creating new indexes, or changing storage devices can improve performance without altering the conceptual view or requiring changes to external views and applications.
    - This ensures that performance optimizations or changes in the storage hardware can be made without affecting the logical structure of the database or how users and applications access the data.
- When a schema at a lower level (internal or conceptual) is modified, only the mappings between this schema and the next higher level need to be updated.
- The higher-level schemas themselves remain unchanged, which means that application programs, which depend on the external schemas, continue to function without modification.

5. Explain database languages and Interfaces.

Database Languages

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
    - High-Level or Non-procedural Languages: These include the relational language SQL.
        - May be used in a standalone way or may be embedded in a programming language.
    - Low Level or Procedural Languages:
        - These must be embedded in a programming language.
- **Data Definition Language (DDL):**
    - Used by the DBA and database designers to specify the conceptual schema of a database.
    - In many DBMSs, the DDL is also used to define internal and external schemas (views).
    - In some DBMSs, separate **storage definition language (SDL)** and **view definition language (VDL)** are used to define internal and external schemas.
        - SDL is typically realized via DBMS commands provided to the DBA and database designers.
- **Data Manipulation Language (DML):**
    - Used to specify database retrievals and updates.
    - DML commands (data sublanguage) can be *embedded* in a general-purpose programming language (host language), such as COBOL, C, C++, or Java.
        - A library of functions can also be provided to access the DBMS from a programming language.
    - Alternatively, stand-alone DML commands can be applied directly (called a *query language*).
- **Types of DML:**
    - **High Level or Non-procedural Language:**
        - For example, the SQL relational language
        - Are "set"-oriented and specify what data to retrieve rather than how to retrieve it.
        - Also called **declarative** languages.
    - **Low Level or Procedural Language:**
        - Retrieve data one record-at-a-time;
        - Constructs such as looping are needed to retrieve multiple records, along with positioning pointers.

DBMS Interfaces

- Stand-alone query language interfaces
    - Example: Entering SQL queries at the DBMS interactive SQL interface (e.g. SQL*Plus in ORACLE)
- Programmer interfaces for embedding DML in programming languages.
- User-friendly interfaces
    - Menu-based, forms-based, graphics-based, etc.
- Programmer interfaces for embedding DML in a programming language:
    - **Embedded Approach**: e.g. embedded SQL (for C, C++, etc.), SQLJ (for Java)
    - **Procedure Call Approach**: e.g. JDBC for Java, ODBC for other programming languages
    - **Database Programming Language Approach**: e.g. ORACLE has PL/SQL, a programming language based on SQL; language incorporates SQL and its data types as integral components.
    - Menu-based, popular for browsing on the web.
    - Forms-based, designed for naïve users.
    - Graphics-based
    - Natural language: requests in written English
    - Combinations of the above. For example, both menus and forms are used extensively in Web interfaces.
    - Speech as Input and Output
    - Parametric interfaces, e.g., bank tellers using function keys.
- Interfaces for the DBA:
    - Creating user accounts, granting authorizations
    - Setting system parameters
    - Changing schemas or access paths

6. Discuss the various component modules of a DBMS and their interaction with a neat diagram.

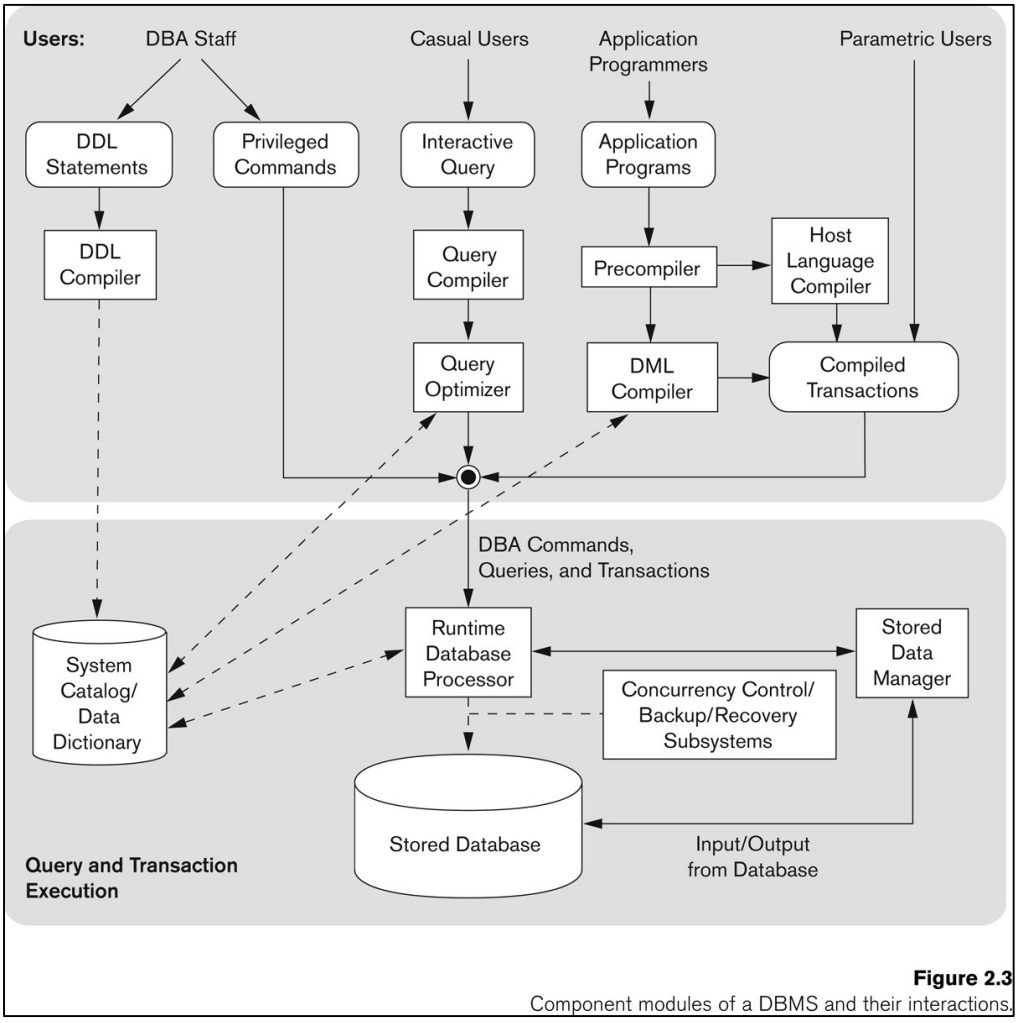


Figure 2.3
Component modules of a DBMS and their interactions.

7. Draw ER diagram of Company Database.

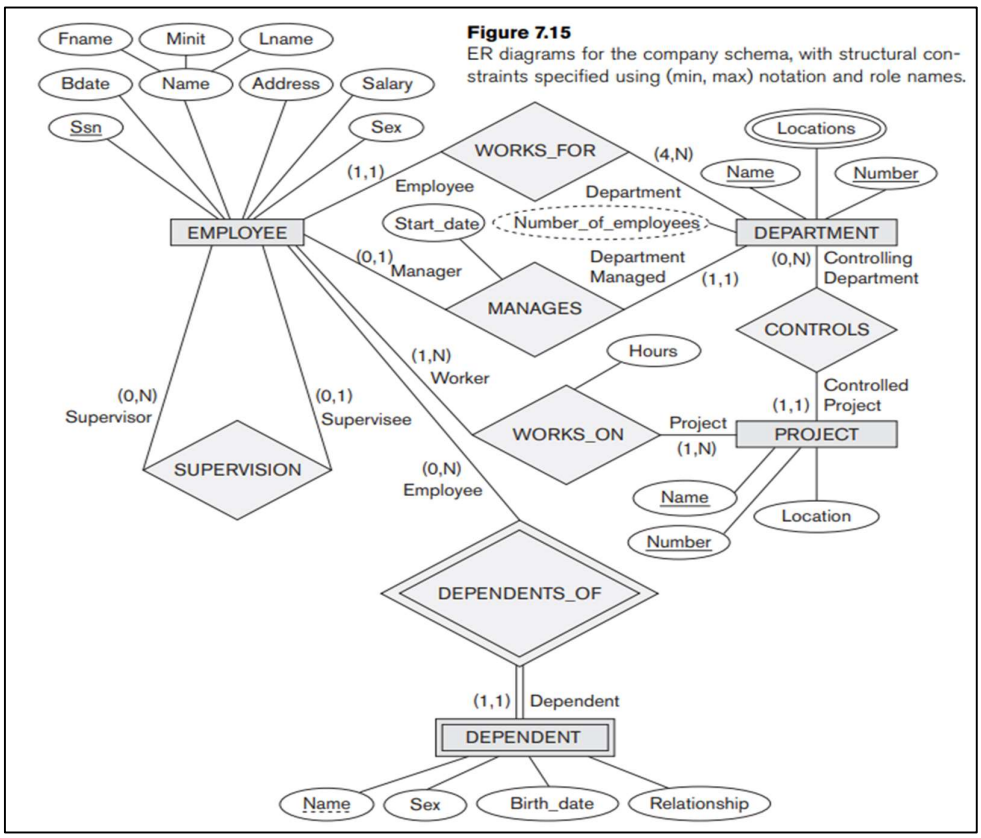


Figure 7.15
ER diagrams for the company schema, with structural constraints specified using (min, max) notation and role names.

8. Draw ER diagram of Bank Database

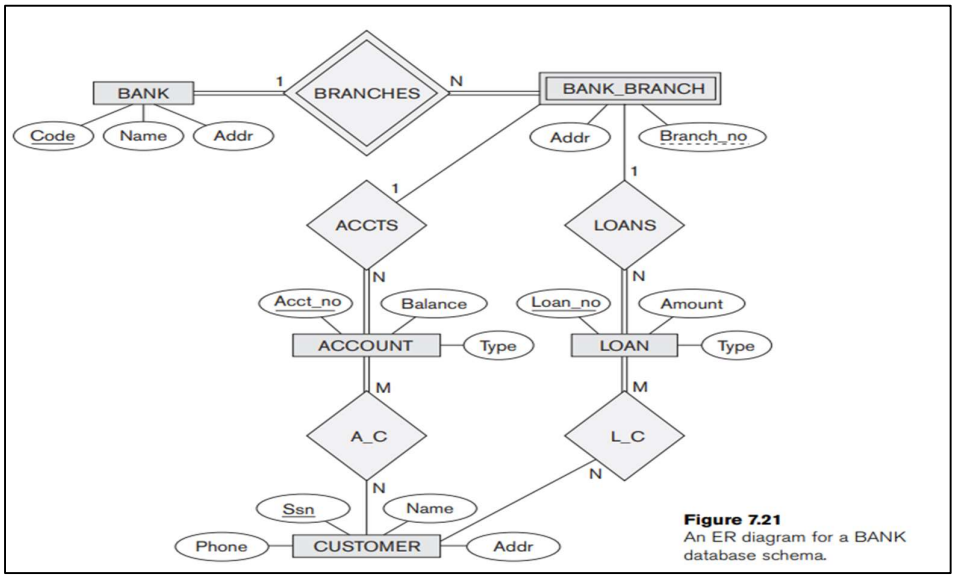


Figure 7.21
An ER diagram for a BANK database schema.

9. Draw ER diagram for Movies Database Schema.

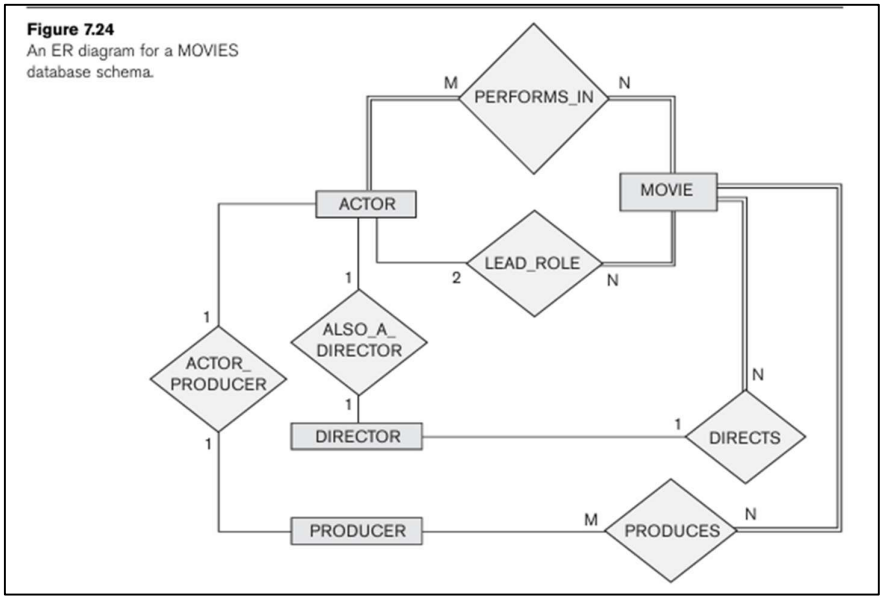
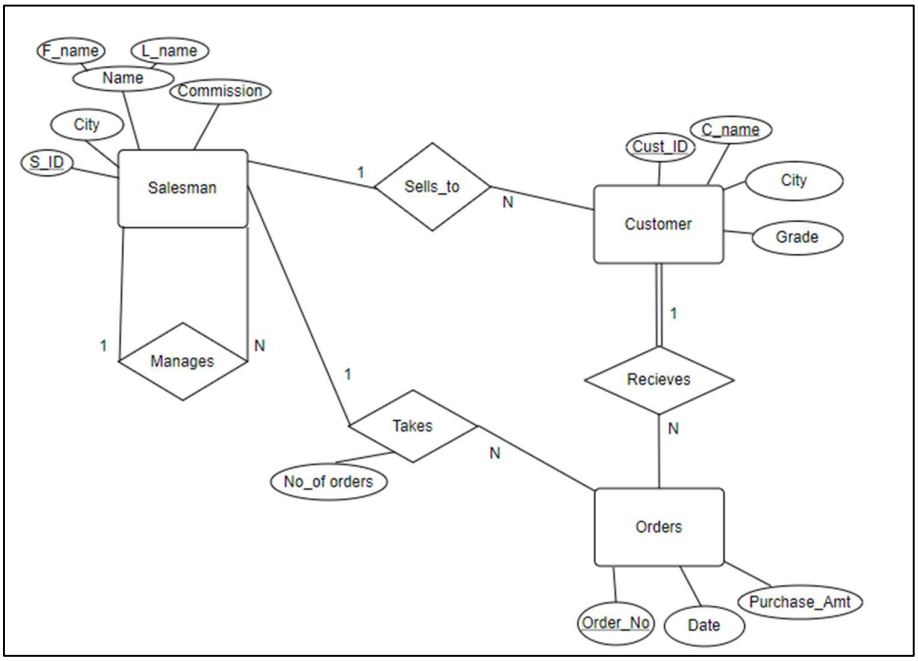


Figure 7.24
An ER diagram for a MOVIES database schema.

10. Draw ER diagram for Order Database Schema.

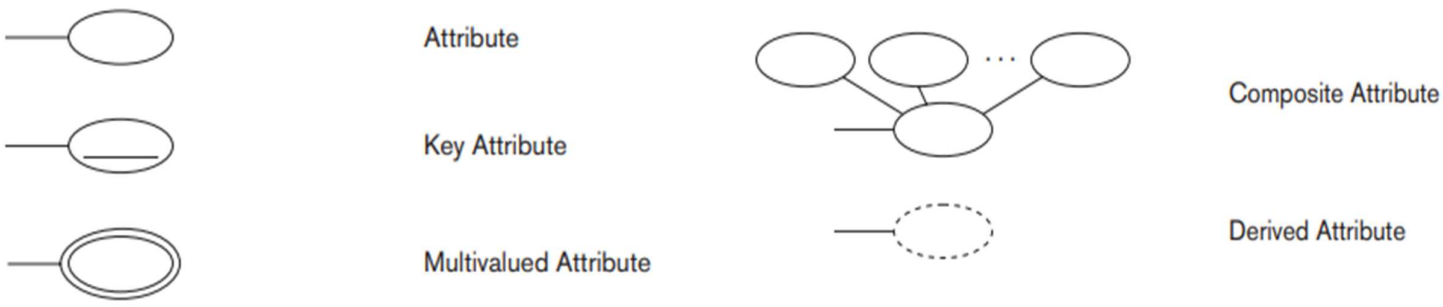


11. Define an entity and an attribute. Explain the different types of attributes that occur in the ER model, with an example.

- **Entity**: An entity is a real-world object or concept that is distinguishable from other objects or concepts. In the context of an Entity-Relationship (ER) model, an entity represents a set of objects or things that share the same characteristics or properties. For example, in a university database, "Student" and "Course" are entities.
- **Attribute**: An attribute is a property or characteristic of an entity. Attributes describe the data that is associated with an entity. For example, the "Student" entity may have attributes like "Student_ID," "Name," "Date_of_Birth," and "Email."

Types of Attributes in the ER Model

- **Simple (Atomic) Attribute**:
  - **Definition**: An attribute that cannot be further divided into smaller components.
  - **Example**: "Student_ID" in a "Student" entity is a simple attribute because it represents a unique identifier that cannot be broken down further.
- **Composite Attribute**:
  - **Definition**: An attribute that can be subdivided into smaller, more specific attributes.
  - **Example**: "Name" can be a composite attribute composed of "First_Name" and "Last_Name."
- **Single-Valued Attribute**:
  - **Definition**: An attribute that holds only one value for a particular entity instance.
  - **Example**: "Date_of_Birth" is a single-valued attribute for the "Student" entity because each student has only one birth date.
- **Multi-Valued Attribute**:
  - **Definition**: An attribute that can hold multiple values for a single entity instance.
  - **Example**: "Phone_Number" for the "Student" entity can be a multi-valued attribute if a student has more than one phone number.
- **Derived Attribute**:
  - **Definition**: An attribute whose value can be derived from other attributes in the database.
  - **Example**: "Age" could be a derived attribute calculated from the "Date_of_Birth" attribute.
- **Key Attribute**:
  - **Definition**: An attribute that uniquely identifies an entity instance within an entity set.
  - **Example**: "Student_ID" is a key attribute for the "Student" entity because it uniquely identifies each student.



12. Explain the Structural Constraints.

Structural Constraints in the Entity-Relationship (ER) model define the rules that limit or govern the relationships between entities. They specify how many instances of an entity can be associated with instances of another entity in a relationship. There are two main types of structural constraints:

Cardinality Constraints

Cardinality constraints define the number of instances of one entity that can or must be associated with each instance of another entity in a relationship.

- **One-to-One (1:1)**:
  - Each instance of one entity is associated with at most one instance of another entity, and vice versa.
  - **Example**: In a database of employees and company cars, each employee might be assigned exactly one company car, and each car is assigned to exactly one employee.
- **One-to-Many (1:N)**:
  - An instance of one entity can be associated with multiple instances of another entity, but each instance of the second entity is associated with only one instance of the first.
  - **Example**: A department may have many employees, but each employee belongs to only one department.
- **Many-to-One (N:1)**:
  - Multiple instances of one entity can be associated with a single instance of another entity.
  - **Example**: Many students may be enrolled in the same course, but each course is associated with only one department.
- **Many-to-Many (M:N)**:
  - Instances of one entity can be associated with multiple instances of another entity, and vice versa.
  - **Example**: Students can enroll in multiple courses, and each course can have multiple students.

Participation Constraints

Participation constraints specify whether all or only some entity instances participate in a relationship.

- **Total Participation**:
  - Every instance of an entity must participate in the relationship. This is also known as a mandatory participation constraint.
  - **Example**: In a database of marriages, every spouse must be part of the "Marriage" relationship, meaning total participation is required for the "Spouse" entity.
- **Partial Participation**:
  - Only some instances of an entity participate in the relationship. This is also known as optional participation.
  - **Example**: In an employee database, some employees might not be assigned a company car, so the "Employee" entity has partial participation in the "Assigned_Car" relationship.

13. What is the cardinality ratio? Explain the possible cardinality ratios for binary relationship types with an example.

The cardinality ratio in a database or data modeling context describes the number of instances of one entity that can or must be associated with instances of another entity in a relationship. It is a key aspect of defining how entities interact with each other.

For binary relationships (relationships between two entities), the possible cardinality ratios are:

- **One-to-One (1:1)**
    - In this relationship, each instance of Entity A can be associated with only one instance of Entity B, and vice versa.
    - **Example**: Consider a database of employees and their company cars. If each employee is assigned exactly one company car, and each company car is assigned to exactly one employee, this is a one-to-one relationship.
- **One-to-Many (1:N)**
    - In this relationship, one instance of Entity A can be associated with multiple instances of Entity B, but each instance of Entity B is associated with only one instance of Entity A.
    - **Example**: A department and its employees. Each department can have many employees, but each employee works in only one department.
- **Many-to-One (N:1)**
    - This is the inverse of the one-to-many relationship. Here, multiple instances of Entity A can be associated with one instance of Entity B, but each instance of Entity B can be associated with only one instance of Entity A.
    - **Example**: Many employees might report to one manager. Each manager supervises multiple employees, but each employee reports to only one manager.
- **Many-to-Many (M:N)**
    - In this relationship, instances of Entity A can be associated with multiple instances of Entity B, and instances of Entity B can be associated with multiple instances of Entity A.
    - **Example**: Students and courses. A student can enroll in multiple courses, and a course can have multiple students enrolled.

Module 2

1. Explain primary key , referential integrity and foreign key concepts with the specific example.

Primary Key:

A **primary key** is a unique identifier for a record in a database table. Each table can have only one primary key, and it ensures that no two rows in the table have the same value for the primary key column(s). The primary key can consist of one or more columns.

Example: Consider a Students table:

StudentID	FirstName	LastName	DateOfBirth
101	John	Doe	2000-01-01
102	Jane	Smith	1999-05-15
103	Jim	Brown	2001-07-20

Here, StudentID is the primary key. Each student has a unique StudentID, which can be used to identify them in the table.

Foreign Key:

A **foreign key** is a column or set of columns in a table that creates a link between the data in two tables. The foreign key in one table points to the primary key in another table, establishing a relationship between the two tables.

Example: Consider a Courses table and a StudentCourses table:

- Courses table:

CourseID	CourseName
CSE101	Data Structures
CSE102	Algorithms
CSE103	Databases

- StudentCourses table:

StudentCourseID	StudentID	CourseID
1	101	CSE101
2	102	CSE102
3	103	CSE103

In the StudentCourses table:

- StudentID is a foreign key that references the StudentID in the Students table.
- CourseID is a foreign key that references the CourseID in the Courses table.

Referential Integrity:

Referential integrity is a concept that ensures that the relationships between tables remain consistent. It means that the foreign key values in one table must correspond to valid primary key values in the related table. If referential integrity is enforced, the database will not allow you to add, delete, or update records that would violate these relationships.

Example: In the StudentCourses table, if you try to insert a record with a StudentID that does not exist in the Students table or a CourseID that does not exist in the Courses table, the database will prevent this action to maintain referential integrity.

For instance, if you attempt to insert a record with StudentID = 104 and CourseID = CSE101, but there is no student with StudentID = 104 in the Students table, the insertion will fail, preserving the integrity of the relationship between students and their courses.

2. Explain union, Intersection and Minus Operations of Relational algebra with examples

Consider these Tables for the examples:

STUDENT	
First	**Last**
Aisha	Arora
Bikash	Dutta
Makku	Singh
Raju	Chopra

FACULTY	
FirstN	**LastN**
Raj	Kumar
Honey	Chand
Makku	Singh
Karan	Rao

Union Operation

- It performs binary union between two given relations and is defined as:-
  - $R \cup S = \{ t \mid t \in R \text{ or } t \in S\}$, Where **R** and **S** are database relations.
- For a union operation, the following conditions must hold:-
  - R and S must have the same number of attributes.
  - Attribute domains must be compatible.
  - Duplicate tuples are eliminated automatically.

Intersection Operation

- It performs a binary intersection between two given relations and is defined as:-
  - $R \cap S = \{ t \mid t \in R \text{ and } t \in S\}$, Where **R** and **S** are database relations.
- The result of Intersection operation, which is denoted by $R \cap S$, is a relation that basically includes all the tuples that are present in both R and S.

Minus Operation

- The result of set difference operation is tuples, which are present in one relation but are not in the second relation.
  - Notation: $R - S$
- Finds all the tuples that are present in **R** but not in **S**.

Examples:

Student ∪ Faculty

First	Last
Aisha	Arora
Bikash	Dutta
Makku	Singh
Raju	Chopra
Raj	Kumar
Honey	Chand
Karan	Rao

Student ∩ Faculty

First	Last
Makku	Singh

Student – Faculty

First	Last
Aisha	Arora
Bikash	Dutta
Raju	Chopra

3. Describe Selection and Projector Operator in Relational Algebra and mention the difference between them with examples.

Select Operation (σ)

- It selects tuples that satisfy the given predicate from a relation.
- Notation − $\sigma_p(r)$
- Where σ stands for selection predicate and r stands for relation. p is prepositional logic formula which may use connectors like and, or, and not. These terms may use relational operators like − =, ≠, ≥, < , >, ≤.

Example:

$\sigma_{subject}$ = "database"(Books)

Output − Selects tuples from books where subject is 'database'.

$\sigma_{subject}$ = "database" and price = "450"(Books)

Output − Selects tuples from books where subject is 'database' and 'price' is 450.

Project Operation (∏)

- It projects column(s) that satisfy a given predicate.
- Notation − $\prod_{A1, A2, An}$ (r)
- Where A1, A2 , An are attribute names of relation r.
- Duplicate rows are automatically eliminated, as relation is a set.

Example:

$\prod_{subject, author}$ (Books)

Selects and projects columns named as subject and author from the relation Books.

Differences between Select and Project:

Feature	**Select (σ)**	**Project (π)**
Function	Filters rows	Selects columns
Symbol	σ	π
Impact on Columns	No direct impact	Reduces columns
Indirect impact on Columns	Yes, if selection condition involves specific columns	No
Subset selection type	Horizontal	Vertical

4. **Develop the following queries in Using Relational Algebra.**

i) Find the names of all the employees whose salary is greater than 30000.

- Command: $\pi_{(name)}$ ($\sigma_{(salary > 30000)}$(Employee))
- σ(salary > 30000)(Employee): This part selects rows from the Employee relation where the "salary" is greater than 30000.
- π(name) (...): This projects only the "name" column from the result of the selection operation.

ii) Retrieve the name and emp id of all employees.

- Command: $\pi_{(name, emp\_id)}$(Employee)
- π(name, emp_id)(Employee): This directly projects both "name" and "emp_id" columns from the Employee relation.

iii) Find the fname and lname of employees in department 4 that earn > 50000

- $\pi_{(fname, lname)}$ ($\sigma_{(department = 4 AND salary > 50000)}$(Employee))
- σ(department = 4 AND salary > 50000)(Employee): This selects rows where "department" is equal to 4 and "salary" is greater than 50000 from the Employee relation.
- π(fname, lname) (...): This projects only the "fname" and "lname" columns from the result of the selection operation.

5. Explain different types of Joins in SQL

The SQL Join clause is used to combine data from two or more tables in a database. When the related data is stored across multiple tables, joins help you to retrieve records combining the fields from these tables using their foreign keys.

Syntax:

```
SELECT column_name(s)
FROM table1
JOIN table2;
```

Types of joins in SQL

SQL provides various types of Joins that are categorized based on the way data across multiple tables are joined together. They are listed as follows −

Inner Join

An INNER JOIN is the default join which retrieves the intersection of two tables. It compares each row of the first table with each row of the second table. If the pairs of these rows satisfy the join-predicate, they are joined together.

Syntax:

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

EmpDetails

ID	Name	Salary
1	John	40000
2	Alex	25000
3	Simon	43000

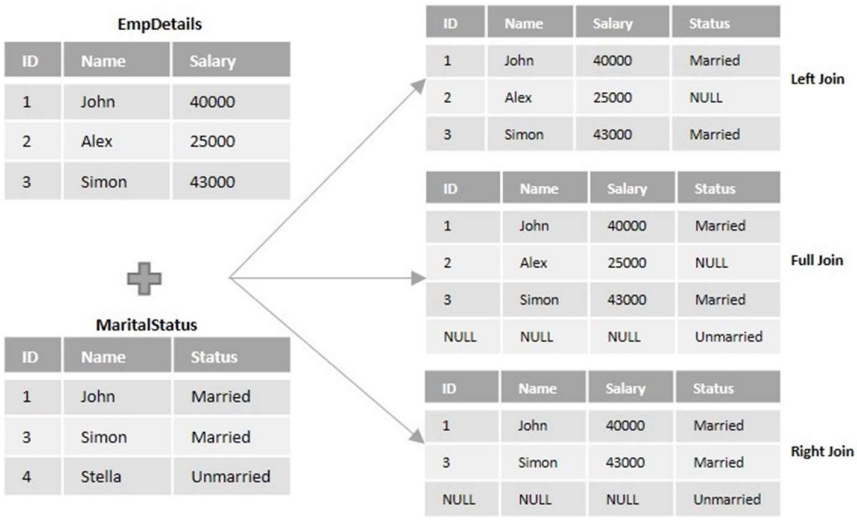
MaritalStatus

ID	Name	Status
1	John	Married
3	Simon	Married
4	Stella	Unmarried

ID	Name	Salary	Status
1	John	40000	Married
3	Simon	43000	Married

Outer Join

An Outer Join retrieves all the records in two tables even if there is no counterpart row of one table in another table, unlike Inner Join. Outer join is further divided into three subtypes - Left Join, Right Join and Full Join.



- LEFT JOIN – returns all rows from the left table, even if there are no matches in the right table.
  Syntax:

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

- RIGHT JOIN – returns all rows from the right table, even if there are no matches in the left table.
  Syntax:

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

- FULL JOIN – returns rows when there is a match in one of the tables.
  Syntax:

```
SELECT column_name(s)
FROM table1
FULL JOIN table2
ON table1.column_name = table2.column_name;
```

Other Joins

- SELF JOIN − is used to join a table to itself as if the table were two tables, temporarily renaming at least one table in the SQL statement.

Color	Name	Color
Blue	John	Red
Green	Alex	Blue
Red	Simon	Green

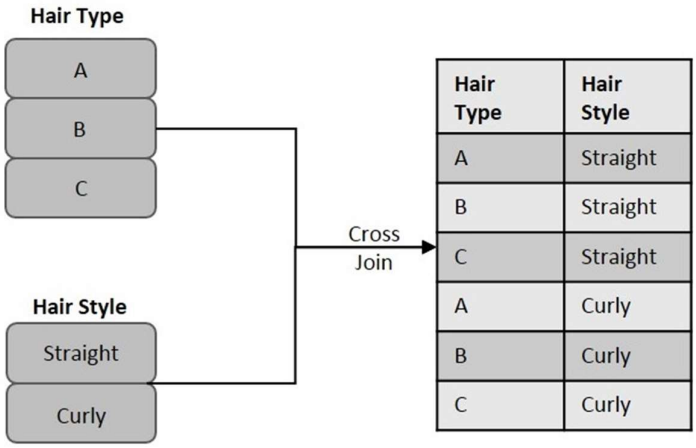
Color	Name	Color
Blue	John	Red
Green	Alex	Blue
Red	Simon	Green

Name	Secret_Santa
John	Simon
Alex	John
Simon	Alex

Syntax:

```
SELECT column_name(s)
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

- CROSS JOIN − returns the Cartesian product of the sets of records from the two or more joined tables.



Hair Type	Hair Style
A	Straight
B	Straight
C	Straight
A	Curly
B	Curly
C	Curly

Syntax:

```
SELECT column_name(s)
  FROM table1
  CROSS JOIN table2;
```

6. Describe the different Join Operators with examples.

Join is a combination of a Cartesian product followed by a selection process. A Join operation pairs two tuples from different relations, if and only if a given join condition is satisfied.

Theta (θ) Join

- Theta join combines tuples from different relations provided they satisfy the theta condition. The join condition is denoted by the symbol **θ**.
- Notation: R1 $\bowtie_\theta$ R2
- R1 and R2 are relations having attributes (A1, A2, .., An) and (B1, B2,.. ,Bn) such that the attributes don't have anything in common, that is R1 ∩ R2 = Φ.
- Theta join can use all kinds of comparison operators.

Equijoin

- When Theta join uses only equality comparison operator, it is said to be equijoin.

Natural Join (⋈)

- Natural join does not use any comparison operator. It does not concatenate the way a Cartesian product does.
- We can perform a Natural Join only if there is at least one common attribute that exists between two relations.
- In addition, the attributes must have the same name and domain.
- Natural join acts on those matching attributes where the values of attributes in both the relations are same.

Left Outer Join(R Left Outer Join S)

- All the tuples from the Left relation, R, are included in the resulting relation.
- If there are tuples in R without any matching tuple in the Right relation S, then the S-attributes of the resulting relation are made NULL.

Right Outer Join: (R Right Outer Join S)

- All the tuples from the Right relation, S, are included in the resulting relation.
- If there are tuples in S without any matching tuple in R, then the R-attributes of resulting relation are made NULL.

Full Outer Join: (R Full Outer Join S)

- All the tuples from both participating relations are included in the resulting relation.
- If there are no matching tuples for both relations, their respective unmatched attributes are made NULL.

7. Explain different steps of ER to Relational Mapping algorithm.

Step 1: Mapping of Regular Entity Types.

- For each regular (strong) entity type E in the ER schema, choose one of the key attributes of E as the primary key for R.
- If the chosen key of E is composite, the set of simple attributes will t form the primary key of R.

Step 2: Mapping of Weak Entity Types

- For each weak entity type W in the ER schema with owner entity type E, create a relation R & include all simple attributes of W as attributes of R.
- Also, include primary key relations as the foreign key of R.
- The primary key of R is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any.

Step 3: Mapping of Binary 1:1 Relation Types

- For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R.
- Choose one of the relation S and include a foreign key in S, the primary key of T. Choose an entity type with total participation in R in the role of S.

Step 4: Mapping of Binary 1:N Relationship Types.

- For each regular binary 1:N relationship type R, identify the relation S that represent the participating entity type at the N-side of the relationship type.
- Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R and include any simple attributes of the 1:N relation type as attributes of S

Step 5: Mapping of Binary M:N Relationship Types.

- For each regular binary M:N relationship type R, create a new relation S to represent R.
- Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S. Also include any simple attributes of the M:N relationship as attributes of S.

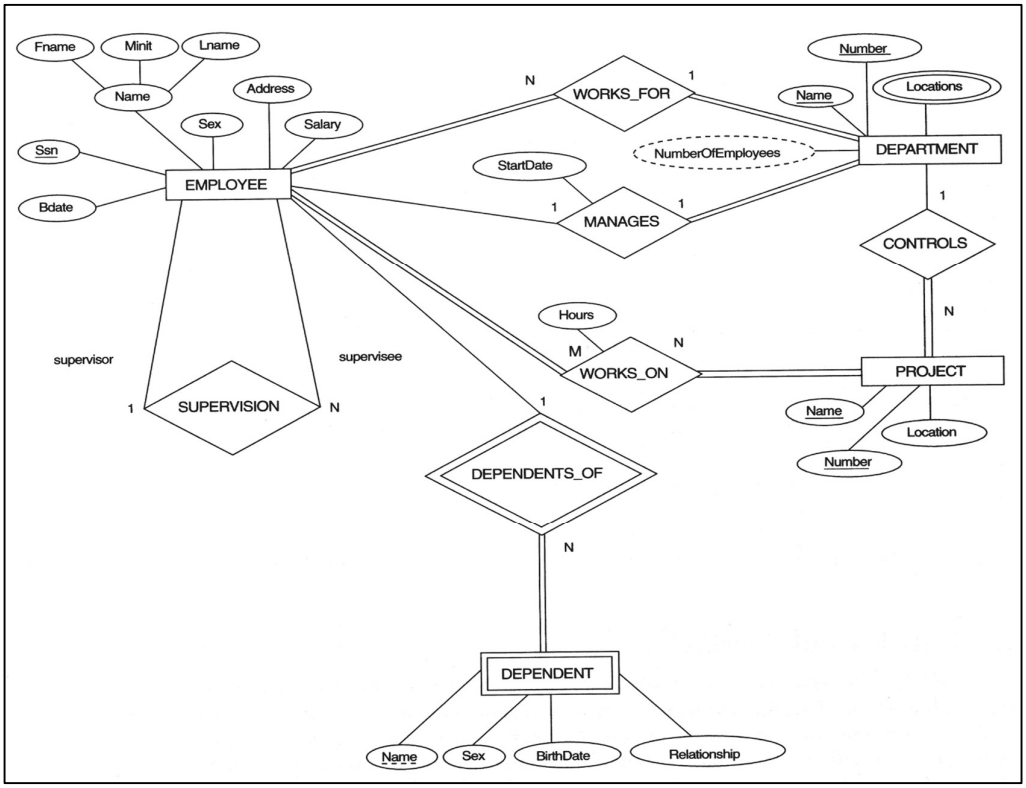
Step 6: Mapping of Multivalued attributes.

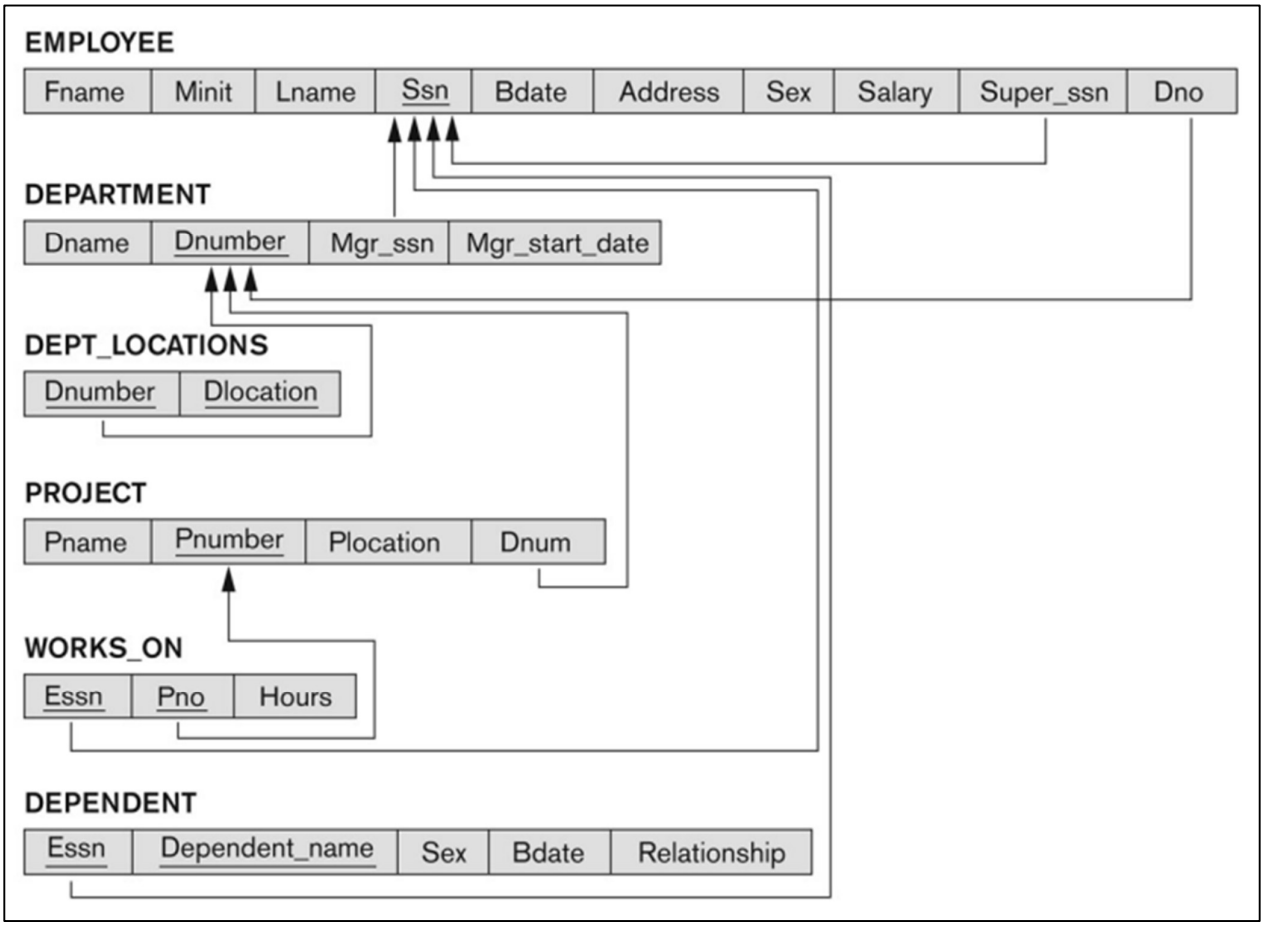
- For each multivalued attribute A, create a new relation R.
- This relation R will include an attribute corresponding to A, plus the primary key attribute K-as a foreign key in R-of the relation that represents the entity type of relationship type that has A as an attribute.

Step 7: Mapping of N-ary Relationship Types.

- For each n-ary relationship type R, where n>2, create a new relationship S to represent R.
- Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the n-ary relationship type as attributes of S.

8. Convert the following ER Diagram to Relational Model





Module 3

1. Explain the data types that are allowed for SQL attributes.

- **Integer (INT):**
  - **Description:** Stores whole numbers (positive, negative, or zero).
  - **Syntax:** `column_name INT`
  - **Example:**

```
CREATE TABLE customers (customer_id INT PRIMARY KEY, age INT );
```

- **Float:**
  - **Description:** Stores single-precision floating-point numbers, which can represent real numbers with decimal places.
  - **Syntax:** column_name FLOAT
  - **Example:**

```
CREATE TABLE products ( price FLOAT NOT NULL );
```

- **Char(n):**
  - **Description:** Stores a fixed-length character string, where 'n' specifies the maximum number of characters allowed. Pads shorter strings with spaces to reach the defined length.
  - **Syntax:** column_name CHAR(n)
  - **Example:**

```
CREATE TABLE states ( state_code CHAR(2) );
```

- **Varchar(n):**
  - **Description:** Stores a variable-length character string, with a maximum capacity of 'n' characters. Only uses the space needed for the actual data.
  - **Syntax:** column_name VARCHAR(n)
  - **Example:**

```
CREATE TABLE users (username VARCHAR(30) UNIQUE, email VARCHAR(25));
```

- **Date:**
  - **Description:** Stores only the year, month, and day information.
  - **Syntax:** column_name DATE
  - **Example:**

```
CREATE TABLE orders( order_id INT PRIMARY KEY, order_date DATE NOT
NULL);
```

- **Time:**
  - **Description:** Stores only the time component, including hours, minutes, and seconds (sometimes with fractions of a second).
  - **Syntax:** column_name TIME
  - **Example:**

```
CREATE TABLE flights( flight_id INT PRIMARY KEY, departure_time TIME NOT
NULL );
```

- **Datetime:**
  - **Description:** Combines both date and time information in a single column.
  - **Syntax:** column_name DATETIME
  - **Example:**

```
CREATE TABLE transactions ( transaction_id INT PRIMARY KEY, timestamp
DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP);
```

2. Outline the different constraints in SQL while creating table.

The constraints while creating a table in SQL are:

Primary Key

- The PRIMARY KEY constraint uniquely identifies each record in a table.
- Primary keys must contain UNIQUE values and cannot contain NULL values.
- A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

Syntax:

```
CREATE TABLE table_name ( attribute_name datatype PRIMARY KEY);
```

Foreign Key

- The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
- A FOREIGN KEY is a field in one table, that refers to the PRIMARY KEY in another table.
- The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent table.
- The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

Syntax:

```
CREATE TABLE table_name (attribute_name datatype FOREIGN KEY REFERENCES
referenced_table(referenced_primary_key));
```

Not Null

- By default, a column can hold NULL values.
- The NOT NULL constraint enforces a column to NOT accept NULL values.
- This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

Syntax:

```
CREATE TABLE table_name (attribute_name datatype NOT NULL);
```

Unique

- The UNIQUE constraint ensures that all values in a column are different.
- Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint.
- However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

Syntax:

```
CREATE TABLE table_name (attribute_name datatype UNIQUE);
```

3. Explain the schema change statements of SQL.

In SQL, schema changes are modifications made to the structure of your database. This can involve altering existing tables, views, indexes, or even creating or dropping them entirely. There are two primary statements used for schema changes:

- ALTER TABLE: This statement allows you to modify the definition of a table.
  - Add a new column: Specify the column name, data type, and any.
    ```
    ALTER TABLE customers ADD phone_number VARCHAR(15);
    ```
  - Modify an existing column: Change the data type, rename the column, or alter constraints.
    ```
    ALTER TABLE products MODIFY COLUMN price FLOAT;
    ```
  - Drop a column: Remove a column from the table.
    ```
    ALTER TABLE posts DROP COLUMN created_at;
    ```
- DROP TABLE: This permanently removes the specified table along with all its data, columns etc.
  - Some database systems allow specifying CASCADE with DROP TABLE to automatically drop dependent objects as well.
    ```
    DROP TABLE orders;
    ```

4. Write the SQL queries for the following:

a) Write the syntax to Create, Alter and Drop table.

- `CREATE TABLE table_name ( column1 data_type[constraint1, constraint2,...], column2 data_type[constraint1, constraint2...]);`
- `ALTER TABLE table_name ADD COLUMN new_column_name data_type [constraint1, constraint2, ...], OR ALTER COLUMN existing_column_name data_type [constraint1, constraint2, ...], OR DROP COLUMN column_to_drop;`
- `DROP TABLE table_name;`

b) Write SQL Query to Create Employee table with the following attributes: eid, ename and salary.

- `CREATE TABLE employee ( eid INT PRIMARY KEY, ename VARCHAR(50) NOT NULL, salary DECIMAL(10,2) NOT NULL );`

c) Alter table employee by adding one more attribute called address.

- `ALTER TABLE employee ADD COLUMN address VARCHAR(25);`

d) Give syntax to drop table employee and drop column salary.

- `DROP TABLE employee;`
- `ALTER TABLE employee DROP COLUMN salary;`

5. **Write the SQL queries for the following:**

a) **Retrieve the birth date and address of employee whose employee id is 10.**

- `SELECT dob, address FROM employee WHERE eid = 10;`

b) **Retrieve the name and address of all employees who work for 'Research' department.**

- `SELECT ename, address FROM employee WHERE department = 'Research';`

c) **Retrieve all employees in department 5 whose salary is between 30000 and 40000**

- `SELECT * FROM employee WHERE department = 5 AND salary BETWEEN 30000 AND 40000;`

d) **Retrieve distinct salaries of employees.**

- `SELECT DISTINCT salary FROM employee;`

6. **Explain the ALTER TABLE command. Explain how the new constraint can be added and also an existing constraint can be removed using suitable examples.**

The ALTER TABLE command in SQL is a powerful tool for modifying the structure of existing tables. It allows you to add new features, change existing definitions, or remove unnecessary elements without completely recreating the table.

- You can use ALTER TABLE to add various constraints to existing columns in a table. These constraints enforce data integrity and ensure the validity of your data.
  - `ALTER TABLE customers ADD CONSTRAINT pk_customers PRIMARY KEY (customer_id);`
  - `ALTER TABLE orders ALTER COLUMN order_date SET NOT NULL;`
  - `ALTER TABLE orders ADD CONSTRAINT fk_orders_customer FOREIGN KEY (customer_id) REFERENCES customers(customer_id);`
- You can also use ALTER TABLE to remove constraints that are no longer needed.
  - `ALTER TABLE products DROP CONSTRAINT unique_product_name;`

7. Explain INSERT, DELETE, UPDATE statements in SQL taking suitable examples.

In SQL, modifying data within tables is achieved through three primary statements: INSERT, UPDATE, and DELETE.

INSERT

- **Purpose:** Used to add new rows (records) to an existing table.
- **Syntax:**

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

- **Example:**

```
INSERT INTO customers (customer_name, email, phone_number) VALUES
('John Doe','john.doe@example.com', '123-456-7890');
```

UPDATE

- **Purpose:** Used to modify existing data within a table based on a specific condition.
- **Syntax:**

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

- **Example:**

```
UPDATE products SET price = price * 1.1  -- Increase price by 10%
WHERE category = 'Electronics';
```

DELETE

- **Purpose:** Used to remove rows from a table based on a specific condition.
- **Syntax:**

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

- **Example:**

```
DELETE FROM orders WHERE order_status = 'Cancelled';
```

8. Explain the aggregate functions in SQL? Explain with examples.

Aggregate functions in SQL perform calculations on groups of data in a table and return a single summarized value. They are essential for condensing large datasets and generating meaningful insights.

COUNT(*)

- **Purpose:** Counts the number of rows in a table or the number of non-null values in a specific column.
- **Example:**

```
SELECT COUNT(*) FROM customers;  -- Counts all customer records
```

SUM(column_name)

- **Purpose:** Calculates the sum of values in a numeric column.
- **Example:**

```
SELECT SUM(salary) FROM employees;
```

AVG(column_name)

- **Purpose:** Calculates the average of values in a numeric column. Ignores null values by default.
- **Example:**

```
SELECT AVG(order_total) FROM orders;
```

MIN(column_name)

- **Purpose:** Returns the minimum value in a column.
- **Example:**

```
SELECT MIN(price) FROM products;
```

MAX(column_name)

- **Purpose:** Returns the maximum value in a column.
- **Example:**

```
SELECT MAX(score) FROM exams;
```

9. Explain the command used for ordering the query results? Explain with the syntax and an example

ORDER BY sorts the retrieved data based on columns in ascending (default) or descending order.

Syntax:

```
SELECT column1, column2, ... FROM table_name [WHERE condition] ORDER BY
column_name ASC|DESC [, column_name2 ASC|DESC, ...];
```

Explanation:

column_name: The name of the column you want to sort by.

ASC: Sorts the data in ascending order This is the default order if not specified.

DESC: Sorts the data in descending order

Example:

```
SELECT name, age FROM customers ORDER BY age DESC;
```

10. Write SQL queries for following:

Student(Enrno, name, courseId, emailId, cellno)
Course(courseId, course_nm, duration)

a) Add a column city in student table.

- `ALTER TABLE student ADD COLUMN city VARCHAR(50);`

b) Find out list of students who have enrolled in "computer" course.

- `SELECT name FROM student s INNER JOIN course c ON s.courseId c.courseId WHERE c.course_nm = 'computer';`

c) List name of all courses with their duration.

- `SELECT course_nm, duration FROM course;`

d) List name of all students start with „a".

- `SELECT name FROM student WHERE name LIKE 'a%';`

e) List email Id and cell no of all mechanical engineering students

- `SELECT emailId, cellno FROM student s INNER JOIN course c ON s.courseId = c.courseId WHERE c.course_nm = 'mechanical';`

11. Explain Group by and having clause in SQL with an example.

GROUP BY Clause

- **Purpose:** Used in conjunction with aggregate functions to group rows based on a specific column before performing the aggregation.
- **Example:**

```
SELECT department, COUNT(*) AS employee_count
FROM employees
GROUP BY department;
```

HAVING Clause (Optional)

- **Purpose:** Used with aggregate functions after the GROUP BY clause to filter groups based on a condition on the aggregate values.
- **Example:**

```
SELECT department, AVG(salary) AS average_salary
FROM employees
GROUP BY department
HAVING AVG(salary) > 50000;
```

12. Explain Views in SQL. Give the syntax to create and drop views.

In SQL, views act as virtual tables that provide a customized view of the underlying data stored in one or more base tables. They offer several advantages:

- Data Abstraction: Views can simplify complex queries by hiding the underlying table structure and join logic from the user. Users can interact with the view as if it were a regular table.
- Security: Views can restrict access to sensitive data by limiting the columns or rows visible through the view.
- Data Focus: Views can present a focused subset of data from multiple tables, making it easier for users to retrieve the information they need.

Syntax:

- `CREATE VIEW view_name AS SELECT column1, column2, ... FROM table_name1 [JOIN table_name2 ON join_condition ...] [WHERE condition];`
- `DROP VIEW view_name;`

13. Consider the below table:
Orders(ord_no , purch_amt, ord_date , customer_id, salesman_id)

a) Write a SQL query to calculate total purchase amount of all orders.

- `SELECT SUM(purch_amt) AS total_purchase_amount FROM Orders;`

b) Write a SQL query to calculate the average purchase amount of all orders.

- `SELECT AVG(purch_amt) AS average_purchase_amount FROM Orders;`

c) Write a SQL query that counts the number of unique salespeople.

- `SELECT COUNT(DISTINCT salesman_id) AS unique_salespeople FROM Orders;`

d) Write a SQL query to find the maximum and minimum purchase amount.

- `SELECT MAX(purch_amt) AS max_purchase, MIN(purch_amt) AS min_purchase FROM Orders;`

14. Develop the SQL queries for the following:

a) Retrieve the birth date and address of employee whose employee id is 10.

- `SELECT dob, address FROM employee WHERE eid = 10;`

b) Retrieve the name and address of all employees who work for 'Research' department.

- `SELECT ename, address FROM employee WHERE department = 'Research';`

c) Retrieve all employees in department 5 whose salary is between 30000 and 40000.

- `SELECT * FROM employee WHERE department = 5 AND salary BETWEEN 30000 AND 40000;`

d) Retrieve distinct salaries of employees.

- `SELECT DISTINCT salary FROM employee;`

Module 4

1. Construct Functional Dependency. Give Inference rules for Functional Dependencies.

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

$X \rightarrow Y$

The left side of FD is known as a determinant, the right side of the production is known as a dependent.
For example, assume we have an employee table with attributes:
Emp_Id, Emp_Name, Emp_Address
Here Emp_Id attribute can uniquely identify the Emp_Name attribute of employee table because if we know the Emp_Id, we can tell that employee name associated with it.
Functional dependency can be written as:

Emp_Id → Emp_Name

We can say that Emp_Name is functionally dependent on Emp_Id.
Functional Dependencies:

- Are used to specify formal measures of the "goodness" of relational designs .
- And keys are used to define normal forms for relations.
- Are constraints that are derived from the meaning and interrelationships of the data attributes.

The Inference Rules for Functional Dependencies are:

- IR1: Reflexive Rule: If Y subset-of X, then X -> Y
- IR2: Augmentation Rule: If X -> Y, then XZ -> YZ
- IR3: Transitive Rule: If X -> Y and Y -> Z, then X -> Z
- IR4: Decomposition Rule: If X -> YZ, then X -> Y and X -> Z
- IR5: Union Rule: If X -> Y and X -> Z, then X -> YZ
- IR6: Psuedotransitivity Rule: If X -> Y and WY -> Z, then WX -> Z

The rules IR1, IR2 and IR3 are known as Armstrong's Inference Rules, they are the basic and complete rules on which IR4, IR5 and IR6 are derived from.

2. Explain 1NF with example.

A table is considered to be in 1NF if all the fields contain only scalar/atomic values (as opposed to list of values).
The First Normal Form disallows:

- composite attributes
- multivalued attributes
- nested relations; attributes whose values for an individual tuple are non-atomic

To convert a given Schema into 1NF:

- Place all items that appear in the repeating group on a new table.
- Designate a primary key for each new table produced.
- Duplicate in the new table the primary key of the table from which the repeating group was extracted or vice versa.

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

Not in 1NF

Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

Converted to 1NF

3. Explain 2NF with example.

A relation schema R is in second normal form (2NF) if every non-prime attribute A in R is fully functionally dependent on the primary key. R can be decomposed into 2NF relations through 2NF normalization.

For a table to be in 2NF, there are two requirements:

- The database is in first normal form.
- All non-key attributes in the table must be functionally dependent on the entire primary key.

To convert a given table to 2NF:

- If a data item is fully functionally dependent on only a part of the primary key, move that data item and that part of the primary key to a new table.
- If other data items are functionally dependent on the same part of the key, place them in the new table also.
- Make the partial primary key copied from the original table the primary key for the new table. Place all items that appear in the repeating group on a new table.

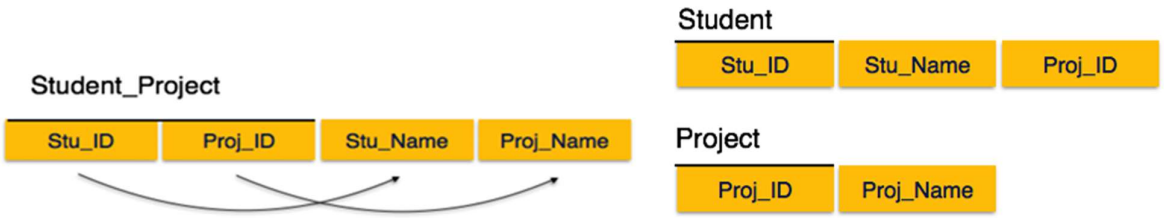
Example 1 (Convert to 2NF)

Old Scheme → {Title, PubId, AuId , Price, AuAddress}

New Scheme → {Title, PubId, AuId , Price}

New Scheme → {AuId , AuAddress}

Example 2



We see here in Student_Project relation that the prime key attributes are Stu_ID and Proj_ID. According to the rule, non-key attributes, i.e. Stu_Name and Proj_Name must be dependent upon both and not on any of the prime key attributes individually. But we find that Stu_Name can be identified by Stu_ID and Proj_Name can be identified by Proj_ID independently. This is called partial dependency, which is not allowed in Second Normal Form. We broke the relation in two. So there exists no partial dependency.

4. Explain 3NF and BCNF with example.

3NF

A relation schema R is in third normal form (3NF) if it is in 2NF, and no non-prime attribute A in R is transitively dependent on the primary key. R can be decomposed through 3NF normalization.

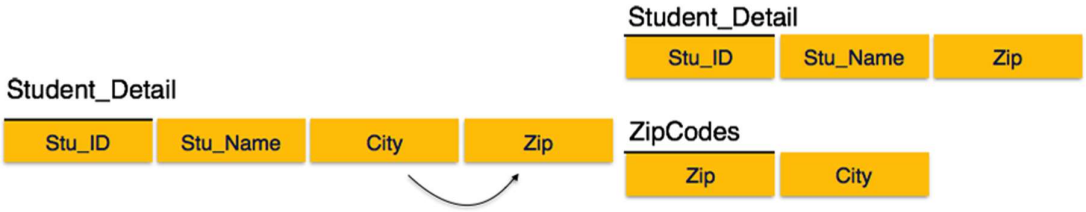
Transitive functional dependency: a FD X -> Z that can be derived from two FDs X -> Y and Y -> Z

For a table to be in 3NF, there are two requirements:

- The table should be second normal form.
- No attribute is transitively dependent on the primary key

To convert a given schema into 3NF:

- Move all items involved in transitive dependencies to a new entity.
- Identify the primary key for the new entity.
- Place the primary key for the new entity as a foreign key on the original entity.



BCNF

A relation schema R is in Boyce-Codd Normal Form (BCNF) if whenever an FD X -> A holds in R, then X is a super key of R. There exist relations that are in 3NF but not in BCNF.

In the above image, Stu_ID is the super-key in the relation Student_Detail and Zip is the super-key in the relation ZipCodes. So, Stu_ID → Stu_Name, Zip and Zip → City Which confirms that both the relations are in BCNF.

Third normal form and BCNF are not same if the following conditions are true:

- The table has two or more candidate key.
- At least two of the candidate keys are composed of more than one attribute.
- The keys are not disjointed i.e. The composite candidate keys share some attributes.

To convert a schema into BCNF:

- Place the two candidate primary keys in separate entities.
- Place each of the remaining data items in one of the resulting entities according to its dependency on the primary key.

Example 1 (Convert to BCNF)

Old Scheme → {City, Street, ZipCode }

New Scheme1 → {ZipCode, Street}

New Scheme2 → {City, Street}

Loss of relation {ZipCode} → {City}

Alternate New Scheme1 → {ZipCode, Street }

Alternate New Scheme2 → {ZipCode, City}

5. **Consider the below Relation**

   R{City, Street, HouseNumber, HouseColor, CityPopulation}

   Assume key as {City, Street, HouseNumber}

   The Dependencies are:

   {City, Street, HouseNumber} →{HouseColor}

   {City} →{CityPopulation}

Check whether the given R is in 2NF? If not convert into 2NF.

Scheme → {City, Street, HouseNumber, HouseColor, CityPopulation}

Key →{City, Street, HouseNumber}

{City, Street, HouseNumber} →{HouseColor}

{City} → {CityPopulation}

CityPopulation does not belong to any key.

CityPopulation is functionally dependent on the City which is a proper subset of the key

Old Scheme → {City , Street, HouseNumber, HouseColor, CityPopulation}
New Scheme → {City, Street, HouseNumber, HouseColor}
New Scheme → {City, CityPopulation}

6. **Consider the relation**

   Emp-Proj ={SSN, Pnumber, Hours, Ename, Pname, Plocation}

   Assume {SSN, Pnumber } as Primary key

   The dependencies are:

   {SSN, Pnumber}->Hours

   SSN->Ename

   Pnumber->{Pname, Plocation}

Normalize the above relation to 3NF

Old Scheme →{SSN, Pnumber, Hours, Ename, Pname, Plocation}
New Scheme →{SSN, Pnumber, Hours}
New Scheme → {SSN, Ename}
New Scheme → {Pnumber, Pname, Plocation}

7. **Consider the following relation**

   R {Title, PubId, AuId, Price, AuAddress}

   Assume primary key as {Title, PubId, AuId}

   The Dependencies are

   {Title, PubId, AuID} → {Price}

   {AuID} → {AuAddress}

Check whether the given R is in 2NF? If not convert into 2NF.

Scheme → {Title, PubId, AuId, Price, AuAddress}

Key → {Title, PubId, AuId}

{Title, PubId, AuID} → {Price}

{AuID} → {AuAddress}

AuAddress does not belong to a key

AuAddress functionally depends on AuId which is a subset of a key

Old Scheme → {Title, PubId, AuId, Price, AuAddress}
New Scheme → {Title, PubId, AuId, Price}
New Scheme → {AuId, AuAddress}

8. **Consider the following relation**
   R {Studio, StudioCity, CityTemp}
   Assum e Primary Key as {Studio}
   The Dependencies are:
   {Studio} → {StudioCity}
   {StudioCity} → {CityTemp}

Check whether the given R is in 3NF? If not convert into 3NF

Scheme → {Studio, StudioCity, CityTemp}
Primary Key → {Studio}
{Studio} →{StudioCity}
{StudioCity} → {CityTemp}
{Studio} →{CityTemp}
Both StudioCity and CityTemp depend on the entire key hence 2NF
CityTemp transitively depends on Studio hence violates 3NF

Old Scheme → {Studio , StudioCity, CityTemp}
New Scheme → {Studio , StudioCity}
New Scheme → {StudioCity , CityTemp}

9. **Give the minimal cover Algorithm. Find the minimal cover using the minimal cover algorithm for the following functional dependency.**
   F = {B->A, D->A,AB->D}

The minimal cover (or canonical cover) for a set of functional dependencies F is an equivalent set of dependencies that is minimal, meaning that it has no extraneous attributes, and each functional dependency has a single attribute on the right-hand side. Here is the algorithm to find the minimal cover:

- ✓ **Decompose the FDs**: Ensure that every functional dependency in F has a single attribute on the right-hand side.
- ✓ **Remove extraneous attributes**: For each functional dependency X→A in F, check if any attribute in A is extraneous.
- ✓ **Remove redundant dependencies**: For each functional dependency X→A in F, check if it is redundant by computing the closure of the remaining dependencies and verifying if it still implies X→A.

The Algorithm is given as follows:

- Set F:=E
- Replace each FD
  - X → {A1, A2, A3 … An} in F by n FD's
  - X → A1
  - X → A2
  - …
  - X → An
- For each FD X→A in F
  - For each attribute B that is an element of X
    - If F→{X→A}U (X-{B}→A) == F
      - Replace X→A with X-{B}→A in F
- For each remaining FD X→A in F
  - If {F – {X→A}} == F, then
    - Remove X→ A from F

✓ **Decompose the FDs**

The given FDs are already decomposed as each has a single attribute on the right-hand side:

F={B→A, D→A, AB→D}.

✓ **Remove Extraneous Attributes**

Check each FD for extraneous attributes.

Check AB→D.

- Consider A in AB→D. We need to check if (AB−A)→D (i.e., B→D) holds.
- Comparing AB → D with other FD in F.
  - B → A, use augmentation rule.
  - BB → AB which can be written as B → AB.
- Now, using the transitive property, we consider AB→D and B → AB.
- We get, B → D, therefore we can replace AB→D, with B→D in F.

✓ **Remove Redundant Dependencies**

Check if any FD is redundant by computing the closure of the remaining dependencies and seeing if it still implies the FD.

Check B→D:

- Remove B→A from F. The remaining set is F′={D→A, B→D}.
- Compute the closure of {B} using F′:
  - Start with {B}.
  - Dependencies in F′ can be applied since B→A, can be derived by applying transitive property on D→A and B→D .

Thus, B→A is redundant.

Minimal Cover

The minimal cover for F={B→A, D→A, AB→D} is: {D→A, B→D}

10. Define Functional Dependency. Consider two sets of Functional dependency F={A→C,AC→D,E→AD,E→H} and G={A→CD, E→AH} Are they equivalent? Explain in Detail.

A **functional dependency** in a relational database context specifies a relationship between two sets of attributes. Specifically, for a given relation, an attribute B is functionally dependent on an attribute A (denoted A→B) if, for any two tuples in the relation, if they have the same value for A, then they must also have the same value for B.

Definition of Equivalence

Two sets of functional dependencies F and G are considered **equivalent** if they imply each other. This means that:

- Every functional dependency in F can be derived from G.
- Every functional dependency in G can be derived from F.

Given Functional Dependencies

Set F:

- A→ C
- AC→ D
- E→AD
- E→H

Set GGG:

- A→ CD
- E→ AH

To determine if F and G are equivalent, we need to check if each set can derive the functional dependencies in the other set.

Deriving from F to G

1. **From F to A→ CD in G**:
    - From A→C in F and AC→ D in F, we can combine these:
        - Since A→ C and AC→D, we can derive A→ CD (because A implies C, and A with C implies D).
2. **From F to E→ AH in G**:
    - From E→ AD and E→ H in F, we can combine these:
        - E→ A (since E→ AD and D is irrelevant for A)
        - E→ H
        - Therefore, E→ AH can be derived (since E implies A and H).

Deriving from G to F

1. **From G to A→ C in F**:
    - A→ CD implies A→ C, which is directly available from G.
2. **From G to AC→ D in F**:
    - A→ CD implies that if A has C, D must follow. So, AC→ D can be derived.
3. **From G to E→ AD in F**:
    - E→ AH implies E→ A and E→ H. We need to see if E→ AD can be derived:
        - E→ A and A→ CD (from G) implies E→ CD, but this is not exactly E→ AD without E→ D.
4. **From G to E→ H in F**:
    - E→ AH implies E→ H, so E→ H is derived.

Conclusion

The sets F and G are **not equivalent** because:

- G implies all dependencies in F except for E→ AD, which cannot be directly derived from G.
- F implies all dependencies in G.

So, while F and G are close in terms of coverage, they are not strictly equivalent because G does not cover all dependencies from F.

Module 5

1. Explain the desirable properties of Transactions.

Atomicity:

- **Definition**: Atomicity ensures that a transaction is treated as a single, indivisible unit of work. This means that either all the operations within the transaction are executed, or none of them are.
- **Importance**: This property ensures that the database remains in a consistent state even if a transaction fails. For example, if a transaction involves transferring money from one account to another, atomicity ensures that either both the debit and credit operations are completed or neither is, preventing scenarios where money could be lost or created from nothing.

Consistency Preservation:

- **Definition**: Consistency ensures that a transaction takes the database from one valid state to another, maintaining the integrity constraints defined in the database schema.
- **Importance**: Consistency guarantees that the database adheres to the rules and constraints defined, such as unique keys, foreign keys, and other business rules. For example, in a banking system, the total amount of money in all accounts should remain constant after a transaction if no money is being added or removed from the system.

Isolation:

- **Definition**: Isolation ensures that the operations of one transaction are not visible to other transactions until the transaction is committed. This property prevents transactions from interfering with each other.
- **Importance**: Isolation avoids the temporary update problem, where one transaction sees an intermediate state of another transaction. For instance, if two transactions are updating the same account balance, isolation ensures that one transaction's updates are not visible to the other until the first transaction is completed, thus preventing incorrect computations based on partial data.

Durability or Permanency:

- **Definition**: Durability ensures that once a transaction is committed, its changes are permanent and will not be lost, even in the event of a system failure.
- **Importance**: Durability guarantees that committed transactions are saved permanently. For example, in an e-commerce application, once an order is placed and committed, the order details should be stored permanently, ensuring that the order can be processed and fulfilled even if there is a subsequent system crash.

2. Explain Different states of Transactions with Diagram.

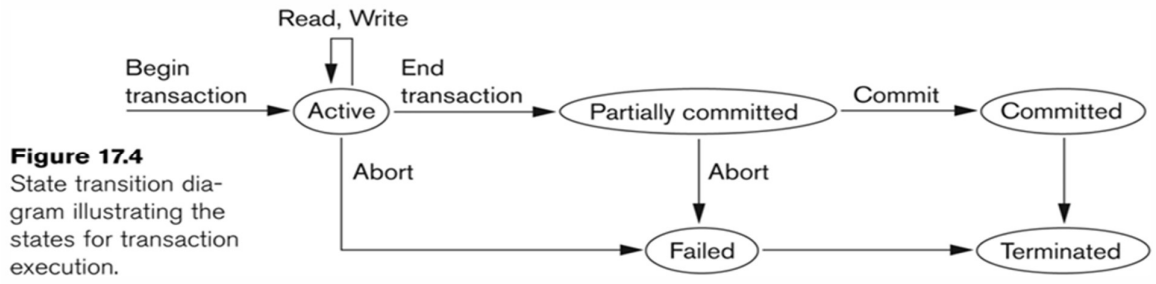
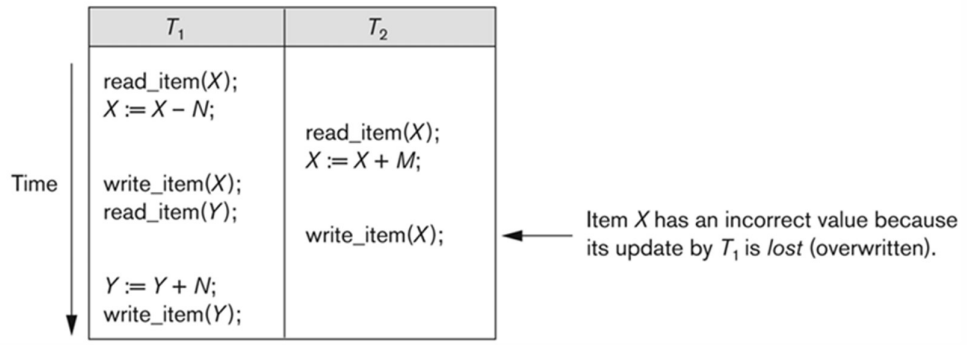


Figure 17.4
State transition diagram illustrating the states for transaction execution.

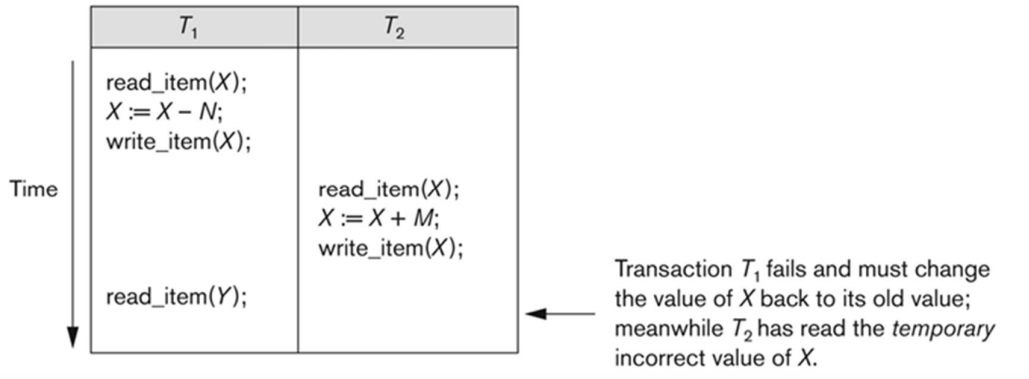
Active state: The transaction is currently executing its operations.
Partially committed state: The transaction has finished executing but has not yet made its changes permanent.
Committed state: The transaction has successfully completed, and all its changes are now permanent.
Failed state: The transaction has encountered an error or was aborted and cannot proceed.
Terminated state: The transaction has finished its execution, either by committing successfully or by failing and rolling back.

3. **Explain Why concurrency is needed?**

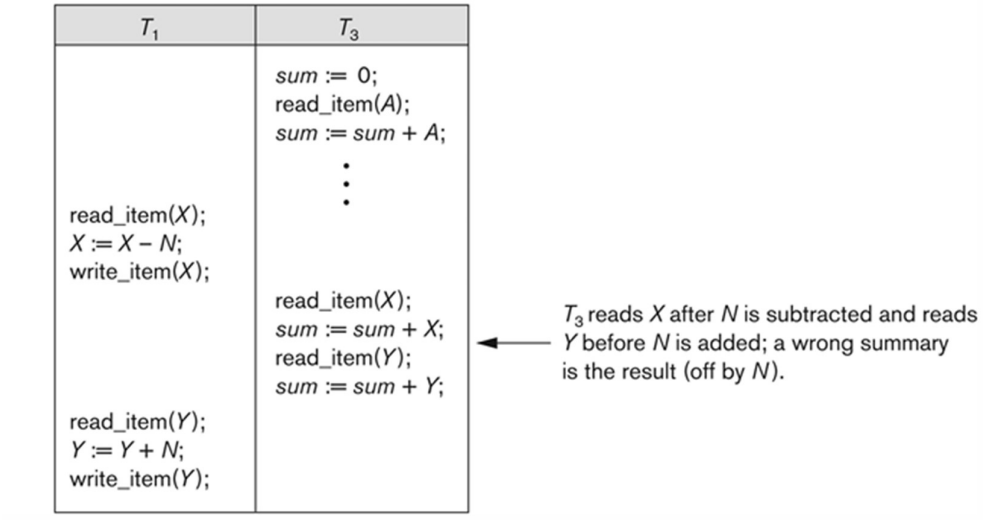
- **The Lost Update Problem**
  - This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.



- **The Temporary Update (or Dirty Read) Problem**
  - This occurs when one transaction updates a database item and then the transaction fails for some reason.
  - The updated item is accessed by another transaction before it is changed back to its original value.



- **The Incorrect Summary Problem**
  - If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

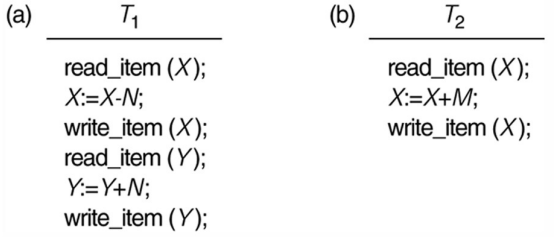


4. Develop the steps involved in read and write operations of transactions.

READ AND WRITE OPERATIONS

Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.

- read_item(X): Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
- read_item(X) command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the buffer to the program variable named X.
- write_item(X): Writes the value of program variable X into the database item named X.
- write_item(X) command includes the following steps:
  - Find the address of the disk block that contains item X.
  - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
  - Copy item X from the program variable named X into its correct location in the buffer.
  - Store the updated block from the buffer back to disk (either immediately or at some later point in time).



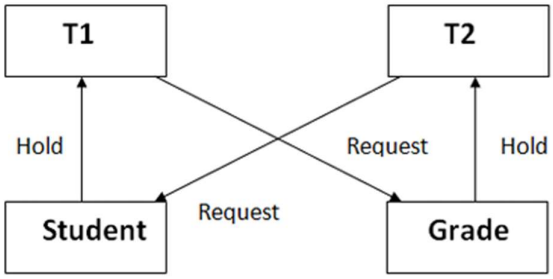
5. Explain the reasons for the failure of transactions.

- **A computer failure (system crash):** A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.
- **A transaction or system error**: Some operations in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.
- **Local errors or exception conditions detected by the transaction:** Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled. A programmed abort in the transaction causes it to fail.
- **Concurrency control enforcement:** The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.
- **Disk failure:** Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
- **Physical problems and catastrophes:** This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

6. Explain Deadlock and Starvation.

Deadlock:

- **Definition**: Deadlock occurs when a set of transactions are waiting for each other in a circular chain, where each transaction holds a resource that the next transaction in the chain is waiting to acquire. This results in all transactions being stuck indefinitely.
- **Scenario**: For example, if transaction T1 holds a lock on resource R1 and is waiting for resource R2, while transaction T2 holds a lock on resource R2 and is waiting for resource R1, both transactions will be stuck in a deadlock.



- **Detection and Prevention**:
  - **Prevention Protocols**: Strategies include locking all required resources in advance or following a strict ordering of resource acquisition to prevent circular wait.
  - **Timestamp-Based Protocols**:
    - **Wait-Die Scheme**: Older transactions wait for resources held by younger ones, but younger transactions are killed and restarted if they request resources held by older ones.
    - **Wound-Wait Scheme**: Older transactions can preempt resources held by younger transactions, forcing the younger transactions to roll back.
  - **No Waiting Algorithm**: Immediately aborts and restarts transactions that cannot obtain a lock.
  - **Deadlock Detection**: Uses mechanisms like wait-for graphs to detect cycles indicating deadlock and then resolves them by aborting one or more transactions.
  - **Victim Selection**: Chooses which transaction to abort based on criteria like the age or priority of transactions.
  - **Timeouts**: Automatically abort transactions that wait longer than a predefined period.

Starvation:

- **Definition**: Starvation occurs when a transaction is repeatedly delayed indefinitely because other transactions are continually given preference. This can happen if a transaction is continually chosen as a victim during deadlock resolution.
- **Scenario**: For instance, if a low-priority transaction is consistently aborted in favor of higher-priority transactions, it may never get a chance to execute.
- **Solution**:
  - **Fairness Mechanisms**: Implementing first-come-first-served queues or ensuring fair scheduling can mitigate starvation by giving each transaction a chance to execute.
  - **Wound-Wait Consideration**: In the Wound-Wait scheme, ensure that younger transactions are not perpetually wounded by long-running older transactions to prevent starvation.

7. Explain the characteristics of NoSQL Databases.

- **Schema-less Data Model**:
  - **Definition**: NoSQL databases do not require a predefined schema, allowing for flexible and dynamic data structures.
  - **Example**: In a document database like MongoDB, each document can have a different structure, allowing for easy updates and changes to the data model without requiring a schema migration.
- **Horizontal Scalability**:
  - **Definition**: NoSQL databases are designed to scale out by adding more servers or nodes, rather than scaling up by adding more power to a single server.
  - **Example**: Distributed databases like Cassandra and HBase can handle large amounts of data and high transaction volumes by distributing the load across multiple servers.
- **Distributed Architecture**:
  - **Definition**: NoSQL databases use a distributed architecture, spreading data across multiple servers to improve performance and reliability.
  - **Example**: Apache Cassandra uses a peer-to-peer distributed system across its nodes, avoiding single points of failure and enabling efficient data distribution and replication.
- **Variety of Data Models**:
  - **Definition**: NoSQL databases support various data models, including key-value, document, column-family, and graph.
  - **Example**:
    - **Key-Value Stores**: Redis, where data is stored as a collection of key-value pairs.
    - **Document Stores**: MongoDB, where data is stored in documents (usually JSON or BSON).
    - **Column-Family Stores**: Cassandra, where data is stored in columns and rows.
    - **Graph Databases**: Neo4j, which stores data in nodes and edges, representing relationships.
- **Eventual Consistency**:
  - **Definition**: Many NoSQL databases embrace eventual consistency, meaning data updates will eventually propagate to all nodes, but may not be immediately consistent.
  - **Example**: Amazon DynamoDB ensures that all copies of data will converge to the same state eventually, allowing for high availability and partition tolerance.
- **Flexible and Efficient Storage**:
  - **Definition**: NoSQL databases are designed to handle large volumes of unstructured or semi-structured data efficiently.
  - **Example**: In Hadoop's HBase, data is stored in a highly compressed format, optimized for read and write operations on large datasets.
- **Support for Big Data and Real-Time Applications**:
  - **Definition**: NoSQL databases are often used in big data and real-time web applications due to their ability to handle large-scale, high-velocity data.
  - **Example**: Apache Cassandra is used by companies like Netflix to manage massive amounts of streaming data in real time.
- **Developer-Friendly**:
  - **Definition**: NoSQL databases provide APIs and query languages that are often more intuitive and easier to use for developers compared to traditional SQL.
  - **Example**: MongoDB uses a JSON-like query language that is simple for developers to understand and use, aligning closely with modern web development practices.

8. Explain Types of databases of NoSQL.

Document-Based Databases

Store data in documents using formats like JSON, BSON, or XML. Each document is a self-contained unit with flexible schema.

- **Features**:
  - Schema-less design allows for dynamic, flexible data structures.
  - Supports nested documents and arrays.
  - Often includes powerful querying capabilities, including indexing and aggregation.

Advantages	Disadvantages
Flexibility in data modeling.	Can be less efficient for complex transactions involving multiple documents.
Easy to scale horizontally.	Potential for data duplication due to denormalization.
Efficient for storing and retrieving complex hierarchical data.	
Good for applications with evolving schemas.	

- **Examples**: MongoDB, CouchDB

Key-Value Stores

Store data as a collection of key-value pairs, where each key is unique and maps directly to a value.

- **Features**:
  - Simple data model with high performance.
  - Efficient for quick lookups and simple data retrieval.
  - Often includes in-memory storage for fast access.

Advantages	Disadvantages
High performance and low latency.	Limited querying capabilities.
Easy to scale horizontally.	Inefficient for complex queries or relationships.
Flexible for various use cases, especially caching and session management.	Typically lacks built-in data integrity constraints.

- **Examples**: Redis, Riak

Column-Oriented Databases

Store data in columns rather than rows, with each column family containing rows with different numbers of columns.

- **Features**:
  - Optimized for read and write performance on large datasets.
  - Supports high compression and efficient storage.
  - Can handle large-scale data across distributed systems.

Advantages	Disadvantages
High performance for read/write operations.	Complex data modeling compared to other NoSQL types.
Efficient storage and retrieval for large datasets.	Potentially high learning curve for setup and maintenance.
Suitable for analytical applications and time-series data.	Can be inefficient for transactional applications.

- **Examples**: Apache Cassandra, HBase

Graph-Based Databases

- **Description**:
  - Store data in nodes and edges, with nodes representing entities and edges representing relationships between entities.
- **Features**:
  - Optimized for managing and querying relationships.
  - Supports complex queries on interconnected data.
  - Often includes ACID transaction support for data integrity.

Advantages	Disadvantages
Ideal for applications with complex relationships.	Less efficient for large-scale bulk data operations.
High performance for graph traversal operations.	May require specialized knowledge for effective use.
Flexible schema for evolving data structures.	Potentially more complex to scale horizontally.

- **Examples**: Neo4j, Amazon Neptune

9. Define the Graph database. List the advantages and Disadvantages of Graph databases.

Graph-Based Data Model:

A data model focusing on building relationships between data elements. Each element is stored as a node, and the associations between these nodes are represented by edges. Properties provide additional information about nodes and edges.

- **Nodes**: Represent instances of data or objects that need to be tracked.
- **Edges**: Represent the relationships or connections between nodes.
- **Properties**: Store information associated with nodes and edges.

Advantages of Graph Data Model

- **Structure**:
  - **Agility**: Graph structures are highly flexible and adaptable to changing data requirements.
- **Explicit Representation**:
  - **Relationship Clarity**: Relationships between entities are explicitly represented, making complex queries straightforward.
- **Real-Time Output**:
  - **Efficient Queries**: Queries return real-time results, providing quick insights into interconnected data.

Disadvantages of Graph Data Model

- **No Standard Query Language**:
  - **Varied Syntax**: Different graph databases use different query languages (e.g., Cypher for Neo4j, GraphQL for DGraph), leading to a lack of standardization.
- **Limited Suitability for Transactions**:
  - **Transactional Challenges**: Graph databases may not be well-suited for systems requiring high transaction throughput, compared to traditional relational databases.
- **Small User Base**:
  - **Support Issues**: The smaller user base may lead to challenges in finding support and resources for troubleshooting and development.

10. Compare NoSQL and RDBMS.

Aspect	NoSQL	RDBMS
Data Model	Schema-less or flexible schema (key-value, document, column, graph)	Structured schema (tables, rows, columns)
Schema Flexibility	Dynamic schema, accommodates unstructured data	Fixed schema, requires predefined structure
Scalability	Horizontal scaling (distributed across multiple servers)	Primarily vertical scaling (upgrading hardware), with some horizontal scaling options
Transaction Support	Eventual consistency; some support for ACID transactions	Strong ACID compliance (Atomicity, Consistency, Isolation, Durability)
Query Language	Custom or varied query languages (e.g., MQL, Cypher, GraphQL)	SQL (Structured Query Language)
Data Integrity	Limited support for constraints; focus on flexibility	Strong support for constraints; data normalization
Use Cases	Big data, real-time analytics, dynamic or evolving data	Structured data, complex queries, transactional systems
Replication	Distributed replication across multiple nodes or data centers	Master-slave or primary-replica replication for redundancy and scaling reads

11. Explain the need of Schema less database.

- **Flexibility in Data Storage**
  - **Dynamic Data**: Schemaless databases allow for the storage of diverse and evolving data types without a predefined schema. This flexibility is crucial for applications where the data model is not fully known or changes over time.
  - **Example**: In a document database like MongoDB, you can store documents with different structures in the same collection, adapting to changes in data requirements without schema migrations.
- **Handling Non-Uniform Data**
  - **Varied Records**: Schemaless databases handle records with different sets of fields effectively. This is useful for datasets where each record may have a different structure.
  - **Example**: In a key-value store like Redis, each key can hold data of any type, accommodating varying data formats without needing a uniform schema.
- **Simplified Data Management**
  - **Avoids Sparse Tables**: Traditional relational schemas can lead to sparse tables with many null columns when records have different attributes. Schemaless databases avoid this by allowing each record to contain only the data it needs.
  - **Example**: In a column-family database like Cassandra, each column family can store rows with different columns, avoiding the issue of unused or null columns.
- **Adaptability to Change**
  - **Evolving Requirements**: Schemaless databases are well-suited for projects where the data model evolves frequently. You can add or remove fields without worrying about schema modifications or data migrations.
  - **Example**: In a graph database like Neo4j, you can dynamically add new types of relationships or properties without changing the overall schema.
- **Implicit Schema in Application Code**
  - **Code Integration**: Although schemaless databases do not enforce a schema, the application code that interacts with the database often relies on an implicit schema. This means that data assumptions and structures are managed at the application level.
  - **Example**: An application might expect a field named email to always contain a string value. Even though the database is schemaless, the application code enforces this expectation.
- **Integration with Web Services**
  - **Encapsulation**: Encapsulating database interactions within a single application and using web services for integration can mitigate the challenges of a schemaless database. This approach centralizes schema management within the application layer.
  - **Example**: An application that provides an API that can handle schema changes internally and present a consistent interface to other applications.
- **Controlled Schema Changes**
  - **Schema Evolution**: While relational databases require schema changes to be explicitly defined, schemaless databases naturally accommodate evolving data requirements. However, it is still essential to manage schema changes carefully to maintain data consistency.
  - **Example**: For a document database, you can introduce new fields in documents as needed without schema constraints, but you must ensure that applications handling the data are updated accordingly.