DBMS

- **Introduction to DBMS**
- **Data Models & Database Design**
- **Relational Model & Relational Algebra**
- **Structured Query Language (SQL)**
- **Database Design & Normalization**
- **Transaction Management**
- **Concurrency Control**
- **Indexing & Query Processing**
- **Recovery & Backup**
- **Advanced Topics in DBMS**

Introduction to DBMS

1.1 Introduction to Database & DBMS

- **Data** → Raw facts (e.g., “101”, “Alice”, “CS”).
- **Database** → Organized collection of related data.
    - Example: *Student Database* = Students, Courses, Results tables.
- **DBMS (Database Management System)** → Software that manages databases.
    - Examples: MySQL, PostgreSQL, Oracle, MongoDB.

Why DBMS?

- Provides **systematic way** to store, retrieve, and manage data.
- Without DBMS → files, manual searching, redundancy.
- With DBMS → queries (SQL), security, concurrency, backups.

Real-world analogy:
Database = Library books
DBMS = Librarian who organizes, finds, lends, and manages books.

1.2 File System vs DBMS

Feature	File System	DBMS
Data Storage	Flat files (txt, csv)	Tables/Collections
Redundancy	High (duplicate copies)	Reduced (relations, normalization)
Data Consistency	Hard to maintain	Ensured via constraints
Querying	Manual search/programs	SQL (powerful queries)
Security	Weak	Strong (GRANT, REVOKE)
Concurrency	Not supported	Supported (transactions)
Backup/Recovery	Manual	Automatic (checkpoints, logs)

Example:

- File System → Store `students.txt` and `courses.txt`. If Alice changes her email, must update in many files.
- DBMS → Single table `Student(StuID, Name, Email)`; change once, reflected everywhere.

Interview Tip:

Q: Why use DBMS over File System?
A: To reduce redundancy, maintain consistency, provide concurrency, ensure security, and support recovery.

1.3 Advantages & Disadvantages of DBMS

Advantages

- **Reduced redundancy** (data stored once).
- **Improved data sharing** (multi-user access).
- **Data consistency & integrity** (constraints, normalization).
- **Security** (authentication, authorization).
- **Backup & recovery support**.
- **Transaction management** (ACID properties).

Disadvantages

- **Cost**: Software + hardware expensive.
- **Complexity**: Requires trained personnel.
- **Performance**: Overhead compared to simple file system for small data.

1.4 DBMS Architecture

1-Tier Architecture

- DBMS + Application + User on same machine.
- Example: Running MySQL locally on your laptop.
- Used for learning/testing.

2-Tier Architecture

- **Client ↔ Server model**.
- Client (frontend app) communicates with DBMS directly.
- Example: Java application directly querying MySQL DB.

3-Tier Architecture

- **Client ↔ Application Server ↔ Database Server**.
- App server acts as middle layer → better scalability, security.
- Example: Web apps (Browser ↔ Django/Node.js ↔ MySQL).

Real-world analogy:

- **1-Tier**: You write your diary yourself.
- **2-Tier**: You ask a friend to fetch info directly from storage.
- **3-Tier**: You ask a receptionist → receptionist talks to storage → gives you info.

1.5 Levels of Abstraction in DBMS

1. **Physical Level (Internal Schema)**
   - *How* data is stored (indexes, files, B+ Trees).
   - Example: RollNo stored as 4-byte integer, Name as VARCHAR.
2. **Logical Level (Conceptual Schema)**
   - *What* data is stored + relationships.
   - Example: Student(RollNo, Name, Course)
3. **View Level (External Schema)**
   - User-specific view (subset of data).
   - Example: A professor only sees Student(RollNo, Name), not passwords.

Real-world analogy:

- Physical → The **books in racks** inside library storage.
- Logical → The **catalog** showing what books exist.
- View → The **reading list** given to a student.

1.6 Data Independence

Definition → Ability to change schema at one level without affecting other levels.

- **Physical Data Independence**
  - Change internal storage (e.g., indexing method) without changing tables.
- **Logical Data Independence**
  - Change logical schema (e.g., split Student table into Student + Department) without affecting views.

Example:

- If DB admin changes storage from SSD to cloud → Applications still run.
- If DBA adds a new attribute DOB in Student table → Existing views (Name, RollNo) still work.

Interview Tip:

Q: Which is harder to achieve — logical or physical data independence?
A: Logical, because changing schema affects applications more directly.

Quick Revision (Unit 1)

- DBMS = Software for managing databases (MySQL, Oracle, etc.).
- File System vs DBMS → DBMS solves redundancy, security, concurrency, recovery.
- Advantages → Security, consistency, backup, concurrency.
- Architecture → 1-Tier, 2-Tier, 3-Tier.
- Levels → Physical, Logical, View.
- Data Independence → Physical & Logical (logical is harder).

Practice Questions

Theory

1. What is a DBMS? How is it different from a File System?
2. Explain the 3-tier DBMS architecture with an example.
3. Differentiate between physical and logical data independence.

SQL/Practical

- Install MySQL or SQLite.
- Create a table Student(RollNo, Name, Course) and insert 3 records.
- Try retrieving only Name and Course → represents **View level** abstraction.

Data Models & Database Design

1. Data Models

A **data model** defines how data is represented, organized, and manipulated in a database.

Types of Data Models

1. **Hierarchical Model**
    - Data is represented in a **tree structure** (parent-child).
    - Each parent can have multiple children, but each child has only **one parent**.
    - Example:

        - Company
        - ├── Department
        - │ ├── Employee
        - │ └── Project
    - **Pros**: Fast retrieval for hierarchical data, simple to understand.
    - **Cons**: Difficult to modify structure, rigid relationships.

2. **Network Model**
    - Data is represented as **nodes** (entities) and **edges** (relationships).
    - A child can have **multiple parents** (many-to-many relationships).
    - Example: Student enrolled in multiple courses.
    - **Pros**: Supports complex relationships.
    - **Cons**: Complicated design & navigation.

3. **Relational Model**
    - Data is stored in **tables (relations)** with rows (tuples) and columns (attributes).
    - Relationships are represented using **keys**.

        - Example:
        - STUDENT(Student_ID, Name, Age, Course_ID)
        - COURSE(Course_ID, Course_Name, Credits)
    - **Pros**: Simple, flexible, widely used (SQL).
    - **Cons**: Performance issues with very large data.

4. **Object-Oriented Model**
    - Combines **object-oriented programming** with databases.
    - Data is stored as **objects** (with attributes & methods).
    - Example: A Car object with attributes (color, model) and methods (start(), stop()).
    - **Pros**: Better for multimedia and complex data.
    - **Cons**: Less widely used compared to relational model.

2. Entity-Relationship (ER) Model

A **high-level conceptual model** used for database design.

Entities

- **Entity**: Real-world object with properties.
  Example: Student, Course, Teacher.

Attributes

- Properties of an entity.
  Example: Student → (Roll_No, Name, Age).
- Types:
    - **Simple**: Cannot be divided further (e.g., Age).
    - **Composite**: Can be divided (e.g., Name → First Name + Last Name).
    - **Derived**: Derived from other attributes (e.g., Age from DOB).
    - **Multivalued**: Can have multiple values (e.g., Phone Numbers).

Relationships

- Association between entities.
- Example: Student **enrolls** in Course.

3. Keys

Keys uniquely identify tuples in a relation.

- **Super Key** → Any attribute(s) that uniquely identify a row.
  Example: {Roll_No, Name}
- **Candidate Key** → Minimal super key (no extra attribute).
  Example: {Roll_No}
- **Primary Key** → One candidate key chosen as main key.
  Example: Roll_No
- **Foreign Key** → Attribute that refers to primary key of another table.
  Example: Course_ID in STUDENT table refers to COURSE table.

4. Strong vs Weak Entities

- **Strong Entity**: Has its own primary key.
  Example: Student(Student_ID, Name)
- **Weak Entity**: Cannot be uniquely identified without another entity.
  Example: Dependent(Name, Age, Student_ID) depends on Student.

5. Enhanced ER (EER) Model

Adds advanced features to ER model.

Specialization

- Dividing an entity into **sub-entities** based on unique characteristics.
- Example: Employee → Manager, Clerk.

Generalization

- Combining multiple entities into a **super-entity**.
- Example: Car, Bike → generalized into Vehicle.

Aggregation

- Treating a **relationship** as an entity for another relationship.
- Example: A relationship Works_On (between Employee & Project) can be treated as an entity to connect with Department.

6. Database Design Example

Example 1: Banking System

Entities:

- Customer(Customer_ID, Name, Address)
- Account(Account_ID, Balance)
- Loan(Loan_ID, Amount)

Relationships:

- Customer **owns** Account
- Customer **takes** Loan

Example 2: University Database

Entities:

- Student(Student_ID, Name, Age)
- Course(Course_ID, Course_Name)
- Instructor(Instructor_ID, Name)

Relationships:

- Student **enrolls** in Course
- Instructor **teaches** Course

7. ER to Relational Schema Conversion

Steps:

1. **Entity → Table**
    - Student(Student_ID, Name, Age)
2. **Relationship → Table or Foreign Key**
    - Many-to-One: Add foreign key.
    - Many-to-Many: Create separate relation.
3. **Weak Entity → Table with Foreign Key of Strong Entity**

    Example:

    ER Diagram →

    Student — enrolls — Course

    Converted Schema:

    Student(Student_ID, Name, Age)
    Course(Course_ID, Course_Name)
    Enrollment(Student_ID, Course_ID)

Relational Model & Relational Algebra

1. Relational Model Concepts

The **Relational Model** (introduced by E.F. Codd in 1970) is the most widely used data model. It organizes data into **relations (tables)** consisting of **rows (tuples)** and **columns (attributes).**

Key Concepts:

- **Relation** → A table with rows and columns.
- **Attribute** → A column in the table (defines the type of data).
- **Tuple** → A row in the table (represents a record).
- **Domain** → The set of possible values for an attribute.
- **Schema** → The structure/definition of the database.
- **Instance** → The actual data stored in the database at a given time.

Example:
Relation: **STUDENT**

STUDENT(SID, Name, Age, Dept)

- Schema: STUDENT(SID, Name, Age, Dept)
- Attributes: SID, Name, Age, Dept
- Tuples:
  (101, 'Alice', 20, 'CSE')
  (102, 'Bob', 22, 'ECE')

Schema = blueprint,
Instance = data at a specific time.

2. Keys & Constraints

Keys are used to uniquely identify tuples and establish relationships between tables.

Types of Keys:

1. **Super Key** → Any set of attributes that uniquely identifies a tuple.
   Example: {SID}, {SID, Name}
2. **Candidate Key** → Minimal super key (no unnecessary attributes).
   Example: {SID}
3. **Primary Key** → Chosen candidate key for identification.
   Example: SID
4. **Alternate Key** → Candidate keys not chosen as primary key.
   Example: (Name) could be alternate if unique.

5. **Foreign Key** → Attribute in one relation that refers to the primary key of another relation.
   Example: DeptID in STUDENT references DeptID in DEPARTMENT.

Constraints:

- **Domain Constraint** → Attribute values must be from the defined domain.
- **Entity Integrity Constraint** → Primary key cannot be NULL.
- **Referential Integrity Constraint** → Foreign key must match an existing value in referenced relation.

3. Relational Algebra Operations

Relational Algebra = Formal query language for relational databases.
It uses **operators** to retrieve/modify data.

Basic Operations:

1. **Select (σ)** → Filters rows based on condition.
   σ_{Age>20}(STUDENT) → Selects students with Age > 20.
2. **Project (π)** → Extracts specific columns.
   π_{Name,Dept}(STUDENT) → Returns only Name and Dept.
3. **Union (∪)** → Combines tuples from two relations (removes duplicates).
   Example: CSE_Students ∪ ECE_Students
4. **Intersection (∩)** → Common tuples from two relations.
   Example: CSE_Students ∩ Hostel_Students
5. **Difference (–)** → Tuples in one relation but not in another.
   Example: CSE_Students – ECE_Students

Join Operations:

1. **Inner Join (⋈)** → Combines tuples where condition matches.
   Example: STUDENT ⋈ DEPARTMENT (on DeptID)
2. **Natural Join (⋈)** → Automatically joins relations using attributes with the same name.
3. **Outer Join** → Keeps unmatched tuples as well:
   - **Left Outer Join** → Keeps all tuples from left relation.
   - **Right Outer Join** → Keeps all tuples from right relation.
   - **Full Outer Join** → Keeps all tuples from both, with NULL for missing.

Division (÷)

Used for queries like "for all".
R ÷ S → Finds tuples in R that are related to *all* tuples in S.

Example:

- R(Student, Course)
- S(Course) = {DBMS, OS}
- R ÷ S → Students who took **both DBMS and OS**.

4. Relational Calculus

Declarative query language → *What to retrieve*, not *How*.
Two types:

1. **Tuple Relational Calculus (TRC):**
   Queries expressed with variables representing tuples.
   Example:
   { t | t ∈ STUDENT ∧ t.Age > 20 }
   → All students with Age > 20.
2. **Domain Relational Calculus (DRC):**
   Queries expressed using domain variables (values of attributes).
   Example:
   { <n,d> | ∃ sid, age (STUDENT(sid, n, age, d) ∧ age > 20) }
   → Names & Depts of students with Age > 20.

At the End:

- Relational model defines **schema, tuples, attributes, domains**.
- **Keys & constraints** ensure data integrity.
- **Relational algebra** provides operators (σ, π, ∪, ∩, – , ⋈, ÷).
- **Relational calculus** is declarative, with TRC and DRC.

Structured Query Language (SQL)

1. SQL Basics

- SQL → Structured Query Language, standard language to interact with relational databases.
- Categories of SQL commands:
  - **DDL** – Define schema/structure
  - **DML** – Manipulate data
  - **DCL** – Control permissions
  - **TCL** – Control transactions

2. Data Definition Language (DDL)

- **CREATE** – Create tables/databases.

```
CREATE TABLE Student (
   ID INT PRIMARY KEY,
   Name VARCHAR(50),
   Age INT
);
```

- **ALTER** – Modify schema.

```
ALTER TABLE Student ADD Email VARCHAR(100);
```

- **DROP** – Delete table/schema.

```
DROP TABLE Student;
```

3. Data Manipulation Language (DML)

- **INSERT** – Add data.

```
INSERT INTO Student VALUES (1, 'Alice', 20);
```

- **UPDATE** – Modify data.

```
UPDATE Student SET Age = 21 WHERE ID = 1;
```

- **DELETE** – Remove data.

```
DELETE FROM Student WHERE ID = 1;
```

4. Data Control Language (DCL)

- **GRANT** – Provide access rights.

```
GRANT SELECT ON Student TO user1;
```

- **REVOKE** – Remove rights.

```
REVOKE SELECT ON Student FROM user1;
```

5. Transaction Control Language (TCL)

- **COMMIT** – Save changes permanently.
- **ROLLBACK** – Undo changes since last COMMIT.
- **SAVEPOINT** – Mark a point to rollback partially.

6. Constraints in SQL

- **NOT NULL** → Disallows NULL values.
- **UNIQUE** → Ensures unique values.
- **PRIMARY KEY** → Unique + NOT NULL.
- **FOREIGN KEY** → Enforces referential integrity.
- **CHECK** → Ensures condition on column.
- **DEFAULT** → Provides default value.

7. Advanced SQL

Joins

- **INNER JOIN** → Matching rows.
- **LEFT JOIN** → All rows from left + matched right.
- **RIGHT JOIN** → All rows from right + matched left.
- **FULL JOIN** → All rows from both (matched + unmatched).

Nested Queries / Subqueries

- Query inside another query.

```
SELECT Name FROM Student
WHERE Age = (SELECT MAX(Age) FROM Student);
```

Views

- Virtual table from query.

```
CREATE VIEW YoungStudents AS
SELECT * FROM Student WHERE Age < 21;
```

Indexes

- Speed up searches.

```
CREATE INDEX idx_name ON Student(Name);
```

Aggregate Functions

- **SUM, AVG, COUNT, MIN, MAX**

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

GROUP BY & HAVING

- **GROUP BY** – Group rows.
- **HAVING** – Filter after grouping.

```
SELECT Age, COUNT(*)
FROM Student
GROUP BY Age
HAVING COUNT(*) > 2;
```

Set Operations

- **UNION** → Combines results (removes duplicates).
- **INTERSECT** → Common rows.
- **EXCEPT** → Rows in first query, not in second.

Database Design & Normalization

1. Functional Dependencies (FDs)

FDs describe the relationship between attributes in a relation.
If $X \rightarrow Y$, then knowing X determines Y.

- **Full FD**: All attributes in X are necessary to determine Y.
  Example: (RollNo, Subject) → Marks (both needed).
- **Partial FD**: Only part of X is enough to determine Y.
  Example: (RollNo, Subject) → StudentName (only RollNo is enough, so it's partial).
- **Transitive FD**: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.
  Example: RollNo → DeptID, DeptID → DeptName ⇒ RollNo → DeptName.
- **Trivial FD**: If Y is a subset of X.
  Example: (RollNo, Name) → RollNo.

2. Armstrong's Axioms (Rules to derive new FDs)

1. **Reflexivity**: If $Y \subseteq X$, then $X \rightarrow Y$.
2. **Augmentation**: If $X \rightarrow Y$, then $XZ \rightarrow YZ$.
3. **Transitivity**: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.
4. **Union**: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$.
5. **Decomposition**: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$.
6. **Pseudo-Transitivity**: If $X \rightarrow Y$ and $WY \rightarrow Z$, then $XW \rightarrow Z$.

These rules are used when checking normalization or attribute closure.

3. Attribute Closure (X^+)

- The **set of all attributes functionally determined by X** using FDs.
- Useful for finding **candidate keys**.

 Example:
Relation R(A, B, C, D)
FDs: $A \rightarrow B$, $B \rightarrow C$, $A \rightarrow D$

- Closure of A: $A^+ = \{A, B, C, D\}$ → So A is a **candidate key**.

4. Decomposition

Breaking a relation into smaller relations to reduce redundancy.

- **Lossless Join**: After decomposition, when we join back the tables, we get the original relation (no data loss).
  Example: R(A, B, C) with FD: A → B.
  Decompose into R1(A, B) and R2(A, C) → Lossless.
- **Dependency Preserving**: After decomposition, all original FDs should still be enforceable without needing to join tables.

5. Normalization (step-by-step)

Goal: Reduce redundancy & anomalies (insertion, deletion, update).

- **1NF (First Normal Form)**:
  - Remove multivalued attributes (atomic values only).
    Example: {Subjects = [Math, Physics]}
    Fix: Break into separate rows
- **2NF (Second Normal Form)**:
  - Must be in 1NF.
  - No **Partial FDs** (non-prime attribute depending only on part of key).
- **3NF (Third Normal Form)**:
  - Must be in 2NF.
  - No **Transitive FDs**.
    Example: RollNo → DeptID, DeptID → DeptName
    Move DeptName to separate table.
- **BCNF (Boyce-Codd Normal Form)**:
  - Stronger than 3NF.
  - Every determinant must be a candidate key.
- **4NF (Fourth Normal Form)**:
  - Must be in BCNF.
  - No multivalued dependencies (separate independent facts into different tables).

Practical Example (Student-Subject Database)

Unnormalized Table (UNF):

Student(RollNo, Name, Subjects, DeptName)
1, Ravi, {Math, Physics}, CSE
2, Anu, {Math, Chemistry}, ECE

Problems: Multivalued attributes, redundancy.

1NF (Atomic values):

RollNo	Name	Subject	DeptName
1	Ravi	Math	CSE
1	Ravi	Physics	CSE
2	Anu	Math	ECE
2	Anu	Chemistry	ECE

2NF (Remove partial dependencies):

- Split into Student(RollNo, Name, DeptName) and Enrollment(RollNo, Subject)

3NF (Remove transitive dependencies):

- Move DeptName into separate Department(DeptID, DeptName)

Final Tables:

- Student(RollNo, Name, DeptID)
- Department(DeptID, DeptName)
- Enrollment(RollNo, Subject)

Now the design is **normalized, less redundancy, and avoids anomalies.**

Transaction Management

1) What is a Transaction?

A **transaction (T)** is a sequence of database operations that represents **one logical task**.
Core actions: READ(x), WRITE(x), COMMIT, ABORT/ROLLBACK.

Everyday examples

- Bank transfer (debit A, credit B)
- E-commerce order (create order → reserve inventory → charge payment)
- Ticket booking (check seat → hold seat → confirm → pay)

SQL shape

```
BEGIN;  -- or START TRANSACTION
-- your statements
COMMIT; -- or ROLLBACK;
```

2) ACID Properties (Reliability in practice)

A. Atomicity ("all or nothing")

Either **all** statements in T happen, or **none** do.

Practical pattern

```
BEGIN;
  UPDATE accounts SET balance = balance - 1000 WHERE id = 1; -- A
  UPDATE accounts SET balance = balance + 1000 WHERE id = 2; -- B
COMMIT; -- if any UPDATE fails, ROLLBACK instead
```

If the second UPDATE fails, the DB **rolls back** the first so no money disappears.

B. Consistency (valid → valid)

Transactions must preserve **constraints** and business rules.

Example rules

- balance >= 0 (CHECK)
- FOREIGN KEY must exist
- Sum of balances before = after transfer

If a statement violates a constraint, the DB **rejects** it and the transaction fails.

C. Isolation (no interference)

Running T1 and T2 together should **appear** as if they ran one-by-one (depending on isolation level).
Typical read problems:

- **Dirty read**: read uncommitted data
- **Non-repeatable read**: same row read twice → different values
- **Phantoms**: re-running a range query returns different **sets** of rows

SQL isolation levels (for intuition)

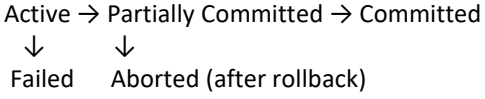
- Read Uncommitted → can see uncommitted writes (rarely allowed)
- Read Committed → no dirty reads (default on many DBs)
- Repeatable Read → stable row reads (no non-repeatable reads)
- Serializable → behaves like a serial schedule (strongest)

(Exact behavior varies slightly by DB engine; details come in Concurrency Control.)

D. Durability (committed = permanent)

After COMMIT, data survives power loss/crash (via logs, WAL, checkpoints).
You can assume: **if you saw COMMIT succeed, your data is safe**.

3) Transaction States (with what triggers each move)



- **Active**: executing statements
- **Partially Committed**: last statement done; about to commit
- **Committed**: success; changes durable
- **Failed**: error/exception detected
- **Aborted**: rolled back; DB restored

SQL mapping

- BEGIN → Active
- COMMIT succeeds → Committed
- Error or ROLLBACK → Aborted

4) Schedules (how interleaving happens)

A **schedule** is an interleaving of operations from multiple transactions.

Serial vs Concurrent

- **Serial**: T1 finishes completely, then T2 starts. (Always safe, often slow)
- **Concurrent**: Interleaved operations of T1, T2… (Fast, needs care)

Classic anomalies (what goes wrong without control)

- **Lost update**: T1 and T2 overwrite each other's writes
- **Dirty read**: T2 reads a value T1 hasn't committed yet
- **Dirty write**: T2 overwrites a value T1 hasn't committed yet
- **Non-repeatable read**: T1 re-reads a row and sees a different value
- **Phantom**: T1's range query returns a different **set** of rows on re-run

Tiny lost-update demo (two sessions):

Session A

```
BEGIN;
SELECT balance FROM accounts WHERE id = 1; -- returns 5000
-- think time...
UPDATE accounts SET balance = 4800 WHERE id = 1; -- 5000-200
-- (not committed yet)
```

Session B (in parallel)

```
BEGIN;
SELECT balance FROM accounts WHERE id = 1; -- also 5000
UPDATE accounts SET balance = 4700 WHERE id = 1; -- 5000-300
COMMIT;
```

Session A now commits its earlier 4800 → **B's 4700 is lost**.
(Proper locking/serialization prevents this; we'll cover mechanisms next unit.)

5) Serializable Schedules (the correctness gold standard)

We want concurrent schedules that **behave like** some **serial** order.

A) Conflict Serializability (most used test)

Two operations **conflict** if:

- they are on the **same item** and
- at least **one** is a **write**.
  (So RW, WR, WW on the same item conflict; RR doesn't.)

A schedule is **conflict-serializable** if you can swap **non-conflicting** operations to reach a **serial order**.

How to test (Precedence / Serialization Graph):

1. Nodes = transactions (T1, T2, …)
2. For every **conflict** where operation of Ti occurs **before** conflicting op of Tj on the same item, draw **Ti → Tj**
3. If the graph has **no cycle**, the schedule is conflict-serializable.
   If there's a **cycle**, it's **not** conflict-serializable.

Mini example

Schedule S:

```
T1: R(A)     W(A)
T2:    R(A)         W(B)
T3:         W(A)
```

Conflicts:

- T1:W(A) before T2:R(A) ⇒ T1 → T2
- T1:W(A) before T3:W(A) ⇒ T1 → T3
- T3:W(A) after T2:R(A) has no conflict on A? (R vs W conflicts; but order is T2:R(A) before T3:W(A) ⇒ T2 → T3)

Graph edges: T1→T2, T1→T3, T2→T3 → **acyclic** → conflict-serializable with order T1 → T2 → T3.

B) View Serializability (broader but harder to test)

Two schedules are **view-equivalent** if:

1. **Reads-from**: Every read sees the same writer in both schedules.

2. **Initial reads**: If a read sees the **initial value** in one schedule, it does in the other.
3. **Final writes**: The transaction that does the **last write** on each item is the same in both.

- All **conflict-serializable** schedules are **view-serializable**, but not vice versa.
- **Blind writes** (write without reading current value) can make a schedule view-serializable but **not** conflict-serializable.

Quick blind-write example

```
S:
T1: W(A)     (no read of A)
T2:    W(A)
```

This may be **view-serializable** (same final writer) but **not** conflict-serializable if edges create a cycle.

6) Recoverability of Schedules (so rollbacks don't explode)

We care about **who read whose writes** and **commit order**.

A) Recoverable

If Tj **reads** a value written by Ti, then **Ti must commit before Tj** can commit.
This avoids Tj committing based on data that might be rolled back.

B) Cascadeless

No transaction ever **reads uncommitted** data.
(Prevents **cascading rollbacks** entirely.)

C) Strict (strongest and preferred)

No transaction may **read or write** a data item **written by an uncommitted** transaction.

- Guarantees **no dirty reads** and **no dirty writes**.
- Makes recovery easy: if a transaction aborts, nobody touched its dirty data.

Why it matters (tiny example)

Bad schedule (not recoverable):

```
T1: W(A) ... (no commit yet)
T2:   R(A) ... COMMIT
     ↑ read uncommitted
T1: ABORT
```

T2 committed based on a value that **vanished** → **not recoverable**.

Cascadeless schedule:

```
T1: W(A)
T1: COMMIT
T2:   R(A)   -- reads only committed data
```

Strict schedule:

```
T1: W(A)
-- nobody else can read/write A here --
T1: COMMIT
T2:   R(A)
```

In practice, **Strict Two-Phase Locking (Strict 2PL)** or **MVCC with appropriate locks** yields **strict** (or at least cascadeless) schedules.

7) Hands-on: try these in SQL

You can run these on MySQL/PostgreSQL/SQLite. Open **two sessions** to feel the interleaving.

Setup

```
CREATE TABLE accounts (
  id INT PRIMARY KEY,
  balance INT NOT NULL CHECK (balance >= 0)
);

INSERT INTO accounts VALUES (1, 5000), (2, 5000);
```

A. Clean transfer (ACID)

```
BEGIN;
UPDATE accounts SET balance = balance - 1000 WHERE id = 1;
UPDATE accounts SET balance = balance + 1000 WHERE id = 2;
COMMIT;
```

B. Observe a dirty read (if your DB allows it — e.g., set isolation to READ UNCOMMITTED)

Session 1

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; -- if supported
BEGIN;
UPDATE accounts SET balance = balance - 500 WHERE id = 1;
-- don't commit yet
```

Session 2

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; -- if supported
BEGIN;
SELECT balance FROM accounts WHERE id = 1; -- may show 4500 (dirty)
COMMIT;
```

Now ROLLBACK in Session 1. Session 2 **saw** a value that never actually committed.

C. Savepoints (partial rollback)

```
BEGIN;
UPDATE accounts SET balance = balance - 200 WHERE id = 1;
SAVEPOINT s1;
UPDATE accounts SET balance = balance + 200 WHERE id = 2; -- oops wrong id?
ROLLBACK TO s1; -- undo second step only
COMMIT;        -- keeps the first update
```

8) Quick mental model & exam/interview bullets

- A transaction is **atomic**, **consistent**, **isolated**, **durable**.
- **Schedules** interleave operations; correctness target = **serializable**.
- **Conflict serializability** → build a **precedence graph**; **cycle = bad**.
- **View serializability** is broader; **blind writes** can be view-serializable but not conflict-serializable.
- Prefer **strict** (or at least **cascadeless**) schedules for easy **recovery**.
- Most real DBs use **locking** or **MVCC** to enforce isolation (details in next unit).

9) Practice (with answers outlined)

1. **Classify**: Dirty read, Non-repeatable read, Phantom

- You read a row that later gets rolled back → **Dirty read**
- You read the same row twice, values differ → **Non-repeatable**
- You run the same query twice and the **number of rows** changes → **Phantom**

2. **Is this schedule recoverable?**

```
T1: W(A) ----
T2:   R(A) ------- COMMIT
T1:             ABORT
```

Answer: **Not recoverable** (T2 committed before T1, whose value it read).

3. **Conflict-serializable?** Build graph for:

1) T1: R(A)
2) T2: W(A)
3) T1: W(A)
4) T2: R(B)
5) T1: W(B)

Conflicts:

- (2) T2:W(A) vs (1) T1:R(A) → T1→T2
- (3) T1:W(A) vs (2) T2:W(A) → T2→T1 (**cycle**)
  → **Not** conflict-serializable.

Concurrency Control

Concurrency control ensures that multiple transactions can execute **simultaneously** in a database **without leading to inconsistencies**.

Concurrency Problems

1. **Dirty Read**
    - Occurs when one transaction reads data written by another transaction **that has not yet been committed**.
    - If the second transaction rolls back, the first transaction has read **invalid data**.

    Example:
    - T1 updates a balance = 1000 → 900 (not committed).
    - T2 reads balance = 900.
    - T1 rolls back.
    - T2 has read a "dirty" value that never really existed.

2. **Lost Update**
    - Happens when two transactions overwrite each other's updates.

    Example:
    - T1 reads balance = 1000.
    - T2 reads balance = 1000.
    - T1 updates balance = 1000 – 200 = 800.
    - T2 updates balance = 1000 – 100 = 900.
    - Final balance = 900 (T1's update is lost).

3. **Phantom Read**
    - When a transaction re-executes a query and finds **new rows inserted by another transaction**.

    Example:

    - T1: SELECT * FROM students WHERE marks > 80; → returns 10 rows.
    - T2 inserts a new student with marks = 85 and commits.
    - T1 runs the same query again → now 11 rows appear (a "phantom" record).

Lock-based Protocols

Locks ensure that only **one transaction can access a resource** in a conflicting way at a time.

1. **Two-Phase Locking (2PL)**
    - Every transaction has **two phases**:
        - **Growing phase**: Can acquire locks, but not release.
        - **Shrinking phase**: Can release locks, but not acquire new ones.
    - Guarantees **conflict-serializability**.

    Example:

    - T1 acquires lock on balance, updates it, then releases.
    - T2 must wait until T1 releases lock.

2. **Strict 2PL**
    - Locks are **not released until commit/abort**.
    - Prevents **dirty reads**.

    Example:

    - T1 updates balance and holds lock until commit.
    - T2 cannot read until T1 commits → ensures consistency.

3. **Rigorous 2PL**
    - Even stricter: **all locks (read + write) held until commit**.
    - Produces **strict schedules** (easy to recover).

Timestamp-based Protocols

- Each transaction gets a **unique timestamp**.
- Ensures that transactions follow **timestamp order**.

Example:

- T1 (timestamp = 5)
- T2 (timestamp = 10)
- If T2 tries to write before T1 finishes, it waits/aborts depending on rules.

Thomas Write Rule

- A modification of timestamp ordering.
- If a transaction's write is **outdated (too old)**, it is simply ignored (not aborted).

Example:

- T1 (TS=5) writes x=100.
- T2 (TS=10) writes x=200.
- If T1 tries again to write x=150, it is ignored (since T2 already has a newer timestamp).

Deadlock in DBMS

A deadlock occurs when two or more transactions **wait for each other's resources indefinitely**.

Example:

- T1 locks Account A and waits for Account B.
- T2 locks Account B and waits for Account A.
- Both are stuck → **Deadlock**.

Deadlock Prevention

- Avoid creating cycles in the wait graph.
- Techniques:
  - **Wait-Die**: Older transaction waits, younger aborts.
  - **Wound-Wait**: Older transaction preempts younger.

Deadlock Detection

- Build a **wait-for graph**.
- If a cycle exists → deadlock detected.

Deadlock Recovery

- Once detected, break deadlock by **aborting one transaction** (usually the youngest or least costly one).

- **Problems**: Dirty Read, Lost Update, Phantom Read.
- **Lock Protocols**: 2PL, Strict 2PL, Rigorous 2PL.
- **Timestamp Protocols**: Ensure order based on time.
- **Thomas Write Rule**: Ignore outdated writes.
- **Deadlocks**: Prevent, Detect, Recover.

Indexing & Query Processing

1. Types of Indexing

(a) Ordered Indices

Indexes are like **book indexes** – they help us quickly locate data without scanning the entire database.

- **Primary Index**
  - Built on **primary key**.
  - Records in data file are stored in sorted order of primary key.
  - Each index entry points to a **block/page** in memory.
  - Example: If students are stored sorted by Roll_No, index helps jump to correct block.
- **Secondary Index**
  - Built on **non-primary attributes**.
  - Data file is not ordered on that attribute.
  - Example: Searching students by Name.
- **Clustering Index**
  - Built when table is **physically ordered** on a non-primary attribute.
  - Example: Employees grouped by Department_ID.

(b) Multi-level Indices

- If index file itself becomes too big → create an **index on index**.
- Works like a **dictionary with sub-dictionaries**.
- Example:
- Level 1 (Main Index) → Points to Level 2
- Level 2 (Secondary Index) → Points to Data Blocks

(c) Dynamic Multilevel Indices

- **B-Trees**: Balanced tree structure. Data can be stored at **internal + leaf nodes**.
- **B+ Trees**: Special case of B-Tree.
  - Data is stored **only in leaf nodes**.
  - Leaf nodes are **linked as a linked list** → efficient range queries.
- Example (Bank DB): If you want all accounts with balance between 10,000 – 50,000, a **B+ Tree** allows quick range search.

In practice, **most databases (MySQL, Oracle, Postgres)** use **B+ Trees** for indexing.

2. Query Processing

When you run a SQL query like:

```
SELECT name FROM Students WHERE roll_no = 202;
```

The DBMS doesn't just "run it" — it processes it in **steps**:

Steps in Query Processing:

1. **Parsing & Translation**
   - SQL → Internal query tree
   - Example: SELECT parsed into operations: σ(roll_no=202)(Students)

2. **Optimization**
    - DBMS checks **different ways** to execute query.
    - Example:
        - Option 1: Scan whole table (slow).
        - Option 2: Use index on roll_no (fast).
    - Chooses the **cheapest execution plan** (based on cost).
3. **Evaluation / Execution**
    - Query is actually executed using the chosen plan.
    - Data fetched from disk using indices.

3. Query Optimization (Basic Idea)

- DBMS tries to find the **most efficient way** to execute a query.
- Example:
  Query:
- SELECT name FROM Employee WHERE dept = 'IT' AND salary > 50000;
  Possible Plans:
    - Scan entire Employee table → check conditions.
    - Use **index on dept** → filter IT employees, then check salary.
    - Use **composite index on (dept, salary)** → direct fetch in one step.

DBMS will choose the **plan with least cost** (measured in disk I/O, CPU, memory).

Practical Example with Indexing

Without Index

```
SELECT * FROM Students WHERE name = 'Rahul';
```

- DBMS scans **all rows** (O(n)).
- If 1 million rows → very slow.

With Secondary Index on **name**

```
CREATE INDEX idx_name ON Students(name);
```

- DBMS goes to **index file** → finds Rahul's entry → directly jumps to record.
- Time complexity → O(log n).

Indexing = Improves search performance (Primary, Secondary, Clustering, Multi-level, B-Tree, B+ Tree).

Query Processing = Parsing → Optimization → Execution.

Query Optimization = Choose lowest-cost execution plan using indices.

Recovery & Backup

Failure Classification

Failures in DBMS can occur at different levels. Major types:

1. **Transaction Failure**
    - When a transaction cannot complete successfully due to logical errors (e.g., division by zero) or system errors.
    - Example: A money transfer transaction deducts money from one account but fails before crediting to another.
2. **System Crash**
    - Hardware/software failure causes the system to stop unexpectedly.
    - Example: Power failure while updating student grades.
3. **Disk Failure**
    - Data on disk becomes inaccessible due to corruption or physical damage.
    - Example: Hard disk crash where the database file is lost.

Recovery Techniques

1. **Deferred Update (No-Undo/Redo)**
    - Updates are recorded in the log but **not written to the database** until the transaction is committed.
    - If crash occurs **before commit** → simply discard.
    - If crash occurs **after commit** → redo from log.
    - Advantage: Easy to implement.
    - Drawback: Higher redo cost.

    Practical Example:

    - Transaction T1 wants to add ₹500 to account A.
    - Log: <T1, write A = A+500> (but DB not updated yet).
    - If crash occurs before commit → ignore T1.
    - If crash occurs after commit → redo and update DB.

2. **Immediate Update (Undo/Redo)**
    - Changes are written to DB **before the transaction commits**.
    - If system crashes:
        - Undo → For uncommitted transactions.
        - Redo → For committed transactions.
    - Advantage: Fast commit.
    - Drawback: More complex recovery.

Example:

- T2 deducts ₹200 from account B.
- DB updated immediately.
- If crash occurs before commit → must **undo** deduction.
- If crash occurs after commit → **redo** if not fully applied.

3. **Shadow Paging**
   - Instead of directly updating pages, a **shadow copy** of the database is maintained.
   - On commit → switch to new shadow pages.
   - On crash → old shadow pages are used (safe).
   - Advantage: No need for complex undo/redo.
   - Drawback: Expensive for large DB.

Example:

- Current DB page: P1.
- Shadow page P1' is created.
- Updates done on P1'.
- If commit → replace old with P1'.
- If crash → keep old P1 (safe).

4. **Checkpoints**
   - A checkpoint is like a **snapshot** of the DB at a safe state.
   - During recovery:
     - Start from the last checkpoint instead of scanning the whole log.
   - Advantage: Faster recovery.
   - Drawback: Takes extra system resources.

Example:

- Checkpoint taken at 10:00 AM.
- Crash at 10:30 AM.
- Recovery only considers logs after 10:00 AM.

Backup & Recovery in Real Life

- **Full Backup** → Copy of entire DB (daily/weekly).
- **Incremental Backup** → Copy only changes since last backup.
- **Log-based Recovery** → Uses transaction logs + checkpoints.

Example:
Banking systems take daily **full backup** at midnight and **incremental backups** every hour. If a crash occurs at 3 PM → restore midnight backup + apply logs until 3 PM.

- Failures: Transaction, System Crash, Disk.
- Recovery Techniques: Deferred Update, Immediate Update, Shadow Paging, Checkpoints.
- Backups ensure data safety and faster recovery.

Advanced Topics in DBMS

1. Distributed Databases (DDBMS)

A **distributed database** is a collection of multiple, logically interrelated databases distributed across different physical locations (sites), connected by a network.

Concepts

- Data is not stored in a single server; instead, it is spread across multiple sites.
- Appears as **one single database** to the user.
- Managed by **Distributed Database Management System (DDBMS)**.

Advantages

- **Improved Availability:** If one site fails, others can still provide data.
- **Improved Performance:** Queries can be executed closer to the location of data.
- **Scalability:** Easy to add new servers.
- **Transparency:** Location transparency (user doesn't care where data is stored).

Challenges

- **Data Replication:** Keeping multiple copies consistent.
- **Network Dependency:** System performance depends on communication links.
- **Concurrency Control:** Managing transactions across multiple sites.
- **Complex Recovery:** Handling failures in a distributed environment.

Example: Google Spanner, Amazon Aurora (distributed SQL databases).

2. NoSQL Databases

NoSQL (Not Only SQL) databases are designed for scalability, flexibility, and handling unstructured/semi-structured data.

Types of NoSQL Databases

1. **Key-Value Stores (Redis)**
    - Stores data as key-value pairs.
    - **Use case:** Caching, session storage, real-time leaderboards.
    - **Example:** userID → {name: "Mazid", age: 22}

2. **Document-Oriented (MongoDB)**
    - Stores data in **JSON-like documents**.
    - Flexible schema → no need to predefine table structure.
    - **Use case:** Content management, product catalogs.
    - **Example:**

```
{
  "name": "Laptop",
  "brand": "Dell",
  "price": 55000
}
```

3. **Column-Oriented (Cassandra, HBase)**
    - Stores data in **columns** instead of rows.
    - Efficient for large-scale analytics.
    - **Use case:** IoT data storage, time-series data.
    - **Example:**

```
ID | Name   | Age | City
--------------------------
1  | Rohit  | 22  | Hyderabad
2  | Mazid  | 21  | Vijayawada
```

Key Point: Unlike RDBMS, NoSQL databases **don't require strict schema** and are optimized for **horizontal scaling**.

3. Big Data Integration with DBMS

- Traditional DBMS cannot efficiently handle **petabytes of unstructured data**.
- Big Data frameworks (like **Hadoop, Spark**) integrate with DBMS for:
    - **Storage** (HDFS, NoSQL databases).
    - **Analytics** (Hive, Pig).
    - **Real-time processing** (Kafka + Cassandra).

Example: E-commerce sites analyzing billions of user clicks + transactions.

4. Cloud Databases

- Databases provided as a **service on the cloud** (DBaaS).
- Users don't worry about installation, scaling, backup.

Examples

- **AWS RDS** – Relational database service (supports MySQL, PostgreSQL, Oracle).
- **Google BigQuery** – Analytics-focused, handles huge datasets.
- **Azure Cosmos DB** – Multi-model, globally distributed.

Advantages

- Auto-scaling.
- High availability (replication across regions).
- Pay-per-use model.

Use case: Startups that want to avoid maintaining on-premise servers.

5. NewSQL Databases

- A new class of databases that **combine scalability of NoSQL with consistency of SQL**.
- Supports **ACID properties** + **SQL queries** + **high scalability**.

Examples

- Google Spanner
- CockroachDB
- VoltDB

Use case: Banking, E-commerce (where both scalability and strong consistency are critical).

- **DDBMS**: Distributed, improves performance, but adds complexity.
- **NoSQL**: Key-Value (Redis), Document (MongoDB), Column-Oriented (Cassandra).
- **Big Data + DBMS**: Used for massive-scale analytics.
- **Cloud DB**: Pay-per-use, managed by providers (AWS RDS, BigQuery).
- **NewSQL**: Best of SQL + NoSQL.

In short:

- **RDBMS** → Best for structured, transactional data (banking, ERP).
- **NoSQL** → Best for unstructured/large-scale data (social media, IoT, big data).
- **NewSQL** → Best of both worlds: SQL power + NoSQL scalability.

Aspect	RDBMS (e.g., MySQL, PostgreSQL)	NoSQL (e.g., MongoDB, Cassandra, Redis)	NewSQL (e.g., Google Spanner, CockroachDB)
Data Model	Relational (tables, rows, columns)	Flexible (Key-Value, Document, Column, Graph)	Relational (tables) but designed for scalability
Schema	Fixed schema (strict)	Schema-less / dynamic	Relational schema with flexibility
Query Language	SQL (Structured Query Language)	Varies (Mongo Query Language, CQL, etc.)	SQL (standard)
Scalability	Vertical scaling (add more power to single server)	Horizontal scaling (distribute across nodes)	Horizontal scaling with ACID guarantees

Aspect	RDBMS (e.g., MySQL, PostgreSQL)	NoSQL (e.g., MongoDB, Cassandra, Redis)	NewSQL (e.g., Google Spanner, CockroachDB)
Transactions	Strong **ACID compliance**	BASE (Basically Available, Soft state, Eventual consistency)	Full **ACID** support at scale
Consistency Model	Strong consistency	Eventual consistency (some provide tunable consistency)	Strong consistency (global transactions)
Performance	High for structured data	High for unstructured & large-scale data	High (balances scalability + transactions)
Use Cases	Banking, ERP, HR systems, Inventory	Big Data, IoT, Social Media, Real-time apps	Hybrid workloads: financial + real-time analytics
Examples	MySQL, PostgreSQL, Oracle, SQL Server	MongoDB, Cassandra, Redis, HBase	Google Spanner, CockroachDB, VoltDB
Strengths	Data integrity, complex queries, reliability	Flexibility, scalability, high speed for large datasets	Combines SQL + scalability + ACID
Limitations	Not scalable for massive distributed systems	Limited transactions, weaker consistency	Still evolving, less mature, costlier