Database & Database Management System (DBMS)

What is a Database?

A database is an organized collection of structured data that represents some aspects of the real world. Databases are created to manage and store large volumes of information systematically, making retrieval, manipulation, and management efficient and accurate. The data within a database is typically structured in a way that enables users to easily access and analyse relevant information.

Database Management System (DBMS)

A Database Management System (DBMS) is specialized software designed to store, retrieve, manage, and manipulate databases. It provides an interface between the database and the users or application programs, ensuring that data is consistently organized and remains accessible. A DBMS handles requests from users or applications, processes them, and provides the necessary data while maintaining security and integrity.

Why Use DBMS?

Traditional file-processing systems have several limitations, which a DBMS effectively addresses:

1. Data Redundancy and Inconsistency
2. Difficulty in Accessing Data
3. Data Isolation
4. Integrity Problems
5. Atomicity of Updates
6. Concurrent Access by Multiple Users
7. Security Problems

Components of DBMS

- **Database Engine:** Handles storage, retrieval, and updates of data.
- **Database Schema:** Defines the logical structure of the database.
- **Query Processor:** Parses and executes database queries.
- **Transaction Manager:** Ensures data integrity through ACID properties (Atomicity, Consistency, Isolation, Durability).
- **Storage Manager:** Manages the storage of data on disk and memory.
- **Security Manager:** Controls user access and ensures secure data management.

Advantages of Using DBMS

- **Reduced Data Redundancy:** DBMS stores data efficiently, minimizing duplication.

- **Improved Data Integrity:** Integrity constraints ensure accuracy and reliability of data.
- **Enhanced Security:** Advanced access controls and authentication mechanisms protect data.
- **Efficient Data Access:** Users can retrieve and manipulate data swiftly through structured queries.
- **Ease of Data Maintenance:** Simplifies tasks such as backup, recovery, and modification of data structures.
- **Concurrency Control:** Supports multiple simultaneous users while preserving data integrity.

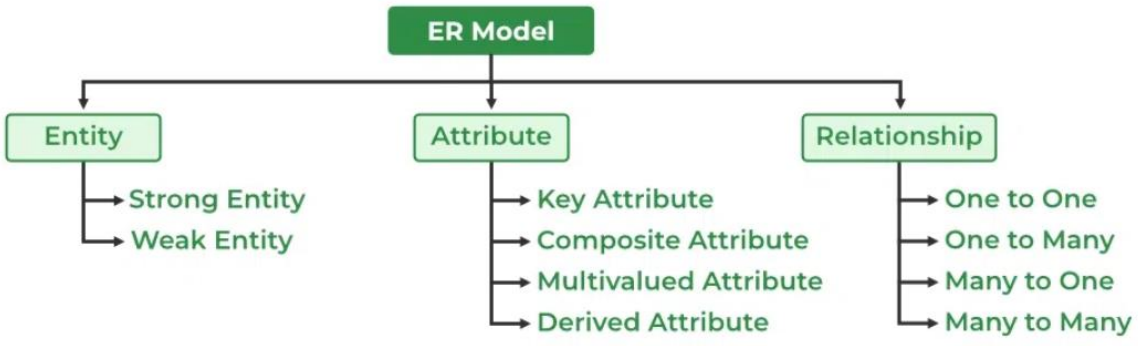
Examples of DBMS

- MySQL
- Oracle Database
- PostgreSQL
- Microsoft SQL Server
- MongoDB (NoSQL)
- IBM DB2

Introduction of ER Model

The Entity-Relationship Model (ER Model) is a conceptual model for designing a databases. This model represents the logical structure of a database, including entities, their attributes and relationships between them.

- **Entity:** An objects that is stored as data such as *Student*, *Course* or *Company*.
- **Attribute:** Properties that describes an entity such as *StudentID*, *CourseName*, or *EmployeeEmail*.
- **Relationship:** A connection between entities such as "a *Student* enrolls in a *Course*".



ER Model in Database Design Process

We typically follow the below steps for designing a database for an application.

- Gather the requirements (functional and data) by asking questions to the database users.
- Create a logical or conceptual design of the database. This is where ER model plays a role. It is the most used graphical representation of the conceptual design of a database.

- After this, focus on Physical Database Design (like indexing) and external design (like views)

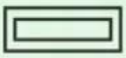
Why Use ER Diagrams In DBMS?

- ER diagrams represent the E-R model in a database, making them easy to convert into relations (tables).
- These diagrams serve the purpose of real-world modeling of objects which makes them intently useful.
- Unlike technical schemas, ER diagrams require no technical knowledge of the underlying DBMS used.
- They visually model data and its relationships, making complex systems easier to understand.

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, such as a student's multiple phone numbers
- **Double Rectangle:** Represents weak entities, which depend on other entities for identification.

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

What is an Entity?

An Entity represents a real-world object, concept or thing about which data is stored in a database. It act as a building block of a database. Tables in relational database represent these entities.

Example of entities:

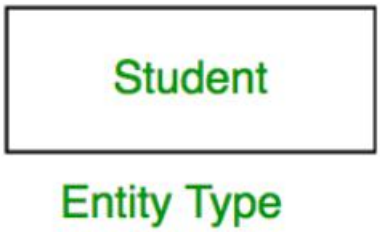
- **Real-World Objects:** Person, Car, Employee etc.
- **Concepts:** Course, Event, Reservation etc.
- **Things:** Product, Document, Device etc.

The entity type defines the structure of an entity, while individual instances of that type represent specific entities.

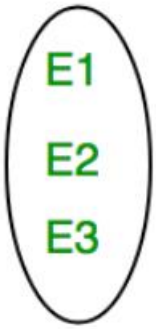
What is an Entity Set?

An entity refers to an individual object of an entity type, and the collection of all entities of a particular type is called an entity set. For example, E1 is an entity that belongs to the entity type "Student," and the group of all students forms the entity set.

In the ER diagram below, the entity type is represented as:



Entity Type



Entity Set

1. Strong Entity

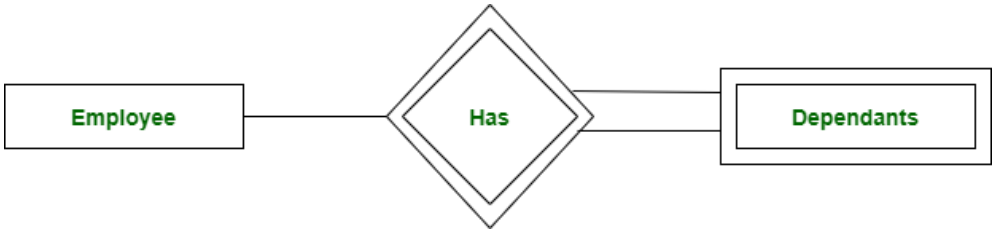
A Strong Entity is a type of entity that has a key Attribute that can uniquely identify each instance of the entity. A Strong Entity does not depend on any other Entity in the Schema for its identification. It has a primary key that ensures its uniqueness and is represented by a rectangle in an ER diagram.

2. Weak Entity

A Weak Entity cannot be uniquely identified by its own attributes alone. It depends on a strong entity to be identified. A weak entity is associated with an identifying entity (strong entity), which helps in its identification. A weak entity are 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.

Example:

A company may store the information of dependents (Parents, Children, Spouse) of an Employee. But the dependents can't exist 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.



Attributes in ER Model

Attributes are the properties that define the entity type. For example, for a Student entity 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.

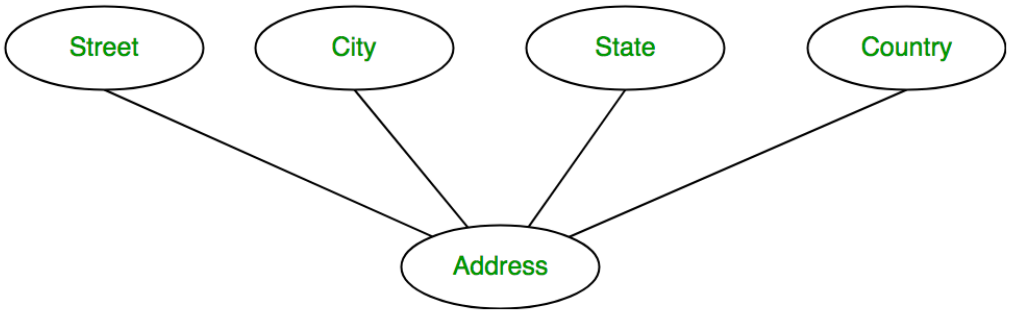
Types of Attributes

1. 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 an underline.

2. Composite 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.



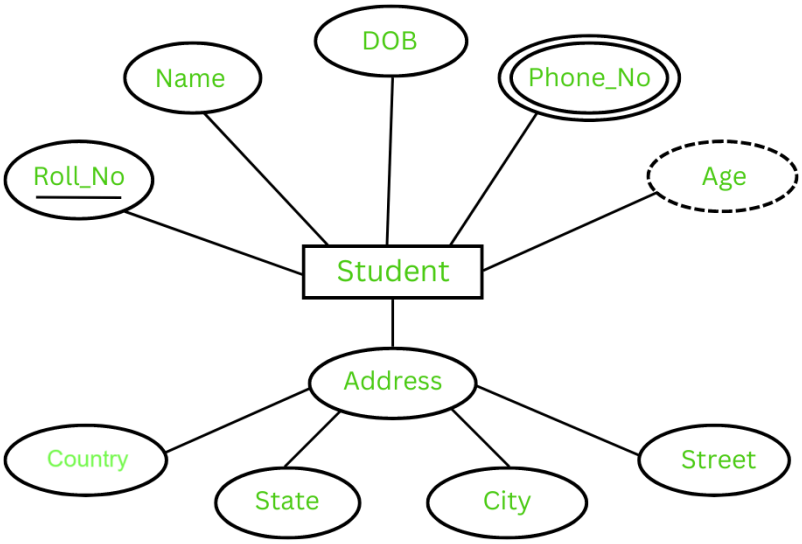
3. Multivalued Attribute

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.

4. Derived Attribute

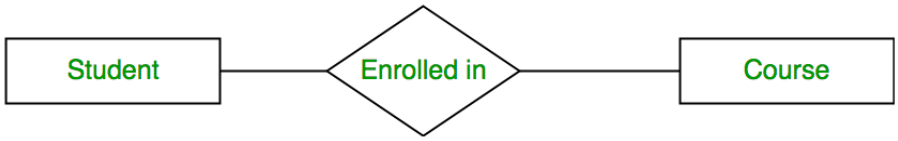
An attribute that can be derived from other attributes of the entity type is known as a derived attribute. e.g.; Age (can be derived from DOB). In ER diagram, the derived attribute is represented by a dashed oval.

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

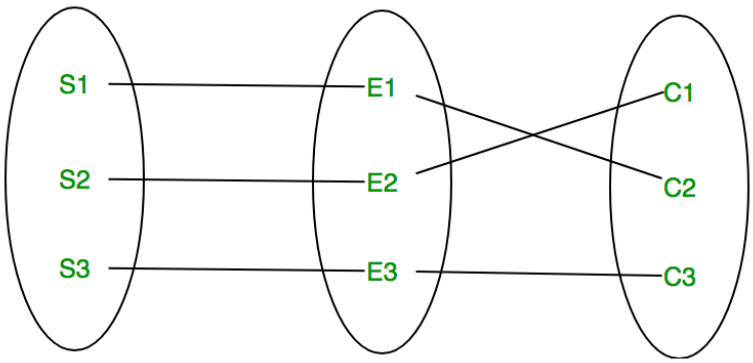


Relationship Type and Relationship Set

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.



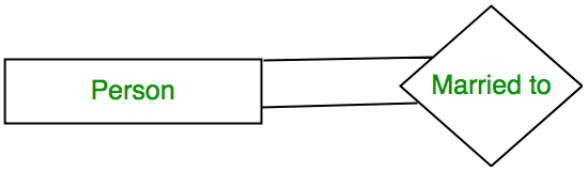
A set of relationships of the same type is known as a **relationship set**. The following relationship set depicts S1 as enrolled in C2, S2 as enrolled in C1, and S3 as registered in C3.



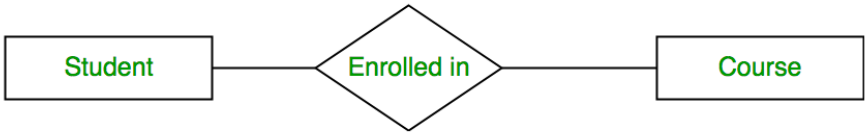
Degree of a Relationship Set

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.



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.



3. Ternary Relationship: When there are three entity sets participating in a relationship, the relationship is called a ternary relationship.

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

Cardinality in ER Model

The maximum number of times an entity of an entity set participates in a relationship set is known as cardinality.

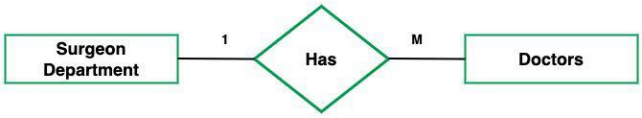
Cardinality can be of different types:

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.

2. One-to-Many

In one-to-many mapping as well where each entity can be related to more than one entity. Let us assume that one surgeon department can accommodate many doctors. So the Cardinality will be 1 to M. It means one department has many Doctors.



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.

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.

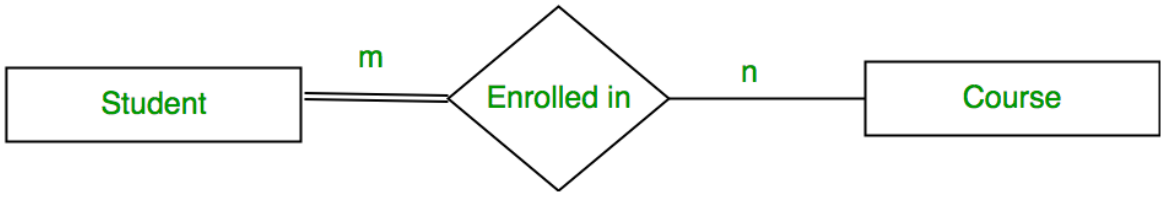
Participation Constraint

Participation Constraint is applied to the entity participating in the relationship set.

1. Total Participation: Each entity in the entity set must participate in the relationship. If each student must enroll in a course, the participation of students will be total. Total participation is shown by a double line in the ER diagram.

2. Partial Participation: The entity in the entity set may or may NOT participate in the relationship.

The diagram depicts the 'Enrolled in' relationship set with Student Entity set having total participation and Course Entity set having partial participation.



Keys in DBMS

Keys are attributes or sets of attributes that help uniquely identify tuples (rows) in a relation. They are fundamental for ensuring entity integrity and supporting efficient querying.

- **Super Key:** A set of attributes that uniquely identifies a tuple. May contain extra attributes.
- **Candidate Key:** A minimal super key; removing any attribute would cause it to fail uniqueness.
- **Primary Key:** A candidate key chosen to uniquely identify tuples in a relation. Must be unique and not null.
- **Alternate Key:** Other candidate keys not selected as the primary key.
- **Composite Key:** A key made of more than one attribute (e.g., {CourseID, StudentID})
- **Foreign Key:** An attribute in one relation that references the primary key of another relation.
- **Unique Key:** Similar to primary key, but can accept one NULL value and is not necessarily the primary identifier.

Relational Constraints in DBMS

Constraints are rules enforced on data in a database to ensure its validity, accuracy, and integrity.

- **Domain Constraint:** Specifies the valid set of values for an attribute (e.g., Age must be integer).
- **Tuple Uniqueness Constraint:** No two rows can be identical in a relation.
- **Key Constraint:** Primary key must be unique and not null.
- **Entity Integrity Constraint:** No primary key attribute can be null.
- **Referential Integrity Constraint:** Foreign keys must match a primary key in another relation or be null.

Functional Dependency

Functional dependency expresses the relationship between attributes in a relation. It states that if two tuples have the same value for attribute X, they must have the same value for attribute Y.

Notation: $X \rightarrow Y$ (Y is functionally dependent on X)

Types of Functional Dependency

- **Trivial Dependency:** $X \rightarrow Y$ is trivial if $Y \subseteq X$

- **Non-Trivial Dependency:** $Y \not\subset X$

Closure of an Attribute Set

The closure of an attribute set X (denoted as X^+) is the set of all attributes that are functionally dependent on X under a given set of functional dependencies.

Steps to Find Closure

1. Start with $X^+ = X$
2. Iteratively apply functional dependencies to add new attributes to X^+
3. Stop when no more attributes can be added

Example: If $F = \{ A \rightarrow B, B \rightarrow C \}$, then $A^+ = \{A, B, C\}$

Concept	Description	Example
Simple Attribute	Indivisible, atomic values	Age, Salary
Composite Attribute	Can be split into sub-attributes	Name, Address
Multi-valued Attribute	Multiple values per entity	PhoneNumbers
Derived Attribute	Computed from other attributes	Age (from DOB)
Key Attribute	Uniquely identifies an entity	EmployeeID
Functional Dependency	Y depends on X if $X \rightarrow Y$	$A \rightarrow B$
Closure	All attributes derivable from a given set	$A^+ = \{A, B, C\}$

Decomposition & Normalization in DBMS

What is Decomposition?

Decomposition is the process of breaking a complex relation (table) into two or more simpler relations to eliminate redundancy, maintain data integrity, and improve design. It helps ensure that a database structure is easy to maintain and update.

Properties of Decomposition

- **Lossless Join Decomposition:** Ensures that no data is lost when relations are joined back together. It guarantees that original relation can be perfectly reconstructed.
- **Dependency Preservation:** Ensures that all functional dependencies in the original relation are still enforceable after decomposition without needing to perform joins.

Types of Decomposition

- **Lossless Join:** If R is decomposed into R1 and R2, and $R1 \bowtie R2 = R$, the decomposition is lossless.
- **Lossy Join:** If $R1 \bowtie R2 \supset R$ or does not equal R, information is lost in the join and the decomposition is lossy.

What is Normalization?

Normalization is a systematic approach of organizing data in a database to reduce redundancy and improve data integrity. It involves decomposing large tables into smaller ones and defining relationships among them.

The goals of normalization include:

- Eliminating data redundancy (repetition)
- Ensuring data dependencies make sense
- Improving consistency and reducing anomalies (insertion, update, deletion)

Normal Forms

Normal forms are rules that relations must satisfy to be considered normalized. The commonly used normal forms are:

First Normal Form (1NF)

- Each cell in a table must contain only atomic (indivisible) values.
- No repeating groups or arrays.
- **Example Violation:** A table with multiple phone numbers in one cell.
- **Correction:** Store each phone number in a separate row.

Second Normal Form (2NF)

- The relation must already be in 1NF.
- No partial dependency should exist; i.e., no non-prime attribute should depend only on a part of a candidate key.
- **Example Violation:** In a table with {StudentID, CourseID} as the primary key, if StudentName depends only on StudentID.
- **Correction:** Split the table to separate student and course details.

Third Normal Form (3NF)

- The relation must be in 2NF.
- There must be no transitive dependency for non-prime attributes.
- **Example Violation:** If $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$ is transitive. This should be avoided if C is a non-key attribute.
- **Correction:** Decompose into separate tables for $A \rightarrow B$ and $B \rightarrow C$.

Boyce-Codd Normal Form (BCNF)

- It is a stricter version of 3NF.
- Every non-trivial functional dependency $X \rightarrow Y$ must have X as a super key.
- **Example Violation:** If a functional dependency exists where determinant is not a super key.
- **Correction:** Further decompose tables to ensure all dependencies meet the BCNF condition.

Part V – Transactions & Concurrency

What is a Transaction?

A transaction is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions consist of multiple operations like reading and writing data, and they are used to ensure data integrity in multi-user environments.

Basic Operations in a Transaction

- **Read(A):** Reads the value of item A into memory (buffer).
- **Write(A):** Writes the updated value from memory back to the database.

Transaction States

1. **Active:** Initial state where operations are being executed.
2. **Partially Committed:** Final operation has been executed, but data not yet saved permanently.
3. **Committed:** All changes have been permanently saved in the database.
4. **Failed:** Transaction cannot proceed due to an error.
5. **Aborted:** All operations are rolled back, restoring original values.
6. **Terminated:** Transaction has either committed or aborted and its life cycle ends.

ACID Properties

Normal Form	Condition	Purpose
1NF	Atomic values only; no multi-valued attributes	Eliminates repeating groups
2NF	1NF + No partial dependency	Removes dependencies on part of a key
3NF	2NF + No transitive dependency	Removes indirect dependencies
BCNF	Every determinant is a super key	Stronger form of 3NF

ACID properties ensure reliability of transactions in a DBMS:

- **Atomicity:** All operations of a transaction must complete successfully or none at all.
- **Consistency:** A transaction must transform the database from one valid state to another.
- **Isolation:** Transactions must execute independently, without interference.
- **Durability:** Once a transaction is committed, its changes are permanent.

Schedules

A schedule is a sequence in which operations from multiple transactions are executed. Schedules determine the outcome of concurrent transaction execution.

Types of Schedules

- **Serial Schedule:** Transactions are executed one after another, no interleaving.
- **Non-Serial Schedule:** Operations of transactions are interleaved.

Recoverability of Schedules

- **Recoverable Schedule:** If a transaction T_j reads a value from T_i, it commits only after T_i commits.
- **Irrecoverable Schedule:** T_j commits before T_i, making recovery impossible if T_i fails.

Serializability

Serializability is the highest level of isolation, ensuring a non-serial schedule yields the same result as some serial execution.

Types of Serializability

- **Conflict Serializability:** A schedule is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.
- **View Serializability:** Two schedules are view serializable if they result in the same reads and writes.

Types of Recoverable Schedules

- **Cascading Schedule:** Failure of one transaction causes multiple dependent transactions to roll back.
- **Cascadeless Schedule:** A transaction can only read committed data, preventing cascading aborts.
- **Strict Schedule:** A transaction cannot read or write a data item until the last transaction that modified it has committed or aborted.

Concept	Description
Transaction	Logical unit of database operations
Atomicity	All-or-nothing execution
Consistency	Maintains valid state transitions
Isolation	Prevents interleaved operations from interfering
Durability	Changes persist after commit
Serializability	Equivalence to serial execution
Recoverable Schedule	Commits only after related transactions commit
Strict Schedule	Waits for previous transaction to finish commit/abort

Relational Algebra

Relational Algebra is a procedural query language that operates on relations (tables) and returns relations as results. It forms the theoretical foundation for SQL and is essential for understanding how queries are evaluated internally by a DBMS.

Basic Operators

1. Selection (σ)

Selects rows that satisfy a given predicate.

Syntax: $\sigma_{condition}(Relation)$

Example: $\sigma_{age > 30}(Employees)$

2. Projection (π)

Retrieves specific columns (attributes) from a relation.

Syntax: $\pi_{attributes}(Relation)$

Example: $\pi_{name, salary}(Employees)$

3. Cross Product (×)

Combines tuples from two relations in all possible combinations (Cartesian product).

Example: Employees × Departments

4. Union (∪)

Combines tuples from two relations, eliminating duplicates.

Condition: Both relations must be union-compatible (same attributes).

Example: $\pi_{name}(Managers) \cup \pi_{name}(Engineers)$

5. Set Difference (−)

Finds tuples in one relation but not in the other.

Example: $\pi_{name}(Employees) - \pi_{name}(Managers)$

6. Rename (ρ)

Changes the name of the relation or attributes.

Syntax: $\rho_{newName}(Relation)$

Extended Operators

1. Intersection (∩)

Returns tuples present in both relations.

Example: $\pi_{name}(Employees) \cap \pi_{name}(Managers)$

2. Conditional Join ($\bowtie_{condition}$)

Combines tuples from two relations based on a specified condition.

Example: $Employees \bowtie_{Employees.dept\_id = Departments.dept\_id} Departments$

3. Equi Join

A special case of conditional join using only equality comparisons.

4. Natural Join (⋈)

Automatically joins relations on all attributes with the same name and removes duplicate columns.

Note: If no common attributes, it behaves like a cross product.

5. Outer Joins

- **Left Outer Join (⟕):** Includes all tuples from the left relation and matched tuples from the right; fills unmatched with NULLs.
- **Right Outer Join (⟖):** Includes all tuples from the right relation and matched tuples from the left.
- **Full Outer Join (⟗):** Includes all tuples from both relations; unmatched parts are filled with NULLs.

6. Division (÷)

Returns tuples in A that are related to all tuples in B.

Condition: Attributes of B must be a subset of attributes of A.

Result: Attributes in A − Attributes in B.

Example Use Case of Division

Relation A: (Student, Course)

Relation B: (Course)

Query: Find students who have taken all courses listed in B.

Operation: A ÷ B

Operator	Symbol	Description
Selection	σ	Filters rows based on condition
Projection	π	Selects specific columns
Cross Product	×	Combines all tuples of two relations
Union	∪	Combines tuples and removes duplicates
Set Difference	−	Returns tuples in one relation but not the other
Rename	ρ	Renames relation or attributes
Intersection	∩	Tuples common to both relations

Conditional Join	$\bowtie_{cond}$	Join based on condition
Equi Join	$\bowtie$	Join using only equality
Natural Join	$\bowtie$	Join on common attributes
Left Outer Join	$⟕$	All from left + matches from right
Right Outer Join	$⟖$	All from right + matches from left
Full Outer Join	$⟗$	All from both relations
Division	$\div$	Find tuples related to all in another set

Storage, B-Trees & Indexing

File Structures in DBMS

In database systems, data is stored in files on secondary storage. Efficient access to this data is crucial, and file structures and indexing methods are employed to facilitate this.

1. Primary Index

- Created on a sorted data file using a primary key (unique and non-null).
- Consists of an ordered list of key values and pointers to their corresponding data blocks.
- **Access Time:** On average, $log_2Bi + 1$ block accesses, where Bi is number of index blocks.
- **Use Case:** Searching for a customer by their unique ID in a sorted list.

2. Clustering Index

- Created on a non-key attribute where records are physically ordered based on the clustering field.
- Multiple records may have the same value for the clustering attribute.
- **Use Case:** Employees grouped and stored together based on department number.

3. Secondary Index

- Provides an alternate access path to data that is not sorted on the indexing attribute.
- Can be created on any attribute (not necessarily unique or ordered).
- **Use Case:** Searching employees by name when the file is ordered by employee ID.

B-Trees

B-Trees are balanced tree data structures used to maintain sorted data and enable efficient insertion, deletion, and search operations.

Properties of B-Trees

- Each node may have a maximum of **P** children, where **P** is the order of the tree.
- Each internal node contains between ⌈P/2⌉ and P children (except root).
- Each internal node can have ⌈P/2⌉-1 to P-1 keys.
- The tree is height-balanced, minimizing disk I/O for search operations.

Advantages of B-Trees

- Efficient for large datasets due to balanced height.
- Supports range queries efficiently.
- Used extensively in database indexing and file systems.

B+ Trees

B+ Trees are a variation of B-Trees where data is stored only in the leaf nodes and internal nodes only maintain keys for routing.

Differences from B-Trees

- Non-leaf nodes do not store actual data pointers—only keys and child pointers.
- Leaf nodes contain data pointers and are linked for sequential access.
- Provides better range query performance due to linked leaf nodes.

Benefits of B+ Trees

- Less tree height compared to B-Trees for same number of entries.
- All data entries appear in leaves, making range queries simpler.

Concept	Description	Use Case
Primary Index	Index on sorted file using primary key	Search by unique customer ID
Clustering Index	dex on non-key, physically ordered field	roup employees by department
Secondary Index	Additional index on unsorted attribute	Search by employee name
B-Tree	Balanced tree with data in internal/leaf nodes	Used in hierarchical storage/indexes
B+ Tree	Only leaf nodes store data; better for range	Index large tables with range access

- Widely used in relational databases for indexing large tables.

Indexing in DBMS

Indexing is a technique used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It acts like a roadmap to efficiently locate data without scanning every row in a table.

What is an Index?

An index is a data structure, typically a B-Tree or Hash Table, that maintains a reference to records in a database table based on the values of one or more columns. Indexes are created on columns that are frequently used in WHERE clauses, joins, or for sorting operations.

Benefits of Indexing

- Faster search and retrieval of records
- Efficient execution of SELECT queries with conditions
- Improved sorting and grouping performance
- Speeds up join operations

Types of Indexes

- **Single-Column Index:** Created on only one column of a table
- **Composite Index:** Created on two or more columns
- **Unique Index:** Ensures all values in the indexed column(s) are unique
- **Clustered Index:** Alters the way records are stored as per the index
- **Non-Clustered Index:** Does not alter the actual table structure but stores a pointer to the data

When to Use Indexing?

- On columns frequently used in WHERE clauses
- On columns used in join conditions
- On columns used in ORDER BY or GROUP BY
- When dealing with large datasets that require high performance

When Not to Use Indexing?

- On tables with very few rows
- On columns that are frequently updated or inserted into (can slow down write performance)
- On columns with high number of duplicate values

Example

CREATE INDEX idx_customer_name ON Customers (Name);

This creates an index on the 'Name' column of the Customers table, which helps accelerate queries like:

SELECT * FROM Customers WHERE Name = 'Alice';

Advanced DBMS Concepts for Interview

1. Advanced Transactions & Concurrency Control

Two-Phase Locking (2PL) – IMP FOR INTERVIEWS

This is the most widely used protocol to ensure serializability. It involves two phases:

- **Growing Phase:** The transaction acquires all the locks it needs without releasing any.
- **Shrinking Phase:** After releasing the first lock, it cannot acquire any more locks.

This ensures that the schedule is conflict-serializable but may lead to deadlocks.

Timestamp Ordering Protocol

Each transaction is assigned a unique timestamp. The protocol ensures that conflicting operations are executed according to their timestamps.

- Read and write operations are checked against the most recent write timestamps.
- If a rule is violated, the transaction is aborted and restarted with a new timestamp.

Deadlock Handling

- **Prevention:** Use strategies like Wait-Die and Wound-Wait to preemptively avoid deadlocks.
- **Detection:** Construct a wait-for graph and check for cycles.
- **Avoidance:** Use resource allocation graphs and bankers algorithm in theory.

Isolation Levels

SQL defines isolation levels to balance performance and correctness:

- **Read Uncommitted:** Allows dirty reads.
- **Read Committed:** Prevents dirty reads, but allows non-repeatable reads.
- **Repeatable Read:** Prevents dirty and non-repeatable reads, but phantom reads may occur.
- **Serializable:** Highest isolation; prevents all concurrency anomalies.

2. Query Optimization

The query optimizer determines the most efficient way to execute a given query by evaluating possible query plans.

Types of Optimization

- **Cost-Based Optimization:** Chooses the query plan with the lowest estimated cost based on data statistics.
- **Rule-Based Optimization:** Uses heuristics and predefined rules to generate execution plans.
- **Join Order Optimization**

Rearranges join operations to minimize intermediate results. For example, joining smaller tables first can reduce overall cost.

Index Selection

Choosing the right indexes can drastically reduce the number of disk accesses and improve query performance.

Subqueries vs Joins

- Subqueries are preferred for readability but may be less optimized by the DBMS.
- Joins are typically more efficient for large datasets.

3. Views and Triggers

Views

A view is a virtual table created by a SELECT query. It doesn't store data itself but presents data from underlying tables.

- Can be used for abstraction and security.
- Updatable views can only be updated if they are based on a single table without GROUP BY, DISTINCT, etc.

Triggers

A trigger is a piece of code that automatically executes in response to certain events on a table like INSERT, UPDATE, DELETE.

Example:

CREATE TRIGGER log_change AFTER UPDATE ON Employees FOR EACH ROW INSERT INTO Logs VALUES (OLD.id, NOW());

4. Stored Procedures and Functions

Stored procedures are precompiled SQL code that can be executed repeatedly. They improve performance and encapsulate logic.

Functions return a value and can be used inside SQL expressions.

CREATE PROCEDURE GetSalary(IN emp_id INT) BEGIN SELECT salary FROM Employees WHERE ID = emp_id; END;

5. Complex SQL Patterns

Window Functions

These functions allow calculations across rows related to the current row:

- RANK(), DENSE_RANK(), ROW_NUMBER(), LEAD(), LAG()

SELECT name, RANK() OVER (PARTITION BY dept ORDER BY salary DESC) AS rank FROM employees;

Common Table Expressions (CTEs)

CTEs allow you to define temporary result sets:

WITH SalesPerDept AS (SELECT dept, SUM(sales) FROM Orders GROUP BY dept) SELECT * FROM SalesPerDept;

6. SQL vs NoSQL – IMP FOR INTERVIEWS

SQL (Relational):

- Structured schema and data
- Supports complex queries and joins
- ACID compliant

NoSQL (Non-Relational):

- Schema-less or dynamic schema
- BASE compliant (Basically Available, Soft state, Eventually consistent)
- Designed for scalability and high throughput
- Types: Document (MongoDB), Key-Value (Redis), Columnar (Cassandra), Graph (Neo4j)

SQL

Structured Query Language (SQL) is the standard language used to interact with relational databases.

- Mainly used to manage data. Whether you want to create, delete, update or read data, SQL provides the structure and commands to perform these operations.
- Widely supported across various database systems like MySQL, Oracle, PostgreSQL, SQL Server and many others.
- Mainly works with Relational Databases (data is stored in the form of tables)

1. SQL Sub-Languages

- **DDL**: CREATE, ALTER, DROP, TRUNCATE, RENAME
- **DML**: SELECT, INSERT, UPDATE, DELETE, MERGE
- **DCL**: GRANT, REVOKE
- **TCL**: COMMIT, ROLLBACK, SAVEPOINT

2. DDL Operations

- **CREATE**:

```
CREATE TABLE Employees (EmpID INT PRIMARY KEY, Salary DECIMAL(10,2));
```

- **ALTER**:

```
ALTER TABLE Users ADD DateOfBirth DATE;
```

- **DROP**:

```
DROP TABLE Orders;
```

- **TRUNCATE**: (faster, not rollback-able)

```
TRUNCATE TABLE Students;
```

- **RENAME**:

```
RENAME TABLE Customers TO Clients;
```

3. Basic SQL Commands

- **SELECT**:

```
SELECT Name, Email FROM Users;
```

- **WHERE**:

```
SELECT * FROM Customers WHERE Country = 'India';
```

- **ORDER BY**:

```
SELECT * FROM Employees ORDER BY Salary DESC;
```

- **INSERT**:

```
INSERT INTO Students (ID, Name) VALUES (101, 'Rahul');
```

- **UPDATE**:

```
UPDATE Users SET Email='abc@example.com' WHERE ID=10;
```

- **DELETE**:

```
DELETE FROM Students WHERE Result='Fail';
```

4. Aggregate Functions

- **MIN / MAX / COUNT / SUM / AVG**

```
SELECT AVG(Age) FROM Students;
```

5. Filtering Data

- **LIKE**: Name LIKE 'A%'
- **IN**: Country IN ('UK','USA','Canada')
- **BETWEEN**: Price BETWEEN 100 AND 500
- **IS NULL**: Email IS NULL

6. Joins

- **INNER JOIN** → Only matching rows
- **LEFT JOIN** → All left + matching right
- **RIGHT JOIN** → All right + matching left
- **FULL OUTER JOIN** → All from both

7. Set Operations

- **UNION** → Removes duplicates
- **UNION ALL** → Keeps duplicates

8. GROUP BY & HAVING

```sql
SELECT CustomerID, COUNT(*)
FROM Orders
GROUP BY CustomerID
HAVING COUNT(*) > 3;
```

9. Subqueries

- **IN with Subquery:**

```sql
SELECT Name FROM Customers
WHERE ID IN (SELECT CustomerID FROM Orders);
```

- **Compare with Aggregate:**

```sql
SELECT * FROM Products
WHERE Price > (SELECT AVG(Price) FROM Products);
```

10. Constraints

- **NOT NULL | UNIQUE | PRIMARY KEY | FOREIGN KEY | CHECK | DEFAULT**

11. Views, Triggers & Stored Procedures

- **View:**

```sql
CREATE VIEW HighValue AS
SELECT * FROM Customers WHERE TotalSpent > 10000;
```

- **Trigger:**

```sql
CREATE TRIGGER log_insert
AFTER INSERT ON Orders
FOR EACH ROW
INSERT INTO Logs VALUES ('Insert', NOW());
```

- **Procedure:**

```sql
CREATE PROCEDURE GetEmployee(IN empID INT)
BEGIN
  SELECT * FROM Employees WHERE ID = empID;
END;
```

SQL Questions - Link

Practice Questions

1. What is an attribute? List and define the different types.

Answer:

- **Simple attribute:** Cannot be decomposed further (e.g., *Age*).
- **Composite attribute:** Made up of multiple simple attributes (e.g., *FullName* → *FirstName*, *LastName*).
- **Multi-valued attribute:** Can hold multiple values for a single entity (e.g., *PhoneNumbers*).
- **Derived attribute:** Computed from another attribute (e.g., *Age* derived from *DateOfBirth*).
- **Key attribute:** Identifies entities uniquely (e.g., *RollNo*).

2. What are strong and weak entity sets? Give an example.

Answer:

- **Strong entity set:** Has a primary key to uniquely identify its instances (e.g., *Student* with *StudentID*).
- **Weak entity set:** Lacks its own primary key and relies on a related strong entity; has a partial key (discriminator) (e.g., *Dependent* in an insurance database, identified by *PolicyID* plus *DependentName* as discriminator).

3. Differentiate 1:1, 1:N, and M:N relationships, with examples.

Answer:

- **1:1 (One-to-One):** A person has one passport.
- **1:N (One-to-Many):** A department has many employees.
- **M:N (Many-to-Many):** Students enroll in multiple courses, and courses have multiple students (modeled via a *Enrollment* table).

4. What is a functional dependency? Define trivial and non-trivial dependencies.

Answer:
A functional dependency X → Y holds if rows with the same X also have the same Y.

- **Trivial:** Y is a subset of X (e.g., {A, B} → A)
- **Non-trivial:** Y is not a subset of X (e.g., A → B)

5. Explain candidate key, superkey, primary key, and composite key.

Answer:

- **Superkey:** Any set of attributes that uniquely identifies a tuple (e.g., {RollNo, Name}).
- **Candidate key:** Minimal superkey; no redundant attributes.
- **Primary key:** Chosen candidate key (e.g., *RollNo*).
- **Composite key:** Primary key involving multiple attributes (e.g., {CourseID, Semester}).

6. Define 1NF, 2NF, 3NF, and BCNF with examples.

Answer:

- **1NF:** Atomic values only (e.g., a table with columns *Roll, Subjects* should be split so *Subjects* is atomic).
- **2NF:** 1NF + no partial dependency on part of a composite key.
- **3NF:** 2NF + no transitive dependency (e.g., DeptID → DeptName, StudentID → DeptID; avoid going StudentID → DeptName transitively).
- **BCNF:** Every determinant is a candidate key (stronger than 3NF).

7. What is lossless decomposition? Why is dependency preservation important?

Answer:

- **Lossless decomposition:** Breaking a relation into smaller ones without losing information; can rejoin to get original data.
- **Dependency preservation:** Decomposed tables should still enforce original functional dependencies independently.

8. List the states of a transaction with brief definitions.

Answer:

1. **Active:** Transaction is executing.
2. **Partially Committed:** Last action executed; not saved permanently yet.
3. **Committed:** Changes stored successfully to the database.
4. **Failed:** Problem occurred; cannot continue.
5. **Aborted:** All changes rolled back.
6. **Terminated:** Transaction lifecycle has ended.

9. Explain ACID properties with examples.

Answer:

- **Atomicity:** Either all operations in a transaction occur or none do (e.g., both debit and credit in fund transfer).
- **Consistency:** DB remains in a valid state (e.g., constraints maintained).
- **Isolation:** Concurrent transactions don't interfere (e.g., serial equivalence).
- **Durability:** Once committed, changes persist (even on system crash).

10. What are recoverable, cascading, and strict schedules?

Answer:

- **Recoverable schedule:** No transaction commits before the one it depends on commits.
- **Cascading schedule:** One failure leads to cascading rollbacks among dependent transactions.
- **Strict schedule:** Transactions wait until writes are committed or aborted before reading/writing (avoids cascading).

SQL & Relational Algebra

11. Write SQL to find students scoring above average.

```
SELECT Name
FROM Students
WHERE Marks > (SELECT AVG(Marks) FROM Students);
```

12. Differentiate INNER, LEFT, RIGHT, and FULL joins with examples.

Answer:

- **INNER JOIN:** Returns matched rows only.
- **LEFT JOIN:** All rows from left table; NULLs for unmatched right.
- **RIGHT JOIN:** All rows from right table; NULLs for unmatched left.
- **FULL JOIN:** All rows from both tables; NULLs for unmatched sides.

13. Translate to relational algebra: Select StudentName and CourseID from Enrollment where grade = 'A'.

Answer:

$$\pi_{\text{StudentName, CourseID}}(\sigma_{\text{grade}='A'}(\text{Enrollment}))$$

14. When would you use UNION vs UNION ALL? Provide an example.

Answer:

- **UNION:** Removes duplicates.
- **UNION ALL:** Keeps duplicates (faster).

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers;

SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers;
```

15. What's the difference between clustered and non-clustered index?

Answer:

- **Clustered index:** Physically orders records in the table.
- **Non-clustered index:** Separate structure that points to rows; leaves data untouched.

16. How do you create a view? Provide an example.

```
CREATE VIEW HighScorers AS
SELECT Name, Marks
FROM Students
WHERE Marks > 90;
```

TL;DR Summary Table

Q1. What are prime and non-prime attributes? Give examples.

Answer:

- **Prime attribute** → An attribute that is part of at least one candidate key.
- **Non-prime attribute** → An attribute that is **not** part of any candidate key.

Example:
Relation: **R(A, B, C, D)**
Functional dependencies:

1. **A → B**
2. **C → D**
3. **AC → B**

Suppose **AC** is a candidate key.

- **Prime attributes** = A, C (because they are part of the candidate key AC).

- **Non-prime attributes** = B, D.

Q2. Why is the distinction important in normalization?

Answer:

- **2NF**: Eliminates partial dependencies where a **non-prime** attribute depends on part of a candidate key.
- **3NF**: Allows a dependency X → Y if X is a superkey **or** Y is a prime attribute.

Q3. Quick Check

Relation: R(P, Q, R, S)
FDs:

1. P → Q
2. QR → S
3. S → P

Find prime and non-prime attributes.

Solution:

1. Find candidate key(s):
    - Start with {P}: P → Q
      → Closure = {P, Q} (not all attributes)
    - Try {P, R}:
      {P, R} → Q (from P → Q)
      Then QR → S, so {P, Q, R} → S
      Then S → P gives no new attribute
      Closure = {P, Q, R, S} → Candidate key = {P, R}
2. Prime = {P, R}
3. Non-prime = {Q, S}.

Q4. Problem:

Given R(A, B, C, D, E) with FDs:

1. A → B
2. BC → D
3. D → E
4. E → A

Find all candidate keys and identify prime/non-prime attributes.

Step-by-Step Solution:

1. **Find closure of single attributes:**
    - $A^+ = \{A, B\} \rightarrow \{A, B\}$ (no further gain, since BC needed for D).
    - $B^+ = \{B\}$ (nothing extra).
    - $C^+ = \{C\}$ (nothing extra).
    - $D^+ = \{D, E, A, B\}$ ($E \rightarrow A$, $A \rightarrow B$) $\rightarrow \{D, E, A, B\}$ (missing C).
    - $E^+ = \{E, A, B\}$ (missing C, D).
2. **Test combinations:**
    - AC^+: $A \rightarrow B$, so $AC \rightarrow BC \rightarrow D \rightarrow E \rightarrow A \rightarrow \{A, B, C, D, E\}$ ☑ Candidate key.
    - DC^+: $D \rightarrow E \rightarrow A \rightarrow B$ gives $\{D, E, A, B, C\}$ ☑ Candidate key.
    - EC^+: $E \rightarrow A \rightarrow B$, so $EC \rightarrow BC \rightarrow D \rightarrow E \rightarrow \{A, B, C, D, E\}$ ☑ Candidate key.
3. **Candidate Keys: AC, DC, EC**
4. **Prime attributes:** A, C, D, E
   Non-prime attributes: B

Q5. Trick Question:

If a relation has only one candidate key, what can we say about its prime attributes?
Answer:

- All attributes in that candidate key are prime.
- All others are non-prime.