UNIT I — Introduction to DBMS

Topics: Purpose of DBMS, View of Data, DB Languages, DB System Structure, ER Model, ER Diagram, Extended ER Features, Converting ER to Tables

Q1. Explain the levels of abstraction in a database system with example.

A database system hides the complexity of how data is physically stored from the users. To do this, it uses three levels of abstraction — each level hides the details of the level below it. These three levels together form the ANSI/SPARC architecture.

1. Physical Level (Lowest Level)

- This is the lowest level of abstraction.
- It describes HOW the data is actually stored on disk — file formats, storage structures, access paths (like B-trees, hash tables), and block sizes.
- Database administrators and system software work at this level.
- Example: Student records stored as binary files with B-tree index on RollNo.

2. Logical Level (Conceptual Level)

- This is the middle level of abstraction.
- It describes WHAT data is stored and the relationships between the data, without worrying about how it is physically stored.
- Database developers and DBAs design the logical schema at this level.
- Example: Student(rollno INT, name VARCHAR(50), marks INT) — a table definition.

3. View Level (Highest Level)

- This is the highest level, closest to the end user.
- It shows only a part of the entire database to specific users, hiding the rest for security and simplicity.
- Different users can have different views of the same database.
- Example: A student can only see their own marks. A faculty member can see all student marks but not salary data.

Importance: These three levels provide data abstraction, data independence, and security. Users at higher levels do not need to know lower-level implementation details.

Q2. Draw and explain the Database System Structure with its components.

The database system structure consists of two main functional components: the Storage Manager and the Query Processor. They work together on top of disk storage.

A. Storage Manager

The Storage Manager is responsible for storing, retrieving, and updating data on disk. It acts as a bridge between the physical data on disk and the queries processed by the system. Its sub-components are:

- — Authorization and Integrity Manager

Checks whether a user has permission to perform the requested operation and whether integrity constraints are satisfied.

- — Transaction Manager

Ensures that the database remains in a consistent state even in case of failures. It handles concurrency and recovery.

- — File Manager

Manages disk space allocation and the data structures (files, pages, blocks) used to store data on disk.

- — Buffer Manager

Manages the transfer of data between disk and main memory (RAM). It keeps frequently accessed data in a buffer pool in RAM for fast access.

B. Query Processor

The Query Processor handles user queries and converts them into efficient low-level instructions to fetch or modify data. Its sub-components are:

- — DDL Interpreter

Interprets DDL statements like CREATE TABLE, DROP TABLE, and updates the data dictionary (system catalog) accordingly.

- — DML Compiler

Converts DML queries (SELECT, INSERT, UPDATE, DELETE) written in SQL into an efficient query evaluation plan (execution plan). It performs query optimization.

- — Query Evaluation Engine

Executes the low-level instructions generated by the DML compiler and returns results to the user.

C. Disk Storage

The disk storage contains: Data files (actual data), Data dictionary (metadata about tables and columns), Indices (for faster search), Statistical data (used for query optimization), and Log files (for recovery).

Q3. Explain Storage Manager and Query Processor in DBMS architecture.

(This question overlaps with Q2. Refer to Q2 for complete Storage Manager and Query Processor explanation.)

Additional Points on Storage Manager:

- It provides an interface for the file system to the DML compiler.
- It is responsible for the physical organization of data.
- The Buffer Manager is the most critical component — it minimizes disk I/O by caching data in RAM.
- The Transaction Manager implements the ACID properties during concurrent execution.

Additional Points on Query Processor:

- The DML compiler performs semantic checking (e.g., verifying column names exist).
- Query optimization chooses the most efficient execution plan among many possible plans.
- The evaluation engine uses algorithms like nested-loop join, hash join, and merge join.

Q4. What is Specialization and Generalization in Extended ER Diagram? What is the difference between them?

Generalization

- Generalization is a BOTTOM-UP approach in ER modeling.
- Multiple lower-level entities with common attributes are combined into a single higher-level entity.
- It abstracts common features from several entity sets into a superclass.
- Example: 'Car' and 'Bus' both have attributes like vehicle_no, colour, speed. These are generalized into a higher-level entity 'Vehicle'.
- The process: identify commonalities → merge into superclass → keep unique attributes in subclasses.

Specialization

- Specialization is a TOP-DOWN approach in ER modeling.
- A higher-level entity is divided into several lower-level entities based on distinguishing characteristics.
- It allows different subclasses to have their own special attributes.
- Example: 'Employee' can be specialized into 'Manager' (with bonus attribute) and 'Technician' (with skills attribute).
- Specialization is represented using ISA (is-a) relationships and a triangle/arc in the ER diagram.

Difference between Specialization and Generalization

Feature	Generalization	Specialization
Approach	Bottom-Up	Top-Down
Direction	Subclass → Superclass	Superclass → Subclass
Purpose	Combine common attributes	Highlight unique attributes
Example	Car, Bus → Vehicle	Employee → Manager, Technician

Why no difference in schema diagram: Both generalization and specialization result in exactly the same database schema — a superclass table and subclass tables linked by a primary key. The difference is only in the design thought process, not in the final structure. Hence, schema diagrams do not distinguish between them.

Q5. Explain Physical Data Independence with example. Also explain its importance.

Data independence refers to the immunity of application programs to changes made in the definition of data. There are two types:

- Physical Data Independence — immunity to changes in physical storage
- Logical Data Independence — immunity to changes in logical schema

Physical Data Independence — Definition

- Physical data independence means we can change the physical storage structure or access method of data WITHOUT affecting the logical schema or application programs.
- The logical level (what data is stored) remains unchanged even if the physical level (how it is stored) changes.
- It is the easier of the two types of independence to achieve.

Example

- Suppose student data is currently stored as a sequential flat file.
- The DBA decides to change storage to a B-tree indexed file for faster search.
- After this change, the SQL queries, table definitions, and application programs remain exactly the same.
- Users and developers do not even notice the change.

Importance of Physical Data Independence

- Allows DBAs to tune and optimize storage without breaking applications.
- Enables technology upgrades (e.g., moving from HDD to SSD storage) transparently.
- Reduces maintenance cost — developers don't need to rewrite code when storage changes.
- Enables performance improvements through better indexing or partitioning without application-level changes.
- Supports migration between different storage engines or database versions.

Q6. Explain Candidate Key, Primary Key, and Foreign Key with example.

Candidate Key

- A candidate key is any attribute (or set of attributes) that can UNIQUELY identify each row in a table.
- There can be multiple candidate keys in a single table.
- A candidate key must satisfy: Uniqueness (no two rows have the same value) and Minimality (no subset of the key is sufficient to uniquely identify rows).
- Example in Person(driver_id, name, address, contactno): Both driver_id and contactno can uniquely identify a person — both are candidate keys.

Primary Key

- The primary key is the candidate key chosen by the database designer to uniquely identify rows in a table.
- There can be only ONE primary key per table.
- A primary key value must be: Unique (no duplicates), Not NULL (must always have a value).
- Example: In Person table, driver_id is chosen as the primary key.
- In Car(licence, model, year), licence is the primary key.

Foreign Key

- A foreign key is an attribute in one table that references the primary key of another table.

- It is used to establish a relationship between two tables.
- It enforces REFERENTIAL INTEGRITY — a foreign key value must exist in the referenced table (or be NULL).
- Example: In Owns(driver_id, licence):
- driver_id is a foreign key referencing Person(driver_id)
- licence is a foreign key referencing Car(licence)

Summary Table for given schema

Table	Primary Key	Foreign Key	Candidate Keys
Person(driver_id, name, address, contactno)	driver_id	None	driver_id, contactno
Car(licence, model, year)	licence	None	licence
Owns(driver_id, licence)	driver_id + licence (composite)	driver_id → Person, licence → Car	driver_id + licence

---

Q7. Characteristics of Database Approach vs. Traditional File System.

Main Characteristics of Database Approach

- — Self-Describing Nature

A DBMS contains a complete description of the database structure and constraints (called metadata) in the system catalog (data dictionary). The data and its description are stored together.

- — Insulation between Programs and Data (Data Abstraction)

In DBMS, the structure of data files is stored in the catalog separately from the access programs. A change in structure does not require changing access programs.

- — Multiple Views of the Data

DBMS allows different users to have different views of the database, showing only relevant portions to each user.

- — Sharing of Data and Multiuser Processing

DBMS supports concurrent access by multiple users while maintaining data consistency using concurrency control mechanisms.

- — Controlled Data Redundancy

DBMS stores data in one place and shares it among all applications, reducing redundancy. Any necessary redundancy is controlled and managed.

- — Enforcement of Integrity Constraints

Constraints like primary key, foreign key, check, and domain constraints are automatically enforced by the DBMS.

- — Backup and Recovery

DBMS provides automatic backup and recovery mechanisms to restore data after failures.

How it differs from Traditional File System

Feature	Database Approach (DBMS)	Traditional File System
Data Redundancy	Minimal — data stored once centrally	High — same data in multiple files

Data Inconsistency	Avoided through central storage	Common — different files have different values
Data Sharing	Easy — multiple users, controlled access	Difficult — programs own their data
Data Independence	High — schema change doesn't affect programs	Low — programs depend on file structure
Integrity Constraints	Automatically enforced by DBMS	Must be coded in every application
Security	Fine-grained access control built-in	No built-in security
Concurrent Access	Supported with locking/transactions	Not supported
Backup & Recovery	Built-in by DBMS	Manual, application responsibility

---

Q8. Compare DBMS and File Processing System (Data Isolation, Redundancy, Inconsistency, Integrity).

1. Data Redundancy

- File System: The same data (e.g., a student's address) may be stored in multiple files maintained by different departments. This wastes storage.
- DBMS: Data is stored centrally in a single location. All applications share the same data store, eliminating unnecessary duplication.

2. Data Inconsistency

- File System: When data is duplicated across files, updating one file but not others leads to inconsistency. Example: A student's address updated in one department's file but not another's.
- DBMS: Since data is stored once, any update automatically reflects everywhere. Consistency is maintained automatically.

3. Data Isolation

- File System: Data is scattered across different files in different formats (some CSV, some binary, some text). Accessing combined data from multiple files requires custom programming.
- DBMS: All data is accessible through a unified query language (SQL). Joins and relationships allow easy combined access without custom programs.

4. Data Integrity

- File System: Integrity constraints (like age must be > 0, or student must belong to a valid department) must be manually programmed in each application. If one application skips the check, invalid data enters.
- DBMS: Constraints are defined once in the database schema (PRIMARY KEY, FOREIGN KEY, CHECK, NOT NULL) and automatically enforced by the DBMS for all applications.

Q9. Redundancy when Weak Entity is converted to Strong Entity.

A weak entity is one that does not have a primary key of its own and depends on a strong (owner) entity for its identification. It uses a discriminator (partial key) along with the primary key of its owner entity.

Example

- Strong Entity: Loan(loan_no, amount, branch_name) — loan_no is primary key
- Weak Entity: Loan_Payment(payment_no, payment_date, payment_amount) — identified by payment_no + loan_no

What happens when we make it Strong?

- We add loan_no as a regular attribute to Loan_Payment, making it: Loan_Payment(loan_no, payment_no, payment_date, payment_amount)
- Now Loan_Payment has its own composite primary key: {loan_no, payment_no}

Redundancy that Results

- loan_no now appears in BOTH the Loan table and the Loan_Payment table.
- If we need to update loan information, we must update it in two places — Loan table AND every row in Loan_Payment table that has that loan_no.
- Failing to update in all places causes UPDATE ANOMALY.
- If a loan is deleted from Loan table, all related payments in Loan_Payment become orphaned — DELETE ANOMALY.
- We cannot add a loan with no payments yet without adding a dummy payment — INSERT ANOMALY.

Conclusion: Converting a weak entity to strong by duplicating the owner's primary key introduces unnecessary redundancy and all types of anomalies — exactly what good database design tries to avoid.

Q10. ER Diagram for Banking System.

Entities and Attributes

- Branch: branch_name (PK), city, assets
- Account: account_no (PK), balance, type (savings/current)
- Customer: customer_id (PK), name, address, contact_no
- Loan: loan_no (PK), amount, loan_type

Relationships and Cardinalities

- Customer HOLDS Account → Many-to-Many (a customer can have multiple accounts; an account can be held by multiple customers)
- Customer BORROWS Loan → Many-to-Many (a customer can have multiple loans; a loan can be taken by multiple customers)
- Branch OFFERS Account → One-to-Many (one branch offers many accounts)
- Branch GRANTS Loan → One-to-Many (one branch grants many loans)

Keys Identified

- Branch: branch_name | Account: account_no | Customer: customer_id | Loan: loan_no

Tables after Conversion from ER

```
Branch (branch_name, city, assets)
Customer (customer_id, name, address, contact_no)
Account (account_no, branch_name, balance, type)
   -- branch_name is FK referencing Branch
Loan (loan_no, branch_name, amount, loan_type)
   -- branch_name is FK referencing Branch
Holds (customer_id, account_no)
   -- customer_id FK → Customer, account_no FK → Account
Borrows (customer_id, loan_no)
   -- customer_id FK → Customer, loan_no FK → Loan
```

Q11. ER Diagram for Company Database.

Entities and Attributes

- Department: dept_name (PK), dept_no, locations (multi-valued)
- Project: proj_name (PK), proj_no (unique), location
- Employee: ssno (PK), name, address, salary, sex, birthdate
- Dependent: name (partial key), birthdate, relationship — Weak Entity of Employee

Relationships and Cardinalities

- Employee MANAGES Department → One-to-One (one employee manages one department, with start_date attribute on relationship)
- Department CONTROLS Project → One-to-Many (one department controls many projects)
- Employee WORKS_FOR Department → Many-to-One (one employee works for one department; one department has many employees)
- Employee WORKS_ON Project → Many-to-Many (an employee works on many projects; a project has many employees — hours_per_week is attribute of this relationship)
- Employee SUPERVISES Employee → Recursive/Unary (one employee supervises many employees)
- Employee HAS Dependent → One-to-Many (weak entity relationship)

Tables after Conversion

```
Department (dept_no, dept_name)
Dept_Locations (dept_no, location)          -- multi-valued attribute
Project (proj_no, proj_name, location, dept_no)   -- dept_no FK →
Department
Employee (ssno, name, address, salary, sex, birthdate, dept_no,
supervisor_ssno)
   -- dept_no FK → Department, supervisor_ssno FK → Employee
Works_On (ssno, proj_no, hours_per_week)
```

```
   -- ssno FK → Employee, proj_no FK → Project
Manages (ssno, dept_no, start_date)
   -- ssno FK → Employee, dept_no FK → Department
Dependent (ssno, dep_name, birthdate, relationship)
   -- ssno FK → Employee (weak entity)
```

Q12. ER Diagram for Post Office System.

Entities and Attributes

- Letter: letter_id (PK), date_of_registration, date_of_arrival, status
- Sender: sender_id (PK), name, address
- Receiver: receiver_id (PK), name, address
- PostOffice: po_id (PK), city, pincode
- Postman: postman_id (PK), name
- Area: area_id (PK), area_name
- Street: street_id (PK), street_name
- Building: building_no (PK), building_name
- Family: family_id (PK), family_name

Relationships and Cardinalities

- Sender SENDS Letter → One-to-Many (one sender sends many letters)
- Letter DESTINED_TO Receiver → Many-to-One (many letters go to one receiver)
- Letter ORIGINATES_FROM PostOffice → Many-to-One
- Letter DESTINED_TO PostOffice → Many-to-One
- Postman COVERS Area → Many-to-One
- Area CONTAINS Street → One-to-Many
- Street HAS Building → One-to-Many
- Building HOUSES Family → One-to-Many

Status values (stored as attribute of Letter)

- Not yet taken for delivery | Delivered | Address not available
- Address not known | Addressee refused | Redirected | Sent back to sender

Key Tables

```
Letter (letter_id, sender_id, receiver_id, origin_po_id, dest_po_id,
reg_date, arrival_date, status)
Postman (postman_id, name, area_id)
Building (building_no, building_name, street_id)
```

Q13. ER Diagram for Hospital System.

Entities and Attributes

- Doctor: doctor_id (PK), name, specialization, address, phone

- Patient: patient_id (PK), name, address, age, date_admitted
- Room: room_no (PK), type (general/ICU/private), charges_per_day
- Lab: lab_id (PK), lab_name, location
- Test: test_id (PK), test_name

Relationships and Cardinalities

- Patient ADMITTED_TO Room → Many-to-One (one room holds many patients at different times)
- Doctor TREATS Patient → Many-to-Many (one doctor treats many patients; one patient is treated by many doctors)
- Patient UNDERGOES Test → Many-to-Many (with result, test_date attributes)
- Lab CONDUCTS Test → One-to-Many (one lab conducts many tests)

Tables after Conversion

```
Doctor (doctor_id, name, specialization, address, phone)
Patient (patient_id, name, address, age, date_admitted, room_no)
Room (room_no, type, charges_per_day)
Lab (lab_id, lab_name, location)
Test (test_id, test_name, lab_id)           -- lab_id FK → Lab
Treats (doctor_id, patient_id, treatment_date, notes)
   -- doctor_id FK → Doctor, patient_id FK → Patient
Undergoes (patient_id, test_id, test_date, result)
   -- patient_id FK → Patient, test_id FK → Test
```

UNIT II — SQL and PL/SQL

Topics: DDL, DML, Views, Indexes, Joins, Nested Queries, DCL, TCL, Cursors, Stored Procedures, Functions, Triggers

Q14. SQL Queries on Emp, Dept, Address Schema.

Schema: Emp(Emp_no, Emp_name, Dept_no) | Dept(Dept_no, Dept_name) | Address(Dept_name, Dept_location)

i) Display location of department where employee 'Ram' works

```
SELECT A.Dept_location
FROM Address A, Dept D, Emp E
WHERE E.Emp_name = 'Ram'
  AND E.Dept_no = D.Dept_no
  AND D.Dept_name = A.Dept_name;
```

ii) Create View for total employees per department in ascending order

```
CREATE VIEW Dept_Employee_Count AS
  SELECT D.Dept_name, COUNT(E.Emp_no) AS Total_Employees
  FROM Dept D
  LEFT JOIN Emp E ON D.Dept_no = E.Dept_no
  GROUP BY D.Dept_name
  ORDER BY Total_Employees ASC;
```

iii) Find department where no employee is working

```
SELECT Dept_name
FROM Dept
WHERE Dept_no NOT IN (
  SELECT DISTINCT Dept_no FROM Emp
);
```

Q15. SQL Queries on Supplier, Parts, Shipments Schema.

Schema: Supplier(SNO, Sname, Status, City) | Parts(PNO, Pname, Color, Weight, City) | Shipments(SNO, PNO, QTY)

i) Shipment info where QTY < 157

```
SELECT S.SNO, S.Sname, P.PNO, P.Pname, SH.QTY
FROM Supplier S
JOIN Shipments SH ON S.SNO = SH.SNO
JOIN Parts P ON P.PNO = SH.PNO
WHERE SH.QTY < 157;
```

ii) Suppliers whose shipment QTY > average QTY

```
SELECT S.SNO, S.Sname, P.PNO, P.Pname
```

```
FROM Supplier S
JOIN Shipments SH ON S.SNO = SH.SNO
JOIN Parts P ON P.PNO = SH.PNO
WHERE SH.QTY > (SELECT AVG(QTY) FROM Shipments);
```

iii) Aggregate QTY of green part PNO=1692 shipped from Mumbai

```
SELECT SUM(SH.QTY) AS Total_Quantity
FROM Shipments SH
JOIN Parts P ON SH.PNO = P.PNO
JOIN Supplier S ON SH.SNO = S.SNO
WHERE P.PNO = 1692
  AND P.Color = 'Green'
  AND S.City = 'Mumbai';
```

Q16. SQL Queries on Movies, Director, Rating Schema.

Schema: MOVIES(Mov_Id, Mov_Title, Mov_Year, Dir_Id) | DIRECTOR(Dir_Id, Dir_Name) | RATING(Mov_Id, Rev_Stars)

i) Movies directed by 'RAJ KAPOOR'

```
SELECT M.Mov_Title
FROM MOVIES M
JOIN DIRECTOR D ON M.Dir_Id = D.Dir_Id
WHERE D.Dir_Name = 'RAJ KAPOOR';
```

ii) Movie names and stars, sorted by title ASC and stars DESC

```
SELECT M.Mov_Title, R.Rev_Stars
FROM MOVIES M
JOIN RATING R ON M.Mov_Id = R.Mov_Id
ORDER BY M.Mov_Title ASC, R.Rev_Stars DESC;
```

iii) Update rating of Steven Spielberg movies to 9

```
UPDATE RATING
SET Rev_Stars = 9
WHERE Mov_Id IN (
  SELECT M.Mov_Id
  FROM MOVIES M
  JOIN DIRECTOR D ON M.Dir_Id = D.Dir_Id
  WHERE D.Dir_Name = 'Steven Spielberg'
);
```

Q17. What is a View? Can you update a View?

What is a View?

- A VIEW is a virtual table that is created from a SELECT query on one or more base tables.
- It does NOT store data physically — it stores only the definition (the SELECT query).
- When the view is accessed, the underlying query is executed dynamically.
- Views provide data security, simplify complex queries, and give customized access to users.

Creating a View

```
CREATE VIEW High_Salary_Emp AS
  SELECT emp_id, emp_name, salary
  FROM Employee
  WHERE salary > 50000;
```

To query the view:

```
SELECT * FROM High_Salary_Emp;
```

Can you update a View?

- YES — views can be updated, but only under specific conditions.

Conditions when a view CAN be updated

- The view is based on a SINGLE base table.
- The view does not use: GROUP BY, DISTINCT, aggregate functions (SUM, COUNT, etc.), set operations (UNION, INTERSECT).
- All NOT NULL columns of the base table are included in the view.
- The view does not include computed columns (e.g., salary * 1.1).

Conditions when a view CANNOT be updated

- View is based on multiple joined tables — database cannot determine which base table row to modify.
- View uses GROUP BY or aggregate functions — no direct row-to-row mapping.
- View uses DISTINCT — multiple base rows map to one view row.

Q18. What is an Index? Advantages and Disadvantages.

Definition of Index

- An INDEX is a data structure (like B-tree or hash) that is associated with a table to speed up data retrieval operations.
- It works like an index in a book — instead of scanning every page, you go directly to the right page.
- Indexes are created on one or more columns of a table.

Creating an Index

```
CREATE INDEX idx_emp_name ON Employee (emp_name);
CREATE UNIQUE INDEX idx_emp_id ON Employee (emp_id);
```

Types of Indexes

- Primary Index — built on the primary key of an ordered file.
- Secondary Index — built on a non-primary key field.
- Clustered Index — data rows are physically sorted in the same order as the index.
- Non-Clustered Index — index is separate from data; pointers point to actual data rows.
- Unique Index — ensures all values in the indexed column are unique.

Advantages of Indexes

- Dramatically speeds up SELECT queries, especially on large tables with millions of rows.
- Speeds up JOIN operations when joining on indexed columns.
- Enforces uniqueness (UNIQUE index).
- Improves ORDER BY performance as data is already pre-sorted.
- Reduces the need for full table scans.

Disadvantages of Indexes

- Extra storage space is required to maintain the index data structure.
- INSERT, UPDATE, DELETE operations become slower because the index must also be updated.
- Too many indexes on a table can slow down write-heavy applications significantly.
- Index maintenance requires regular rebuilding to avoid fragmentation.
- Query optimizer may choose an index incorrectly, sometimes making queries slower.

Q19. Explain Referential Integrity and Entity Integrity Constraints.

Entity Integrity

- Entity integrity states that the PRIMARY KEY of a relation must NEVER contain a NULL value.
- Every tuple (row) must be uniquely identifiable.
- If the primary key is NULL, we cannot distinguish one row from another, violating the purpose of a primary key.
- Example: In Student(rollno, name, marks), rollno is the primary key. No student can have a NULL rollno.

```
CREATE TABLE Student (
  rollno INT PRIMARY KEY,   -- rollno cannot be NULL
  name VARCHAR(50) NOT NULL,
  marks INT
);
```

Referential Integrity

- Referential integrity states that a FOREIGN KEY value in a child table must either match an existing PRIMARY KEY value in the parent table, OR be NULL.
- It prevents orphan records — you cannot have a child record without a corresponding parent record.
- Example: In Enrollment(rollno, course_id), rollno references Student(rollno). You cannot enroll a student who doesn't exist.

```
CREATE TABLE Enrollment (
```

```
  rollno INT,
  course_id INT,
  FOREIGN KEY (rollno) REFERENCES Student(rollno)
    ON DELETE CASCADE
    ON UPDATE CASCADE
);
```

Referential Integrity Actions

- ON DELETE CASCADE — if parent row is deleted, child rows are also deleted automatically.
- ON DELETE SET NULL — if parent is deleted, foreign key in child becomes NULL.
- ON DELETE RESTRICT — prevents deletion of parent if child records exist.

Q20. Importance of Constraints. Explain 4 Constraints with examples.

Constraints are rules defined on database tables to maintain data accuracy, consistency, and reliability. Without constraints, invalid, duplicate, or orphan data can enter the database, causing serious problems.

1. NOT NULL Constraint

- Ensures that a column cannot have a NULL (empty) value.
- Used for mandatory fields that must always have a value.

```
CREATE TABLE Student (
  rollno INT,
  name VARCHAR(50) NOT NULL,    -- name is mandatory
  email VARCHAR(100) NOT NULL
);
```

2. UNIQUE Constraint

- Ensures all values in a column (or combination of columns) are distinct — no two rows can have the same value.
- A table can have multiple UNIQUE constraints.

```
CREATE TABLE Employee (
  emp_id INT PRIMARY KEY,
  email VARCHAR(100) UNIQUE,    -- no two employees share an email
  pan_number CHAR(10) UNIQUE
);
```

3. PRIMARY KEY Constraint

- Combines NOT NULL and UNIQUE — uniquely identifies each row.
- Only ONE primary key per table; can be composite (multiple columns).

```
CREATE TABLE Department (
  dept_no INT PRIMARY KEY,      -- unique, not null
  dept_name VARCHAR(50)
);
```

4. CHECK Constraint

- Ensures that values in a column satisfy a specific condition.
- Acts as a filter — rejects any value that does not pass the condition.

```
CREATE TABLE Employee (
  emp_id INT PRIMARY KEY,
  age INT CHECK (age >= 18 AND age <= 65),
  salary DECIMAL CHECK (salary > 0)
);
```

5. FOREIGN KEY Constraint (Bonus)

- Links two tables by referencing the primary key of another table.
- Enforces referential integrity.

```
CREATE TABLE Orders (
  order_id INT PRIMARY KEY,
  customer_id INT,
  FOREIGN KEY (customer_id) REFERENCES Customer(customer_id)
);
```

Q21. Explain Cursor in SQL with example.

What is a Cursor?

- A cursor is a database object used to retrieve and process query results ONE ROW AT A TIME.
- Normally, SQL processes all rows at once (set-based). Cursors allow row-by-row processing.
- Cursors are mainly used in PL/SQL when you need to perform calculations or operations on each row individually.

Types of Cursors

- Implicit Cursor — automatically created by Oracle for single-row SELECT, INSERT, UPDATE, DELETE. No explicit control needed.
- Explicit Cursor — manually declared and controlled by the programmer for multi-row queries.

Steps to use an Explicit Cursor

1. DECLARE — define the cursor with a SELECT query
2. OPEN — execute the SELECT query and populate the cursor result set
3. FETCH — retrieve rows one at a time into variables
4. CLOSE — release the cursor and free memory

Example — Print all student names from Student table

```
DECLARE
  CURSOR student_cursor IS
    SELECT rollno, name FROM Student;
```

```
  v_rollno Student.rollno%TYPE;
  v_name   Student.name%TYPE;
BEGIN
  OPEN student_cursor;
  LOOP
    FETCH student_cursor INTO v_rollno, v_name;
    EXIT WHEN student_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Roll: ' || v_rollno || ' Name: ' || v_name);
  END LOOP;
  CLOSE student_cursor;
END;
```

Cursor Attributes

- %FOUND — TRUE if last FETCH retrieved a row
- %NOTFOUND — TRUE if last FETCH returned no row (used to exit loop)
- %ROWCOUNT — number of rows fetched so far
- %ISOPEN — TRUE if cursor is open

Q22. Define Stored Procedure. Explain creation and calling with example.

What is a Stored Procedure?

- A stored procedure is a named, precompiled collection of SQL and PL/SQL statements stored in the database.
- It is compiled once and can be executed multiple times, improving performance.
- It reduces network traffic — instead of sending multiple SQL statements, only the procedure call is sent.
- It supports modular programming — complex logic is encapsulated in reusable procedures.

Advantages of Stored Procedures

- Performance: Compiled once, executed many times — faster than ad-hoc queries.
- Security: Users can execute the procedure without direct access to underlying tables.
- Reusability: Same procedure called from multiple applications.
- Maintainability: Business logic in one place, not scattered across applications.
- Reduced Network Traffic: Only the CALL command is sent over the network.

Syntax to Create a Stored Procedure

```
CREATE OR REPLACE PROCEDURE procedure_name
  (param1 IN datatype, param2 OUT datatype)
IS
  -- local variable declarations
BEGIN
  -- SQL and PL/SQL statements
END procedure_name;
```

Example — Procedure to get salary of an employee

```
CREATE OR REPLACE PROCEDURE Get_Employee_Salary
  (p_emp_id IN Employee.emp_id%TYPE,
   p_salary OUT Employee.salary%TYPE)
IS
BEGIN
  SELECT salary INTO p_salary
  FROM Employee
  WHERE emp_id = p_emp_id;
  DBMS_OUTPUT.PUT_LINE('Salary: ' || p_salary);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Employee not found.');
END Get_Employee_Salary;
```

Calling the Stored Procedure

```
DECLARE
  v_sal Employee.salary%TYPE;
BEGIN
  Get_Employee_Salary(101, v_sal);
END;
```

Q23. What is a Trigger? How to create it? Types of Triggers.

What is a Trigger?

- A trigger is a special type of stored procedure that automatically executes (fires) in response to specific events (INSERT, UPDATE, DELETE) on a table or view.
- Triggers do NOT need to be called explicitly — they fire automatically.
- They are used to: enforce business rules, maintain audit logs, validate data, automatically update related tables.

Syntax to Create a Trigger

```
CREATE OR REPLACE TRIGGER trigger_name
  {BEFORE | AFTER | INSTEAD OF}
  {INSERT | UPDATE | DELETE} [OF column_name]
  ON table_name
  [FOR EACH ROW]
BEGIN
  -- trigger body
END trigger_name;
```

Types of Triggers

- — BEFORE Trigger

Fires BEFORE the triggering DML operation. Used to validate or modify data before it is saved. Example: Check if salary is valid before INSERT.

- — AFTER Trigger

Fires AFTER the triggering DML operation is completed. Used to log changes, update other tables, or send notifications. Example: Log every salary update.

- — INSTEAD OF Trigger

Fires IN PLACE OF the DML operation. Used mainly on VIEWS that cannot be updated directly. The trigger defines what should happen instead.

- — Row-Level Trigger (FOR EACH ROW)

Fires once for EACH ROW affected by the DML statement. Uses :NEW and :OLD to access new and old values.

- — Statement-Level Trigger

Fires ONCE for the entire DML statement, regardless of how many rows are affected.

Example — Log salary changes

```
CREATE OR REPLACE TRIGGER Salary_Audit
AFTER UPDATE OF salary ON Employee
FOR EACH ROW
BEGIN
  INSERT INTO Audit_Log(emp_id, old_salary, new_salary, changed_date)
  VALUES (:OLD.emp_id, :OLD.salary, :NEW.salary, SYSDATE);
END;
```

Q24. Trigger to preserve old student fee values before update.

Schema: Student_fee_details(rollno, name, fee_deposited, date)
We need to save old values to a backup table before any update occurs.

Step 1: Create Backup Table

```
CREATE TABLE Student_fee_backup (
  backup_id   NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  rollno      NUMBER,
  name        VARCHAR2(50),
  fee_deposited NUMBER,
  fee_date    DATE,
  backup_time DATE DEFAULT SYSDATE
);
```

Step 2: Create the BEFORE UPDATE Trigger

```
CREATE OR REPLACE TRIGGER Preserve_Old_Fee
BEFORE UPDATE ON Student_fee_details
FOR EACH ROW
BEGIN
  -- Save old values to backup table before they are overwritten
  INSERT INTO Student_fee_backup
    (rollno, name, fee_deposited, fee_date, backup_time)
  VALUES
```

```
    (:OLD.rollno, :OLD.name, :OLD.fee_deposited, :OLD.date, SYSDATE);
END Preserve_Old_Fee;
```

Explanation

- :OLD.rollno — refers to the original rollno before update
- :OLD.fee_deposited — the fee amount before the update
- BEFORE UPDATE — fires before the change is applied, so old values are still available
- FOR EACH ROW — fires once per row being updated

Q25. PL/SQL block to check student attendance.

Schema: student_attendance(RollNo, Attendance)

```
DECLARE
  v_rollno  student_attendance.RollNo%TYPE;
  v_att     student_attendance.Attendance%TYPE;
BEGIN
  -- Accept RollNo from user
  v_rollno := &Enter_RollNo;

  -- Fetch attendance for that roll number
  SELECT Attendance
  INTO v_att
  FROM student_attendance
  WHERE RollNo = v_rollno;

  -- Display the result
  DBMS_OUTPUT.PUT_LINE('Roll No  : ' || v_rollno);
  DBMS_OUTPUT.PUT_LINE('Attendance: ' || v_att || '%');

  IF v_att >= 75 THEN
    DBMS_OUTPUT.PUT_LINE('Status: Eligible to appear in exam.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Status: Detained. Attendance below 75%.');
  END IF;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Error: Roll No not found in records.');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Unexpected error: ' || SQLERRM);
END;
```

Q26. PL/SQL Procedure to list rooms at hotel 'TAJ'.

Schema: Hotels(hotel_no, hotel_name, city) | Rooms(Room_no, hotel_no, price, type)

```
CREATE OR REPLACE PROCEDURE List_TAJ_Rooms IS
  -- Cursor to fetch rooms of TAJ hotel
  CURSOR room_cur IS
    SELECT R.Room_no, R.type, R.price
    FROM Rooms R
    JOIN Hotels H ON R.hotel_no = H.hotel_no
    WHERE H.hotel_name = 'TAJ';

  v_room   Rooms.Room_no%TYPE;
  v_type   Rooms.type%TYPE;
  v_price  Rooms.price%TYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('=== Rooms at Hotel TAJ ===');
  OPEN room_cur;
  LOOP
    FETCH room_cur INTO v_room, v_type, v_price;
    EXIT WHEN room_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(
      'Room: ' || v_room ||
      ' | Type: ' || v_type ||
      ' | Price: Rs.' || v_price
    );
  END LOOP;
  CLOSE room_cur;
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END List_TAJ_Rooms;
```

Calling the procedure:

```
EXEC List_TAJ_Rooms;
```

Q27. What is a Sequence in SQL?

Definition

- A SEQUENCE is a database object that automatically generates unique sequential numeric values.
- It is mainly used to auto-generate PRIMARY KEY values without manual tracking.
- Sequences are independent of tables — one sequence can be used by multiple tables.

Creating a Sequence

```
CREATE SEQUENCE student_seq
```

```
  START WITH 1          -- starting value
  INCREMENT BY 1        -- step size
  MINVALUE 1            -- minimum value
  MAXVALUE 99999        -- maximum value
  NOCYCLE               -- does not restart after reaching MAXVALUE
  NOCACHE;              -- does not pre-generate values in memory
```

Using the Sequence

```
INSERT INTO Student (rollno, name, marks)
VALUES (student_seq.NEXTVAL, 'Alice', 85);

-- NEXTVAL: gets and increments the sequence (next value)
-- CURRVAL: gets the current value (without incrementing)
SELECT student_seq.CURRVAL FROM DUAL;
```

Advantages of Sequences

- Guarantees unique values even in concurrent multi-user environments.
- No manual tracking needed — the database manages the counter.
- Performance: can cache values (CACHE option) for faster generation.

Q28. What is a Synonym in SQL?

Definition

- A SYNONYM is an alternative name (alias) for a database object such as a table, view, sequence, procedure, or function.
- It is used to simplify long object names, provide location transparency, and hide the actual schema structure.

Types of Synonyms

- Private Synonym — created by a user and accessible only to that user.
- Public Synonym — created by a DBA and accessible to all database users.

Creating a Synonym

```
-- Private Synonym
CREATE SYNONYM emp FOR HR.Employee_Details;

-- Public Synonym
CREATE PUBLIC SYNONYM emp FOR HR.Employee_Details;
```

Using a Synonym

```
-- Instead of writing the full qualified name
SELECT * FROM HR.Employee_Details;
```

```
-- You can simply write
SELECT * FROM emp;
```

Dropping a Synonym

```
DROP SYNONYM emp;
DROP PUBLIC SYNONYM emp;
```

Advantages

- Simplifies long or complex object names, especially for remote database objects.
- Provides abstraction — the actual table name or owner can change without affecting queries.
- Improves portability — applications don't need to know the exact schema/owner name.
- Security — hides the real table name and structure from end users.

UNIT III — Relational Database Design

Topics: Codd's Rules, Integrity Constraints, Features of Good Design, Normalization (1NF, 2NF, 3NF, BCNF), Functional Dependencies, Decomposition

Q29. Codd's 12 Rules — All Rules with Explanation.

Dr. E.F. Codd proposed 12 rules (Rule 0 to Rule 12) to define what a TRUE relational DBMS must satisfy. Rule 0 is a meta-rule that covers all others.

1. Rule 0 — Foundation Rule: The system must manage databases entirely through relational capabilities. Any access must go through the relational model.
2. Rule 1 — Information Rule: ALL information in the database (including metadata) must be represented as values in TABLES only — no hidden data structures.
3. Rule 2 — Guaranteed Access Rule: Every data value must be accessible using exactly three pieces of information: TABLE NAME + PRIMARY KEY VALUE + COLUMN NAME. No other access method should be required.
4. Rule 3 — Systematic NULL Handling: The system must handle NULLs uniformly and consistently for ALL data types. NULL represents missing or unknown data — not zero or empty string.
5. Rule 4 — Active Online Catalog (Data Dictionary): The entire structure of the database (tables, columns, constraints) must be stored in the database itself as tables (system catalog) and queryable via SQL.
6. Rule 5 — Comprehensive Data Sub-Language Rule: The system must support at least one complete language that handles: DDL (CREATE, DROP), DML (SELECT, INSERT, UPDATE, DELETE), Transaction Control (COMMIT, ROLLBACK), and Security (GRANT, REVOKE).
7. Rule 6 — View Updating Rule: All views that are theoretically updatable must be automatically updatable by the system. The DBMS should handle view updates where possible.
8. Rule 7 — High-Level Insert, Update, Delete: The system must support SET-LEVEL operations — INSERT, UPDATE, DELETE must work on multiple rows at once, not just one row at a time.
9. Rule 8 — Physical Data Independence: Application programs and users must be unaffected by changes in the physical storage organization (files, indexes, storage structures).
10. Rule 9 — Logical Data Independence: Application programs must not be affected by changes in the logical schema (adding columns, splitting tables) that preserve information.
11. Rule 10 — Integrity Independence: Integrity constraints (primary key, foreign key, check) must be stored in the data dictionary, NOT in application programs. They must be enforced automatically.
12. Rule 11 — Distribution Independence: The end user must not be aware of whether data is stored on one machine or distributed across multiple machines. SQL works the same way.
13. Rule 12 — Non-Subversion Rule: If the system has a low-level (non-relational) interface, it must NOT be possible to use it to bypass the integrity constraints defined at the relational level.

Q30 & Q31 & Q32. Selected Codd's Rules with Examples.

Guaranteed Access Rule

- Every individual piece of data must be accessible using: Table Name + Primary Key Value + Column Name.
- Example: To access marks of student with rollno=101, use: Student table + rollno=101 + marks column.
- SQL: SELECT marks FROM Student WHERE rollno = 101;

Comprehensive Data Sub-Language Rule

- At least one language must exist that is complete — covering DDL, DML, DCL, and TCL.
- SQL satisfies this rule: DDL (CREATE TABLE, DROP TABLE), DML (SELECT, INSERT, UPDATE, DELETE), DCL (GRANT, REVOKE), TCL (COMMIT, ROLLBACK, SAVEPOINT).

Integrity Independence

- Integrity rules must be defined in the relational catalog (data dictionary), NOT in application programs.
- Example: Instead of checking in Java if age > 18, the constraint is defined in SQL: age INT CHECK (age > 18).
- This ensures the rule is enforced regardless of which application accesses the database.

Systematic Treatment of NULL Values

- NULL must represent missing, unknown, or inapplicable information — not zero or empty string.
- All operations (arithmetic, comparison, aggregate) must handle NULLs consistently.
- NULL + 5 = NULL. NULL = NULL is UNKNOWN (not TRUE). Use IS NULL to check for nulls.

Physical Data Independence

- Changing the physical storage (B-tree → hash index, HDD → SSD) must not affect SQL queries or application programs.
- Example: If DBA adds a new index on salary column for performance, no application code changes.

High-Level Insert, Update, Delete

- DML must support multi-row operations in a single statement.
- Example: UPDATE Employee SET salary = salary * 1.1 WHERE dept_no = 10; — updates ALL employees in dept 10 at once.

Q33. Relational Integrity Constraints — Domain, Referential, Enterprise.

1. Domain Constraints

- A domain constraint specifies the valid range, format, or type of values that an attribute can take.
- It defines the DATA TYPE and optional CHECK condition for each column.
- Example: age must be an integer between 0 and 120. gender must be 'M' or 'F'. salary must be > 0.

```
CREATE TABLE Employee (
```

```
  emp_id    INT,
  age       INT CHECK (age BETWEEN 18 AND 65),
  gender    CHAR(1) CHECK (gender IN ('M', 'F')),
  salary    DECIMAL(10,2) CHECK (salary > 0)
);
```

2. Referential Integrity

- Referential integrity ensures that a foreign key value in a child table must match an existing primary key in the parent table (or be NULL).
- It maintains consistency between related tables.
- Example: dept_no in Employee must exist in Department(dept_no).

```
CREATE TABLE Employee (
  emp_id  INT PRIMARY KEY,
  dept_no INT,
  FOREIGN KEY (dept_no) REFERENCES Department(dept_no)
  ON DELETE SET NULL
);
```

3. Enterprise Constraints (Semantic Constraints)

- Enterprise constraints are business-specific rules that cannot be fully expressed by primary/foreign key constraints alone.
- They reflect real-world business policies.
- Examples: An employee's salary must not exceed their manager's salary. A student can take at most 6 courses per semester. An account balance must never go negative.
- These are enforced using CHECK constraints, triggers, or application logic.

Q34. Features of Good Relational Database Design.

- — Minimal Redundancy

The same data should not be stored in more than one place unnecessarily. Redundancy wastes storage and causes anomalies.

- — Avoidance of Update, Insert, and Delete Anomalies

A good design ensures that insert, update, and delete operations do not create inconsistencies. Anomalies are signs of poor design.

- — Preservation of Functional Dependencies

All functional dependencies from the original schema must be preserved after decomposition. Every constraint must be enforceable.

- — Lossless Join Decomposition

When a table is split into smaller tables, joining them back must give exactly the original table — no extra or missing rows.

- — Strong Primary Keys

Every table must have a well-chosen primary key that is unique, stable, and minimal.

- — Appropriate Use of NULL Values

NULLs should be minimized. Excessive NULLs indicate that a relation has too many attributes of different entities.

- — Data Integrity Constraints

Domain, referential, and enterprise constraints must be clearly defined and enforced.

Q35 & Q36. Anomalies in Relational Model and Normalization.

Consider a poorly designed table: Student_Course(rollno, student_name, course_id, course_name, marks)

1. Insertion Anomaly

- You CANNOT insert a new course into the database unless at least one student is enrolled in it.
- Example: Want to add course 'AI' (C05), but since no student is enrolled yet, there is no row to add it — the rollno and marks would be NULL/unknown.

2. Update Anomaly

- If a course name changes, you must update it in EVERY row where that course appears.
- Example: Renaming 'DBMS' to 'Database Systems' requires updating 50+ rows (one per student). If even one is missed, the database becomes inconsistent.

3. Deletion Anomaly

- Deleting a student's record may unintentionally delete course information.
- Example: If the only student enrolled in 'AI' drops out and we delete their row, the entire AI course record is lost from the database.

How Normalization Removes Anomalies

- Normalization decomposes the table into smaller, well-structured relations based on functional dependencies.
- Decomposed tables: Student(rollno, student_name), Course(course_id, course_name), Enrollment(rollno, course_id, marks).
- Now a new course can be added to Course table without any student. Updating course_name changes only ONE row in Course. Deleting a student from Enrollment doesn't affect Course table.

Q37. Normalization — 1NF, 2NF, 3NF with examples.

Why Normalization?

- Normalization is the process of organizing database tables to reduce redundancy and avoid anomalies.
- It works by applying a series of rules (Normal Forms) to ensure data is stored correctly.

1NF — First Normal Form

- A table is in 1NF if all column values are ATOMIC (indivisible) and there are no repeating groups.
- Each column must contain single values, not sets or lists.

- Violation Example: Student(rollno, name, {phone1, phone2}) — phone is multi-valued.
- Fix: Create separate row for each phone number OR create a Student_Phone(rollno, phone) table.

2NF — Second Normal Form

- A table is in 2NF if it is in 1NF AND every non-key attribute is FULLY FUNCTIONALLY DEPENDENT on the ENTIRE primary key.
- Applies only when primary key is composite.
- Violation Example: Enrollment(rollno, course_id, student_name, marks). Here student_name depends only on rollno (partial dependency on composite key {rollno, course_id}).
- Fix: Split into Student(rollno, student_name) and Enrollment(rollno, course_id, marks).

3NF — Third Normal Form

- A table is in 3NF if it is in 2NF AND there is NO TRANSITIVE DEPENDENCY (non-key attribute depending on another non-key attribute).
- Violation Example: Employee(emp_id, dept_id, dept_location). dept_location depends on dept_id, and dept_id depends on emp_id — transitive dependency.
- Fix: Split into Employee(emp_id, dept_id) and Department(dept_id, dept_location).

Q38, Q39, Q41. 2NF, 3NF, and BCNF — Detailed with Examples.

2NF — Example

Table: OrderDetail(order_id, product_id, customer_name, product_price, quantity)

- Primary Key: {order_id, product_id} (composite)
- Partial Dependency 1: customer_name depends only on order_id → violates 2NF
- Partial Dependency 2: product_price depends only on product_id → violates 2NF

Decomposition to 2NF:

```
Order(order_id, customer_name)
Product(product_id, product_price)
OrderDetail(order_id, product_id, quantity)   -- fully dependent on composite key
```

3NF — Example

Table: Student(rollno, zip_code, city, state)

- FDs: rollno → zip_code, zip_code → city, zip_code → state
- Transitive Dependency: city and state depend on zip_code, not directly on rollno.

Decomposition to 3NF:

```
Student(rollno, zip_code)
ZipCode(zip_code, city, state)
```

BCNF — Boyce-Codd Normal Form

- BCNF is a stronger version of 3NF.
- A relation is in BCNF if for every functional dependency X → Y, X must be a SUPERKEY.
- Difference: 3NF allows X → Y if Y is a prime attribute (part of candidate key). BCNF does not allow this exception.

Violation Example: Table Teach(student, subject, teacher) with FDs: {student, subject} → teacher AND teacher → subject

- {student, subject} is a candidate key. teacher → subject means teacher determines a prime attribute — this violates BCNF but not 3NF.

Decomposition to BCNF:

```
Teacher_Subject(teacher, subject)    -- teacher is superkey here
Student_Teacher(student, teacher)
```

Comparison: 3NF vs BCNF

Feature	3NF	BCNF
Redundancy Eliminated	Partial	Complete
Dependency Preservation	Always possible	May not always be possible
Lossless Decomposition	Yes	Yes
Exception for Prime Attr	Allowed	Not Allowed
Strictness	Less strict	More strict

Q42. Functional Dependencies — 3NF Check and Conversion.

What is a Functional Dependency?

- A functional dependency X → Y means: for every value of X, there is exactly one value of Y.
- X is called the DETERMINANT and Y is the DEPENDENT.
- FDs are used to identify redundancy and guide normalization.
- Example: rollno → student_name means each rollno determines exactly one student name.

For schema: Student(RollNo, Branch_code, Marks_Obtained, Exam_Name, Total_Marks)

Identified Functional Dependencies:

```
RollNo          → Branch_code
RollNo, Exam_Name → Marks_Obtained
Exam_Name       → Total_Marks
```

Is this in 3NF?

- Primary Key: {RollNo, Exam_Name}
- FD: Exam_Name → Total_Marks — here Exam_Name is part of the primary key and Total_Marks is a non-prime attribute.
- FD: RollNo → Branch_code — here RollNo is part of primary key and Branch_code is non-prime.
- These are partial dependencies (non-prime attributes depend on part of the composite primary key).
- So the relation is NOT even in 2NF, let alone 3NF.

Conversion to 3NF

```
Student_Branch (RollNo, Branch_code)
```

```
   -- RollNo is PK; Branch_code fully depends on RollNo

Exam (Exam_Name, Total_Marks)
   -- Exam_Name is PK; Total_Marks fully depends on Exam_Name

Student_Exam (RollNo, Exam_Name, Marks_Obtained)
   -- Composite PK {RollNo, Exam_Name}; Marks_Obtained fully depends on
both
```

Q43. Functional Dependencies in Market(MarketName, Product, Stock).

Given instance of Market table:

MarketName	Product	Stock
S1	Toothpaste	14
S1	Biscuits	8
S1	Shampoo	8
S2	Toothpaste	30
M1	Chocolates	50
M2	Cakes	14

FDs identified from this instance:

- {MarketName, Product} → Stock — a specific market stocks a specific product in a specific quantity. (Composite PK)
- We cannot determine MarketName → Stock alone (S1 has different stocks for different products).
- We cannot determine Product → Stock alone (Toothpaste has stock 14 in S1 and 30 in S2).
- No single attribute determines another — only the composite {MarketName, Product} uniquely determines Stock.

Conclusion: The only functional dependency is {MarketName, Product} → Stock. The table is already in BCNF since the left side of the only FD is the primary key (a superkey).

Q44 & Q45. Partial, Full, and Transitive Dependencies.

Full Functional Dependency

- Y is FULLY functionally dependent on X if X → Y and removing any attribute from X breaks the dependency.
- Example: In Enrollment(rollno, course_id, marks): {rollno, course_id} → marks. If we remove rollno, course_id alone doesn't determine marks. If we remove course_id, rollno alone doesn't determine marks. So marks is FULLY dependent on the composite key.

Partial Dependency

- Y is PARTIALLY dependent on X if Y depends only on a PROPER SUBSET of a composite key X.
- Partial dependency violates 2NF.
- Example: In Enrollment(rollno, course_id, student_name, marks): student_name depends only on rollno — partial dependency since the full key is {rollno, course_id}.
- Fix: Separate student_name into Student(rollno, student_name).

Transitive Dependency

- A transitive dependency exists when a non-key attribute Y depends on another non-key attribute Z, which in turn depends on the primary key X.
- In other words: X → Z and Z → Y, but Z is NOT a candidate key.
- Transitive dependency violates 3NF.
- Example: Employee(emp_id, dept_id, dept_name): emp_id → dept_id (direct), dept_id → dept_name (dept_name transitively depends on emp_id via dept_id).
- Fix: Create Department(dept_id, dept_name) and Employee(emp_id, dept_id).

Q46. Desirable Properties of Decomposition.

When a relation is decomposed into smaller relations, the decomposition must satisfy these properties:

1. Lossless Join (Non-Additive Join)

- When the decomposed tables are joined back (natural join), the result must be EXACTLY the original relation.
- No extra rows (spurious tuples) should appear, and no rows should be missing.
- This is ensured when the common attribute(s) between the decomposed relations is a CANDIDATE KEY in at least one of them.
- Example: R(A, B, C) decomposed into R1(A, B) and R2(A, C). If A → B or A → C, the join on A gives back the original R without spurious tuples.

2. Dependency Preservation

- Every functional dependency of the original schema should be enforceable in at least one of the decomposed tables WITHOUT joining them.
- If we must JOIN tables to verify a constraint, it adds performance overhead and complexity.
- Example: If FD is A → B, then A and B must appear together in at least one decomposed relation.
- BCNF sometimes sacrifices dependency preservation. 3NF always allows dependency-preserving decomposition.

Why both properties matter:

- Lossless join ensures no information is lost and no false data is generated.
- Dependency preservation ensures data integrity constraints can still be enforced efficiently.

Q47. Lossless/Lossy Decomposition check for F(FN, PN, C, D).

Given FDs: FN, PN → C | C → D | D → F
Decomposition: F1(FN, PN, C) and F2(C, D)

Method: Check common attribute

- Common attribute between F1 and F2: C
- Is C a candidate key in F1? In F1(FN, PN, C), the FD FN,PN → C tells us C is determined by {FN,PN}. C alone does NOT determine everything in F1, so C is NOT a superkey in F1.
- Is C a candidate key in F2? In F2(C, D), the FD C → D tells us C determines D. So C is indeed a KEY (superkey) in F2.

Conclusion

Result: Since C is a key in F2, the decomposition F1(FN, PN, C) and F2(C, D) is LOSSLESS (non-lossy). When we join F1 and F2 on C, we get back exactly the original relation F without any spurious tuples.

Verification using Tableau method

- Start: F1 row — (a, b, c) for FN, PN, C and (d1, d2, d3, d4) for others
- F2 row — (b1, b2, c, d) for FN, PN, C, D
- After applying C → D: both rows get same D value
- A row with all 'a' values (no subscript) is achieved → Lossless confirmed

UNIT IV — Database Transactions

Topics: ACID Properties, Schedules, Serial Schedule, Serializability, Cascaded Aborts, Recoverable Schedules, Concurrency Control, Locking

Q48. ACID Properties and Transaction States.

ACID Properties

A — Atomicity: A transaction is an all-or-nothing unit. Either ALL operations in the transaction complete successfully, or NONE of them are applied. If any operation fails (like a system crash mid-transaction), all previous operations of that transaction are rolled back (undone). Example: In a bank transfer — debit from A and credit to B — if the credit fails, the debit must be reversed.

C — Consistency: A transaction must take the database from one VALID state to another VALID state. All integrity constraints must be satisfied before and after the transaction. Example: Total money in the banking system must remain the same before and after a transfer.

I — Isolation: Transactions executing concurrently must not interfere with each other. Each transaction must see the database as if it alone is executing. Intermediate states of a transaction are NOT visible to other transactions. Example: If T1 is transferring money from A to B, T2 reading balances mid-transfer must not see an inconsistent state.

D — Durability: Once a transaction COMMITS, its changes are PERMANENT — even if the system crashes immediately after. The committed data is saved to stable storage (disk) and can be recovered after restart.

Transaction States

State	Description	Transition Trigger
Active	Transaction is executing (initial state)	Transaction begins
Partially Committed	Final statement executed, not yet committed	Last operation done
Committed	Transaction completed successfully, changes saved	COMMIT issued
Failed	Error encountered, cannot proceed	System failure or error
Aborted	Transaction rolled back, database restored	After ROLLBACK
Terminated	Transaction done (either committed or aborted)	After commit/abort

Possible Sequences of States

- Normal path: Active → Partially Committed → Committed → Terminated
- Failure path: Active → Failed → Aborted → Terminated
- Partial execution failure: Active → Partially Committed → Failed → Aborted → Terminated

Q49. What is Conflict Serializability? How to check it?

What is a Schedule?

- A schedule is a sequence of instructions (read/write operations) from multiple concurrent transactions.
- A SERIAL schedule executes transactions one at a time — no interleaving. Always correct but slow.
- A CONCURRENT schedule interleaves operations from multiple transactions — faster but may be incorrect.

Conflict Serializability

- A schedule is CONFLICT SERIALIZABLE if it can be converted into a serial schedule by swapping non-conflicting adjacent operations.
- Two operations CONFLICT if ALL three conditions hold: they belong to DIFFERENT transactions, they access the SAME data item, and at least one is a WRITE operation.

Conflicting operation pairs

- Read(X) and Write(X) from different transactions — CONFLICT
- Write(X) and Read(X) from different transactions — CONFLICT
- Write(X) and Write(X) from different transactions — CONFLICT
- Read(X) and Read(X) from different transactions — NO CONFLICT

How to check — Precedence Graph (Serialization Graph)

14. Draw one node for each transaction.
15. For each pair of conflicting operations where Ti's operation comes BEFORE Tj's, draw an arrow: Ti → Tj.
16. If the graph has NO CYCLE → schedule is CONFLICT SERIALIZABLE.
17. If the graph has a CYCLE → NOT conflict serializable.

Example

```
T1: R(A), W(A), R(B), W(B)
T2: R(A), W(A)
Schedule: R1(A), R2(A), W2(A), R1(B), W1(A), W1(B)

Conflicts:
  W2(A) before W1(A) → T2 → T1
  R1(A) before W2(A) → T1 → T2

Graph: T1 → T2 and T2 → T1 → CYCLE → NOT conflict serializable
```

Q50. View Serializability — Concept and Check.

View Equivalence

- Two schedules S1 and S2 are VIEW EQUIVALENT if:

- 1) For each data item Q, if Ti reads the INITIAL value of Q in S1, then Ti reads the initial value of Q in S2.
- 2) For each read of Q by Ti in S1, if the value was written by Tj, then in S2 the same read of Q by Ti also reads the value written by Tj.
- 3) For each data item Q, the FINAL WRITE operation on Q is done by the same transaction in both schedules.

View Serializability

- A schedule S is VIEW SERIALIZABLE if it is view equivalent to some SERIAL schedule.
- Every conflict serializable schedule is also view serializable, but NOT vice versa.
- View serializability allows some schedules that conflict serializability rejects — particularly blind writes.

Conflict Equivalent Schedules

- Two schedules are CONFLICT EQUIVALENT if they have the same set of conflicting operations in the same order.

Checking the given schedule (T1 and T2)

- Compare with all possible serial schedules: T1→T2 and T2→T1.
- Check all three conditions of view equivalence for each serial schedule.
- If any serial schedule satisfies all three conditions → schedule is VIEW SERIALIZABLE.
- Note: The given schedule has write(A) by T1 AFTER write(A) by T2 (W1 is the final write on A). Check if this matches the serial schedules.

Q51. Conflict Serializability check for T1, T2, T3, T4.

Operations: T1:R(X) | T2:R(Z),W(X) | T3:R(Y),W(Y) | T4:W(X),W(Y),W(Z)

Step 1: Identify Conflicts

- T1:R(X) before T2:W(X) → T1 → T2 (R-W conflict on X)
- T2:W(X) before T4:W(X) → T2 → T4 (W-W conflict on X)
- T3:W(Y) before T4:W(Y) → T3 → T4 (W-W conflict on Y)
- T2:R(Z) before T4:W(Z) → T2 → T4 (R-W conflict on Z)

Step 2: Draw Precedence Graph

```
T1 → T2 (due to R1(X) before W2(X))
T2 → T4 (due to W2(X) before W4(X) and R2(Z) before W4(Z))
T3 → T4 (due to W3(Y) before W4(Y))
```

Step 3: Check for Cycle

- Edges: T1→T2, T2→T4, T3→T4
- No edge forms a cycle (no path comes back to the same node).

Conclusion: The precedence graph has NO CYCLE. Therefore, the schedule is CONFLICT SERIALIZABLE. The equivalent serial order is: T1, T2, T3, T4 OR T1, T3, T2, T4.

Q52. Recoverable Schedules.

What is a Recoverable Schedule?

- A schedule is RECOVERABLE if: whenever a transaction Tj reads data written by Ti, then Ti must COMMIT before Tj commits.
- In other words: if Tj depends on Ti's data, Ti must commit first.
- This ensures that if Ti aborts, Tj can also be safely aborted — the committed data of Tj is never based on aborted data.

Why is Recoverability Desirable?

- If Tj commits before Ti, and Ti later aborts, then Tj's committed changes are based on invalid (rolled-back) data.
- This creates an IRRECOVERABLE state — we cannot undo Tj's commit (committed data is permanent per Durability).
- Recoverability guarantees that the database can always be restored to a consistent state after any failure.

Non-Recoverable Schedules

- A schedule where Tj commits before Ti (when Tj read Ti's dirty data) is NON-RECOVERABLE.
- These should generally NOT be allowed.
- However, in some real-time applications (like sensor data logging) where performance is critical and rollback is handled externally, non-recoverable schedules might be accepted with full awareness of the risk.

Cascading Rollback

- If Ti aborts and Tj read Ti's data, then Tj must also be rolled back (even if Tj itself has no error).
- If Tk read Tj's data, Tk must also be rolled back — this is CASCADING ROLLBACK.
- Cascadeless schedules avoid this by ensuring transactions only read committed data.

Q53 & Q54. Log-Based Recovery — Deferred and Immediate Modifications.

What is a Transaction Log?

- A LOG is a sequential file on stable storage (disk) that records every operation performed by every transaction.
- The log is written BEFORE the actual database changes (Write-Ahead Logging — WAL protocol).
- Each log record contains: Transaction ID, Data item name, Old value, New value, and operation type.

Types of Log Records

- <Ti, start> — transaction Ti has started
- <Ti, X, old_value, new_value> — Ti wrote value to data item X
- <Ti, commit> — Ti has committed
- <Ti, abort> — Ti has aborted

i) Deferred Database Modification

- In deferred modification, ALL changes are deferred (postponed) until the transaction commits.
- During execution, only LOG records are written to stable storage — the actual database is NOT modified.
- Only after the <Ti, commit> log record is written are the actual database changes applied.
- Recovery: If failure before commit → NO database changes need to be undone (nothing was written). If failure after commit but before database update → REDO all committed transactions using the log.
- No UNDO needed — simplifies recovery. But requires keeping all changes in memory until commit.

ii) Immediate Database Modification

- In immediate modification, database changes are written to disk IMMEDIATELY during transaction execution (even before commit).
- Both log records AND database changes are written before the transaction commits.
- Log must be written BEFORE the database change (WAL — Write-Ahead Logging).
- Recovery: If failure before commit → UNDO all changes of that transaction using old_value from log. If failure after commit → REDO changes using new_value from log to ensure all committed changes are reflected.
- Requires both UNDO (for uncommitted) and REDO (for committed) operations during recovery.

Q55. Locks — Shared Lock and Exclusive Lock.

Why Locks?

- In a concurrent system, multiple transactions access the same data simultaneously.
- Without control, this leads to: Lost Update, Dirty Read, Unrepeatable Read, and Phantom Read problems.
- LOCKS are used to prevent transactions from interfering with each other.

Shared Lock (S-Lock / Read Lock)

- A SHARED LOCK is acquired when a transaction wants to READ a data item.
- Multiple transactions can hold shared locks on the same data item simultaneously.
- While a shared lock is held, NO transaction can get an exclusive lock (no writes allowed).
- Notation: lock-S(X) — shared lock on data item X
- Example: T1 and T2 both want to read balance of account A. Both get S-lock on A and read concurrently.

Exclusive Lock (X-Lock / Write Lock)

- An EXCLUSIVE LOCK is acquired when a transaction wants to WRITE (update/insert/delete) a data item.
- Only ONE transaction can hold an exclusive lock on a data item at any time.
- While an exclusive lock is held, NO other transaction can get ANY lock (shared or exclusive).
- Notation: lock-X(X) — exclusive lock on data item X
- Example: T1 wants to update salary of employee 101. T1 gets X-lock on that row. T2 must wait until T1 releases.

Lock Compatibility Matrix

Request →	S-Lock	X-Lock
S-Lock held	Compatible (GRANT)	Incompatible (WAIT)
X-Lock held	Incompatible (WAIT)	Incompatible (WAIT)

Q56. Two-Phase Locking (2PL) Protocol.

What is 2PL?

- Two-Phase Locking (2PL) is a concurrency control protocol that ensures CONFLICT SERIALIZABILITY.
- A transaction following 2PL divides its execution into exactly TWO phases.

Phase 1 — Growing Phase (Lock Acquisition)

- The transaction can ACQUIRE any type of lock (shared or exclusive).
- The transaction CANNOT release any lock during this phase.
- The point where the transaction acquires its last lock is called the LOCK POINT.

Phase 2 — Shrinking Phase (Lock Release)

- The transaction can RELEASE locks.
- The transaction CANNOT acquire any new lock during this phase.
- Once the first lock is released, the transaction is in the shrinking phase.

Strict 2PL

- All EXCLUSIVE locks are held until the transaction COMMITS or ABORTS.
- Shared locks can be released before commit.
- Prevents dirty reads and cascading rollbacks.
- More restrictive but safer than basic 2PL.

Rigorous 2PL

- ALL locks (both shared and exclusive) are held until the transaction COMMITS or ABORTS.
- Strongest isolation — no lock released until the very end.
- Transactions appear to execute serially.
- Used in most commercial DBMS systems.

Limitation of 2PL

- 2PL guarantees serializability but can cause DEADLOCKS when two transactions each wait for each other's lock.

Q57. Adding Locks to T31 and T32 — Deadlock Possibility.

T31 with 2PL locks

```
lock-S(A);
read(A);
lock-S(B);
read(B);
lock-X(B);   -- upgrade to exclusive for write
if A = 0 then B := B + 1;
write(B);
unlock(A);
unlock(B);
```

T32 with 2PL locks

```
lock-S(B);
read(B);
lock-S(A);
read(A);
lock-X(A);   -- upgrade to exclusive for write
if B = 0 then A := A + 1;
write(A);
unlock(B);
unlock(A);
```

Can Deadlock Occur?

- YES — a deadlock can occur in the following scenario:
- T31 acquires S-lock on A. T32 acquires S-lock on B.
- T31 tries to acquire S-lock on B → WAITS (T32 holds it).
- T32 tries to acquire S-lock on A → WAITS (T31 holds it).
- Both transactions wait for each other indefinitely — this is a DEADLOCK.

Q58. Starvation Problem in Locking.

Problem — Starvation (Indefinite Postponement)

- Starvation occurs when a transaction T waits for an EXCLUSIVE (X) lock on item Q, but a continuous stream of other transactions keep getting SHARED (S) locks on Q.
- As long as at least one shared lock exists, the exclusive lock cannot be granted.
- New shared lock requests keep arriving before old ones are released.

- Transaction T is perpetually delayed and never gets the exclusive lock — it STARVES.

How 2PL Solves Starvation

- Modern implementations of 2PL use a FIRST-COME-FIRST-SERVED (FIFO) waiting queue for locks.
- Once transaction T is waiting for an X-lock on Q, no NEW shared lock requests on Q are granted after T entered the queue.
- Existing shared locks are served out, but new shared lock requests go BEHIND T in the queue.
- Once all existing S-locks are released, T gets the X-lock — starvation is prevented.
- Some systems use priority aging — a waiting transaction's priority increases over time to prevent starvation.

Q59. Deadlocks — When they happen, Prevention, and Recovery.

When do Deadlocks Happen?

- A deadlock occurs when two or more transactions are waiting for each other's locks, forming a circular wait.
- Conditions for deadlock (Coffman's conditions): Mutual Exclusion, Hold and Wait, No Preemption, and Circular Wait — all four must hold.
- Example: T1 holds lock on A, waits for B. T2 holds lock on B, waits for A — deadlock.

Deadlock Prevention

- Wait-Die Scheme: If Ti (older) requests a lock held by Tj (younger), Ti WAITS. If Ti (younger) requests a lock held by Tj (older), Ti is KILLED (dies) and restarted. Older transactions never wait for younger ones.
- Wound-Wait Scheme: If Ti (older) requests a lock held by Tj (younger), Ti PREEMPTS (wounds) Tj — Tj is rolled back. If Ti (younger) requests a lock held by Tj (older), Ti WAITS. Older transactions never wait.
- Timeout-Based: A transaction is aborted if it waits too long for a lock.
- Lock Ordering: All transactions acquire locks in the SAME predefined order — eliminates circular wait.

Deadlock Detection

- The system maintains a WAIT-FOR GRAPH (WFG) where nodes are transactions and edges represent waiting relationships.
- If a CYCLE exists in the WFG → Deadlock detected.
- The deadlock detector runs periodically to check for cycles.

Deadlock Recovery

- Select a VICTIM transaction (usually the one with least cost — fewest updates, youngest, or least progress).
- ROLLBACK the victim — release all its locks.
- Other transactions can now proceed.
- To prevent the same transaction from being victimized repeatedly, use a rollback counter — if a transaction is rolled back too many times, give it higher priority.

Q60 & Q61. Timestamp-Based Concurrency Control Protocol.

What is a Timestamp?

- Every transaction Ti is assigned a unique TIMESTAMP TS(Ti) when it enters the system — usually the system clock or a counter.
- Older transactions have smaller timestamps. Newer transactions have larger timestamps.

R-timestamp and W-timestamp

- R-timestamp(Q) — the LARGEST timestamp of any transaction that SUCCESSFULLY READ data item Q.
- W-timestamp(Q) — the LARGEST timestamp of any transaction that SUCCESSFULLY WROTE data item Q.

Protocol Rules for Read by Ti

- If TS(Ti) < W-timestamp(Q): Ti is trying to read a value that was already overwritten by a newer transaction. Ti is OUT OF ORDER → Ti is ROLLED BACK (aborted and restarted with a new timestamp).
- If TS(Ti) ≥ W-timestamp(Q): The read is valid. Execute the read. Set R-timestamp(Q) = max(R-timestamp(Q), TS(Ti)).

Protocol Rules for Write by Ti

- If TS(Ti) < R-timestamp(Q): A newer transaction already read Q and expected Ti not to write. Writing now would invalidate that read → Ti is ROLLED BACK.
- If TS(Ti) < W-timestamp(Q): A newer transaction already wrote Q. Ti's write is obsolete → Ti is ROLLED BACK.
- Otherwise: Execute the write. Set W-timestamp(Q) = TS(Ti).

Advantages of Timestamp Protocol

- No locks required → No deadlocks possible.
- Automatically ensures conflict serializability.
- Transactions execute in timestamp (arrival) order.

Disadvantages

- A transaction may be rolled back and restarted many times (starvation possible).
- Long-running transactions may keep getting aborted by newer ones.

UNIT V — NoSQL Databases

Topics: Introduction to NoSQL, NoSQL data models, CAP Theorem, BASE Properties, SQL vs NoSQL, MongoDB CRUD, Indexing, Aggregation

Q62. NoSQL vs RDBMS — Key-Value Store Data Model.

How NoSQL Differs from RDBMS

Feature	RDBMS (SQL)	NoSQL
Data Model	Tables with fixed rows/columns	Flexible — key-value, document, column, graph
Schema	Fixed, predefined schema	Schema-less or dynamic schema
Scalability	Vertical (bigger server)	Horizontal (add more servers)
ACID / BASE	ACID compliant	BASE (eventual consistency)
Query Language	SQL (standardized)	Varies by database — no standard
Joins	Supports complex joins	Joins avoided — data denormalized
Best For	Complex queries, transactions	Big data, real-time, unstructured data
Examples	MySQL, Oracle, PostgreSQL	MongoDB, Cassandra, Redis, Neo4j

Key-Value Store — NoSQL Data Model

- The SIMPLEST NoSQL data model.
- Data is stored as key-value PAIRS — like a dictionary or hash map.
- Key: unique identifier. Value: any data — string, number, JSON, binary, image.
- The database does not understand the structure of the value — it is opaque.
- Extremely FAST for simple lookups since access is O(1) via the key.

Example

```
Key                 → Value
'user:123'          → {name: 'Alice', age: 25, city: 'Mumbai'}
'session:abc'       → {user: 'Alice', logged_in: true, expires: '2025-01-01'}
'cart:123'          → ['item1', 'item2', 'item3']
```

Popular Key-Value Stores

- Redis — in-memory, used for caching, session management, leaderboards.
- Amazon DynamoDB — fully managed, used at massive scale.
- Apache Cassandra — handles both key-value and column-family storage.

Use Cases

- Caching (store computed results for fast retrieval)
- Session management (store logged-in user data)
- Shopping carts, user preferences, configuration settings

Q63. Column-Oriented, Graph, and Document-Oriented NoSQL.

i) Column-Oriented (Column-Family) Database

- Data is stored COLUMN BY COLUMN instead of row by row.
- Each row has a row key, and data is grouped into column families.
- Different rows in the same table can have different columns.
- Excellent for READ-HEAVY analytical queries where only a few columns are needed from millions of rows.

```
Row Key: 'user001'
  PersonalInfo: {name: 'Alice', age: 25}
  ContactInfo:  {email: 'alice@mail.com', phone: '9876543210'}
```

- Examples: Apache Cassandra, Apache HBase, Google Bigtable.
- Use Cases: Time-series data, IoT sensor data, financial analytics, social media analytics.

ii) Graph Database

- Data is stored as NODES (entities) and EDGES (relationships between entities) with properties on both.
- Optimized for queries that involve traversing relationships (connections) between many entities.
- Relationship traversal is O(1) — much faster than SQL JOINs for deeply connected data.

```
Node: (Person {name: 'Alice', age: 25})
Node: (Person {name: 'Bob',   age: 30})
Edge: Alice -[FRIENDS_WITH]-> Bob
Edge: Alice -[WORKS_AT]-> (Company {name: 'TechCorp'})
```

- Examples: Neo4j, Amazon Neptune, ArangoDB.
- Use Cases: Social networks (friend suggestions), fraud detection, recommendation engines, knowledge graphs.

iii) Document-Oriented Database

- Data is stored as DOCUMENTS — usually JSON, BSON, or XML format.
- Each document is a self-contained unit with its own structure — no fixed schema.
- Documents are grouped into COLLECTIONS (like tables in RDBMS).
- Different documents in the same collection can have different fields.

```
{
  '_id': 'u001',
  'name': 'Alice',
```

```
    'address': { 'city': 'Mumbai', 'zip': '400001' },
    'orders': [ {id: 1, item: 'Laptop'}, {id: 2, item: 'Mouse'} ]
}
```

- Examples: MongoDB, CouchDB, Firebase Firestore.
- Use Cases: Content management systems, e-commerce product catalogs, user profiles, blogs.

Q68, Q69, Q70, Q71, Q72. CAP Theorem and BASE Properties.

CAP Theorem

- The CAP Theorem (proposed by Eric Brewer, 2000) states that a distributed database system can guarantee AT MOST TWO out of the following three properties SIMULTANEOUSLY:

C — Consistency: All nodes in the distributed system see the SAME data at the SAME time. A read always returns the most recently written value. Example: After a bank transfer, every node shows the updated balance.

A — Availability: Every request (read or write) receives a RESPONSE — it will not result in an error or timeout. The response may not always be the most recent data. Example: The system always responds, even during network issues.

P — Partition Tolerance: The system continues to FUNCTION even when network partitions occur — i.e., some nodes cannot communicate with others due to network failure. In real distributed systems, network failures are unavoidable, so P is usually always required.

CAP Trade-offs

- Since P (partition tolerance) is almost always required in distributed systems, the real choice is between C and A:
- CP systems (Consistency + Partition Tolerance): Sacrifice availability. Example: HBase, MongoDB (in some configs), ZooKeeper. Returns error instead of stale data.
- AP systems (Availability + Partition Tolerance): Sacrifice consistency. Example: Cassandra, DynamoDB, CouchDB. Returns possibly stale data but always responds.
- CA systems (Consistency + Availability): Only possible when there are no network partitions — practical only for single-node systems. Example: Traditional RDBMS (MySQL, PostgreSQL) on a single server.

BASE Properties

BASE is an alternative to ACID used by NoSQL databases:

BA — Basically Available: The system guarantees availability (as defined in CAP). The system may return a response that is stale (not the latest) but will always give a response rather than failing. Example: Amazon shopping cart — always available to add items, even during some node failures.

S — Soft State: The state of the system may CHANGE OVER TIME even without new inputs. This is because data is still propagating through replicas. The system is not always in a consistent state. Example: A social media post may appear to some users immediately and to others after a few seconds.

E — Eventual Consistency: If no new updates are made to a data item, eventually ALL replicas will converge to the SAME value. Consistency is not immediate — it happens eventually. Example: If you update your Facebook profile picture, some users may see the old picture for a few seconds before all servers sync.

Soft State and Eventual Consistency Relationship

- The SOFT STATE of the system directly depends on EVENTUAL CONSISTENCY.
- Because the system is eventually consistent (not immediately), the current state is 'soft' — it can still change as updates propagate.
- Once all replicas converge (eventual consistency is achieved), the soft state stabilizes into a consistent state.
- This allows the system to be highly available and partition tolerant while sacrificing strict immediate consistency.

Q73 & Q74. SQL vs NoSQL — Detailed Comparison.

Feature	SQL (RDBMS)	NoSQL
Full Form	Structured Query Language DB	Not Only SQL
Data Model	Relational tables	Key-Value, Document, Column, Graph
Schema	Fixed, predefined schema	Dynamic, schema-less
Query Language	SQL — standardized	Varies (MongoDB uses JSON-based queries)
Scalability	Vertical scaling (upgrade hardware)	Horizontal scaling (add more nodes)
Transactions	Full ACID compliance	BASE — limited ACID in some systems
Consistency	Strong consistency	Eventual consistency
Relationships	Foreign keys and joins	Embedded documents / denormalization
Data Type	Structured data only	Structured, semi-structured, unstructured
Best Performance	Complex queries, reporting	Simple queries on huge datasets
Examples	MySQL, Oracle, SQL Server, PostgreSQL	MongoDB, Cassandra, Redis, Neo4j, HBase
Use Cases	Banking, ERP, CRM, inventory	Social media, IoT, e-commerce, gaming
Maturity	Very mature (50+ years)	Relatively newer (2000s)

Q75. NoSQL in Social Networking — Need over RDBMS.

Social Networking Data Characteristics

- VOLUME: Billions of users generate terabytes of data daily — posts, likes, comments, messages, images, videos.
- VELOCITY: Data arrives at very high speed — millions of new posts per minute.
- VARIETY: Data is highly diverse — text, images, videos, locations, relationships.
- VARIABILITY: User profiles have different attributes — no fixed structure.

Why RDBMS Fails for Social Networks

- RDBMS requires a fixed schema — social data evolves constantly (new post types, story formats, reels).
- RDBMS scales vertically (bigger server) — not cost-effective for billions of users.
- Complex JOIN operations for friend networks are very slow in RDBMS at scale.
- RDBMS enforces strict ACID — at massive scale, this creates bottlenecks.

Why NoSQL is Better for Social Networks

- Graph databases (Neo4j) model friend connections naturally — finding mutual friends, suggestions are extremely fast using graph traversal.
- Document stores (MongoDB) handle flexible user profiles — each profile can have different fields.
- Key-value stores (Redis) provide ultra-fast session management and caching of trending content.
- Column-family stores (Cassandra) handle time-series data like activity feeds and notifications at massive scale.
- Horizontal scaling — add hundreds of commodity servers as user base grows.
- Eventual consistency is acceptable — seeing a friend's post 2 seconds late is acceptable; a bank transaction cannot be delayed.

Real-World Examples

- Facebook: Uses Cassandra for inbox messages, MySQL + TAO (graph-like) for social graph.
- Twitter: Uses Cassandra for tweets timeline.
- LinkedIn: Uses Voldemort (key-value store).
- Instagram: Uses Cassandra for photo metadata.

Q76. CRUD Operations in MongoDB with Examples.

MongoDB stores data in COLLECTIONS of DOCUMENTS (JSON/BSON format). CRUD = Create, Read, Update, Delete.

C — Create (Insert)

```
// Insert a single document
```

```
db.students.insertOne({
  rollno: 101,
  name: 'Alice',
  branch: 'Computer Science',
  marks: 85
});

// Insert multiple documents
db.students.insertMany([
  { rollno: 102, name: 'Bob',   marks: 78 },
  { rollno: 103, name: 'Carol', marks: 92 }
]);
```

R — Read (Find)

```
// Find all documents
db.students.find();

// Find with condition
db.students.find({ marks: { $gt: 80 } });

// Find one document
db.students.findOne({ rollno: 101 });

// Find with projection (show only name and marks, hide _id)
db.students.find({}, { name: 1, marks: 1, _id: 0 });
```

U — Update

```
// Update one document
db.students.updateOne(
  { rollno: 101 },
  { $set: { marks: 90 } }
);

// Update all documents matching condition
db.students.updateMany(
  { marks: { $lt: 50 } },
  { $set: { status: 'Fail' } }
);
```

D — Delete

```
// Delete one document
db.students.deleteOne({ rollno: 101 });

// Delete all matching documents
db.students.deleteMany({ marks: { $lt: 35 } });
```

Common Query Operators

- $gt (greater than), $lt (less than), $gte, $lte — comparison
- $eq (equal), $ne (not equal)
- $in (value in array), $nin (not in array)
- $and, $or, $not — logical operators

Q77 & Q78. Map-Reduce and Aggregation Pipeline in MongoDB.

Map-Reduce

- Map-Reduce is a data processing technique for large datasets that involves two steps:
- MAP function: Processes each document and emits (key, value) pairs.
- REDUCE function: Combines all emitted values for the same key into a single result.

Map-Reduce Example — Count total orders per customer

```
var mapFunction = function() {
  emit(this.customer_id, this.amount);  // key=customer, value=amount
};

var reduceFunction = function(key, values) {
  return Array.sum(values);  // sum all amounts for same customer
};

db.orders.mapReduce(
  mapFunction,
  reduceFunction,
  { out: 'customer_totals' }  // output collection
);
```

Aggregation Pipeline

- Aggregation Pipeline is a modern, faster, and more powerful alternative to Map-Reduce.
- Documents are processed through a PIPELINE of STAGES — each stage transforms the data.
- Output of one stage becomes input of the next.

Common Aggregation Stages

- $match — filter documents (like WHERE in SQL)
- $group — group documents and aggregate (like GROUP BY + aggregate functions)
- $sort — sort documents
- $project — select/reshape fields (like SELECT)
- $limit — limit number of results
- $lookup — join with another collection (like SQL JOIN)
- $unwind — deconstruct an array field into individual documents

Aggregation Example — Total marks per branch

```
db.students.aggregate([
  { $match: { marks: { $gte: 0 } } },  // filter valid marks
  {
    $group: {
      _id: '$branch',                   // group by branch
      totalMarks:  { $sum: '$marks' },  // sum of marks
      avgMarks:    { $avg: '$marks' },  // average marks
      studentCount:{ $sum: 1 }          // count students
    }
  },
  { $sort: { avgMarks: -1 } }           // sort by avg descending
]);
```

Q79. Document-Based and Key-Value Data Models of NoSQL.

Key-Value Data Model

- Stores data as unique KEY → VALUE pairs.
- The key is a unique identifier; the value can be any data type (string, number, JSON, binary).
- Very simple model — no schema, no relationships.
- Extremely fast for single-key lookups (O(1) access time).
- Example: Redis — used to cache database query results, store session data.

```
SET user:101 '{name: Alice, age: 25}'
GET user:101  → '{name: Alice, age: 25}'
```

Document Data Model

- Stores data as self-describing DOCUMENTS (usually JSON or BSON).
- Each document can have a different structure — no fixed schema.
- Documents are grouped into COLLECTIONS.
- Supports nested objects and arrays within a single document.
- Example: MongoDB — used for user profiles, product catalogs.

```
{
  'product_id': 'P001',
  'name': 'Laptop',
  'price': 45000,
  'specs': {
    'ram': '16GB',
    'storage': '512GB SSD'
  },
  'tags': ['electronics', 'computer', 'portable']
}
```

Key Difference

- Key-Value: Values are opaque to the database — no queries based on value content.
- Document: Values are structured — you can query any field inside the document.

Q80 & Q81. Structured, Semi-Structured, and Unstructured Data.

1. Structured Data

- Structured data is HIGHLY ORGANIZED data that conforms to a predefined schema or format.
- It can be stored in rows and columns of a relational database.
- Easy to search, sort, and analyze using SQL.
- Represents only about 20% of total data in the world.
- Examples: Bank transaction records, employee salary tables, student marks databases, stock prices, inventory records.

```
Table: Employee(emp_id, name, salary, department, doj)
Row: 101 | Alice | 50000 | HR | 2020-01-15
```

2. Semi-Structured Data

- Semi-structured data does NOT conform to a strict tabular structure but has SOME organizational markers like tags, keys, or separators.
- The structure is flexible — different records can have different attributes.
- Can be stored in NoSQL databases (document stores).
- Examples: XML files, JSON files, HTML web pages, CSV files (loosely structured), email (headers + body), log files.

```
{
  'student_id': 101,
  'name': 'Alice',
  'subjects': ['Math', 'Science'],
  'address': { 'city': 'Mumbai' }
}
```

3. Unstructured Data

- Unstructured data has NO predefined format, schema, or structure.
- It cannot be stored in traditional relational tables in its raw form.
- Represents about 80% of all data generated in the world.
- Requires special tools (ML, NLP, image recognition) to extract meaning.
- Examples: Photos, videos, audio files, social media posts, text documents, PDFs, emails (body content), WhatsApp messages, IoT sensor raw data.
- Stored in: Object storage (Amazon S3, Google Cloud Storage), NoSQL databases, distributed file systems (HDFS).

Comparison Table

Feature	Structured	Semi-Structured	Unstructured
Format	Fixed rows and columns	Tags/markers present	No predefined format
Schema	Predefined, rigid	Flexible	None
Storage	RDBMS (MySQL, Oracle)	NoSQL (MongoDB, XML DB)	File systems, Object stores
Searchability	Easy with SQL	Moderate with JSON/XML queries	Requires AI/ML/NLP

Examples	Bank records, student tables	JSON, XML, HTML, CSV	Images, videos, social posts
Data Volume	Small to medium	Medium to large	Very large (big data)

— End of DBMS Complete Question Bank Answers —

All 81 Questions Covered | Unit I to Unit V