DBMS

INDEX

Important topics

DBMS Introduction

1. What is a Database and a DBMS?

A **Database Management System (DBMS)** is software that enables users to create, manipulate, and administer databases. Allows secure data storage and retrieval it quickly, modify it, and add new data whenever needed.
Example: A Picture a college library: books are arranged shelf-by-shelf by subject, so you can walk straight to the "C-Programming" rack instead of digging through every pile. The library catalog system functions like a DBMS index; they make searching easy.

2. Why do we need a DBMS?

A DBMS ensures data consistency, handles concurrent access by multiple users, enforces security policies, and provides recovery mechanisms (e.g., backups) to restore data after failures.
Ex. If every college department kept its own Excel file, the same student's name, phone, and address would be copied again and again. One day Admissions might update the student's address, but the Fees sheet might not, so the data no longer matches.

3. Main Advantages (with quick definitions + everyday examples)

- **Security** - Decide exactly who can view or change which data.
  Example: In our college system, a fee-clerk logs in and sees only names and fee status, while the principal's login also reveals marks. No one else can quietly peek at everything.
- **No Extra Copies** - Minimizes redundancy by storing data in a structured way, though controlled duplication (e.g., indexes) may exist for efficiency.
  Example: A student's address lies in a single "Student" table; Admissions, Exams, and Fees all read that copy. Because there is only one version, departments always access consistent and up-to-date data.
- **Accuracy (ACID)** - Guarantee that every update is all-or-nothing and always valid.
  Ensures ACID properties:
  - **Atomicity:** Transactions are all-or-nothing.
  - **Consistency:** Data remains valid after updates.
  - **Isolation:** Concurrent transactions don't interfere.
  - **Durability:** Committed changes survive crashes.

  Example: When the library issues a book, both "Issued = Yes" and "Available Copies = Total − 1" are written together. If power fails mid-update, the DBMS rolls the whole action back, leaving no half-done record.
- **Speed (Indexing)** - B-trees (O(log n) search) for ranges, Hash indexes (O(1)) for exact matches
  Example: The librarian types a student ID, and the index lets the system jump straight to that row instead of scanning thousands of rows—so the result appears in a split second.
- **Growth (Scalability)** - Supports scalability by allowing vertical (more resources) or horizontal (more servers) expansion with minimal application changes.
  Example: If the college suddenly admits five-times more students or hires extra teachers, the DBMS can simply attach more storage or compute power; existing queries keep working with no code rewrite.

4. Popular Types of Databases with Easy Examples

Type	Definition	Use Case	Real-life picture
Relational (SQL)	Data stored in tables with defined schemas, linked via relationships (e.g., primary / foreign keys), ensuring ACID properties.	Highly structured tasks.	UPI transactions: Each transfer is recorded atomically with debit/credit as a single transaction.
Document	Schema-flexible documents (JSON/BSON) with optional field validation	Flexible menus or profiles.	Swiggy menus: Each restaurant's JSON document can include unique fields (e.g., 'spice level') without schema changes.
Key-Value	Key-Value stores use distributed hash tables with collision resolution	Ultra-fast look-ups.	Session storage: Flipkart uses key-value stores for temporary user sessions (e.g., browsing history), not transactional carts.
Graph	Saves nodes (people) and edges (links).	Connections and networks.	Fraud detection: Identify networks of accounts connected via shared devices/IPs (e.g., a ring of fake profiles).
Time-Series	Optimized for storing and querying time-stamped data (e.g., IoT sensor logs, stock ticks) with efficient time-range scans.	Sensors and stock prices.	IMD weather station stores temperature every minute to spot heat waves.
Object-Oriented	Stores software objects unchanged.	Embedded or niche apps.	A medical device saves each sensor object with its current status, ready to reload.

- Choose your database by the problem, not by popularity.
- SQL is a language; many non-relational databases now let you query in SQL-like style too.

Self Search:

Why might Flipkart use a key-value store for **user sessions**, while UPI requires a relational database?

DBMS Architecture

1. What is the Architecture of DBMS?

The architecture of a database system describes **where each software component resides and how those components communicate**. A clear understanding of this layout enables informed decisions regarding performance, security, and scalability.

2. One, Two, and Three-Tier Architectures

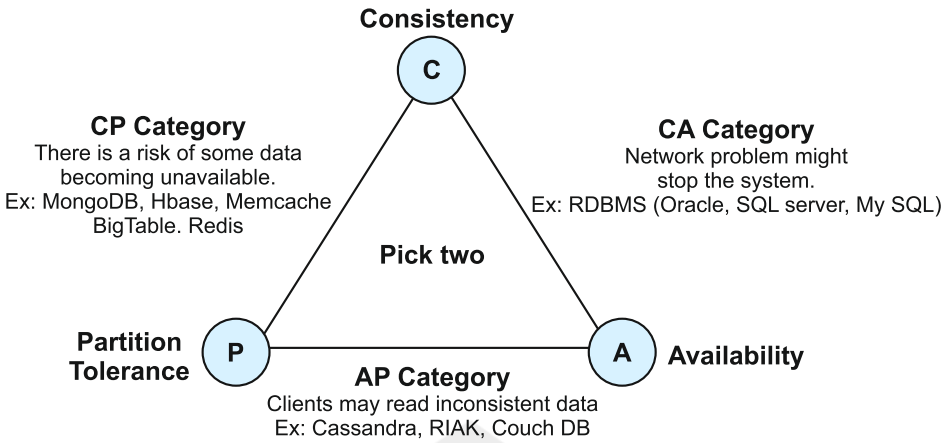
Architecture	Design	Example	Key Observations
One-Tier	The user interface and the database engine run on the same machine.	A student runs a Python script with SQLite on a single machine, where the app and database share the same process.	Setup is instantaneous, yet the solution remains single-user and consumes significant local resources.
Two-Tier	A client application communicates directly with a dedicated database server.	All desktop PCs in the computer laboratory run a Java-based GUI that connects straight to a central MySQL server to store laboratory grades.	Multiple users obtain better performance than in one-tier; however, clients communicate directly with the DB server, often through middleware (e.g., ODBC / JDBC) that manages connections.
Three-Tier	A client application communicates with an intermediary application server, which in turn queries the database server.	The mobile app sends requests to a REST API (e.g., Node.js), which handles authentication and queries PostgreSQL, ensuring stateless scalability.	Business logic and security are centralized in the middle tier, the database remains shielded, and clients stay lightweight.

3. Distributed Databases

A **distributed database** stores segments of data on two or more geographically separated servers, yet presents them as a single logical database.
Example: An educational trust operating campuses in Delhi, Mumbai, and Bengaluru maintains each city's attendance data on a local server to ensure rapid retrieval. The system synchronizes records asynchronously (eventual consistency) or via distributed transactions (strong consistency), depending on requirements.

4. CAP Theorem

In distributed systems, Partition Tolerance is mandatory; trade-off between C and A
Consistency (C) – Every node reflects identical data at any given moment.
Availability (A) – Every request receives a timely response, without guarantee that the data are the most recent.
Partition Tolerance (P) – The system continues operating even when communication between nodes is disrupted.



Scenario	Two Properties being followed	Explanation
UPI-based mess-wallet payments	**Consistency + Availability** (e.g., blocking transactions during partitions to prevent double-spending).	Monetary transfers must never diverge, even if slight delays occur during network disturbances. Temporary unavailability may be tolerated.
Digital notice-board website	**Availability + Partition Tolerance**	Displaying an older notice is preferable to complete inaccessibility during Wi-Fi outages.
Intra-college coding-contest scoreboard	**Availability + Consistency**	Network partitions are rare within the LAN, thus participants expect scores that are simultaneously current and instantly visible.

Self Search:

What 2 properties among CAP are guaranteed in case of financial applications like paytm/paypal, etc.?

Data Abstraction

1. What is Data Abstraction?

Data abstraction hides implementation details (e.g., storage formats, indexing) from users and applications, exposing only relevant data structures and operations. By separating **how** data is stored from **what** the data represents, a DBMS improves security, flexibility, and maintainability.

2. Three Levels of Abstraction

Level	Definition	Example
Physical Level	Describes the exact location in bytes on disk—file structures, indexing methods, and data blocks.	The database team decides that attendance records are stored as 8KB blocks on SSD, with RAID-1 mirroring for redundancy. A B-tree index (logical level) maps Roll No. to block addresses. Students never see these low-level details.
Logical Level	Presents the entire database structure in terms of tables, columns, data types, and relationships.	A 'Students' table defines Roll No. (PRIMARY KEY), Name (VARCHAR), and a foreign key to 'Courses'. Constraints enforce referential integrity. This design is what faculty refer to when writing SQL queries.
View Level	The view level presents customized logical data subsets.	The warden's view filters 'Students' to show Roll No., Name, and a computed FeeStatus field derived from payment dates, whereas the examination branch sees Roll No., Name, and Semester Marks.

Key Point: Each higher level is insulated from changes made at the level beneath it.

3. Data Independence

Type	Definition	Example
Logical Independence	The ability to alter the logical schema without rewriting application programs.	The college restructures the "Students" table by splitting Address into Street, City, and Pin Code. Neither the hostel fee management system nor the attendance app requires code changes because their views are automatically adjusted.
Physical Independence	The capacity to modify physical storage details without affecting the logical schema.	The college IT department moves attendance data from HDDs to faster SSDs and adds an extra index on Date. No SQL query written by faculty or students needs alteration.

4. **Essential Takeaways**

- **Physical (lowest) → Logical → View (highest abstraction)** hierarchy; knowledge flows upward, while changes should ideally flow downward.
- Achieving full data independence is challenging but even partial success greatly reduces maintenance costs.
- In daily student life, the view level is what mobile apps and portals expose, the logical level is what developers design, and the physical level remains hidden in the server room.

Self Search:

Explore the evolution of Instagram's logical and physical data independence from a photo sharing platform to stories, reels, and DM's.

Types of Data Models

1. What are Data Models in DBMS?

A data model supplies the **formal grammar** for describing how facts are organised inside a database. Choosing the correct model influences performance, flexibility, and ease of maintenance.

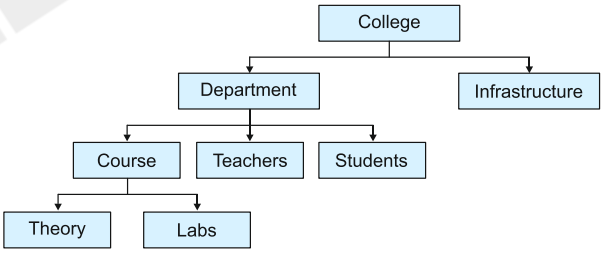
2. At-a-Glance Comparison of various Data Models

Model	Core Structure	Ideal For
Hierarchical	Parent → child tree	One-to-many chains
Network	Records linked by multiple parent–child sets (many-to-many)	Complex associations
Relational	Tables, rows, columns; joined by keys	General-purpose data
Object-Oriented	Persistent programming objects	Highly complex, behaviour-rich data
Entity–Relationship (ER)	Diagrammatic blueprint of entities and relationships	Conceptual design stage

3. Model Descriptions with Examples

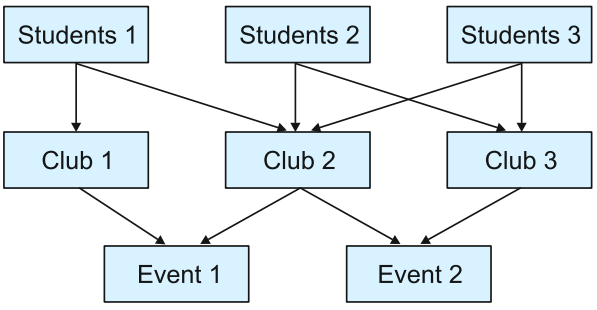
- **Hierarchical Model**

  A hierarchical model arranges records in a strict top-down tree; each child has exactly one parent.
  Example: The university's administrative system stores data as a chain: **University → College → Department → Course → Student**. A query for all students in "CSE, 2022 Batch" simply follows the path from root to leaves.



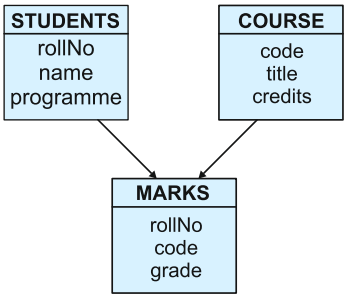
- **Network Model**

  The network model expands the tree by allowing a record to have multiple parents, thereby supporting many-to-many links.
  Example: A **Student** can be linked to several **Clubs**, while each **Club** links back to many **Students** and multiple **Events**. Retrieving every student attending a coding marathon involves traversing these linked sets.



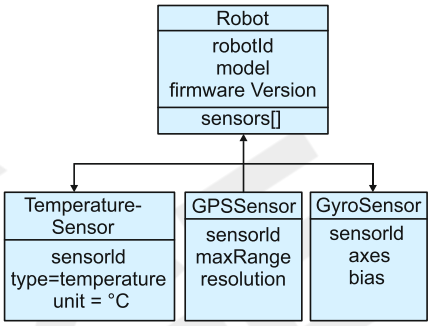
- **Relational Model**

  The relational model stores data in two-dimensional tables and enforces relationships with primary and foreign keys. Queries use structured query language (SQL).
  Example: The college database maintains tables **STUDENT (roll No, name, programme)**, **COURSE (code, title, credits)**, and **MARKS (roll No, code, grade)**. Joins enable the examination section to publish marksheets.



- **Object-Oriented Model**

  This model persists classes exactly as defined in an object-oriented language, keeping both attributes and methods.
  Example: In a robotics research database, each **Robot** object stores arrays of **Sensor** objects. When technicians fetch a robot record, they can immediately get the object associated with it.



- **Entity–Relationship (ER) Model**

  The ER model is a high-level, picture-oriented method for mapping reality into entities, attributes, and relationships before physical implementation.
  Example: During the design phase, database engineers draw an ER diagram containing entities **Student**, **Course**, **Instructor**, and relationships **ENROLLED_IN** and **TAUGHT_BY**. This conceptual map later guides table creation in a relational database.

ER Model

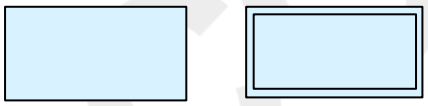
1. Purpose of the ER Model

The ER model is a conceptual-level blueprint (not implementation-specific) that defines entities, attributes, and relationships, bridging real-world scenarios to database design.

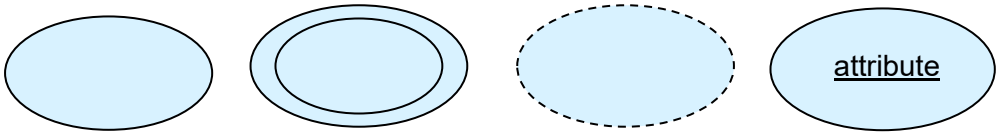
2. Terminologies

Term	Definition	Example
Entity	A **strong entity** (e.g., Student) has a unique key; a **weak entity** (e.g., Dependent) depends on a strong entity for identification.	An individual Student or a specific instance of a Course.
Attribute	A property that describes an entity.	**roll Number** (key attribute), **student Name** (simple), **course Credits** (derived from hours).
Entity Set	A collection of similar entities.	The entire set of **Students** enrolled in the university.
Relationship	A logical association among entities.	A **Student** ENROLLED_IN a **Course**.
Relationship Set	A collection of relationship instances among entities, such as student-course enrolments.	All instances of a binary relationship (e.g., Student ↔ Course) or ternary (e.g., Student ↔ Course ↔ Instructor).

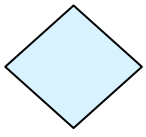
3. ER Diagram Notation (Quick Reference)



Rectangles → Entity sets (strong entity in rectangle and weak entity in double rectangle). Weak entities use double rectangles with a dashed underline for partial keys (e.g., bed Number in Occupancy) and a double diamond for identifying relationships (e.g., Hostel Room → Occupancy).



Ellipses → Attributes (Double ellipse → multivalued attribute (e.g., phone numbers), Dashed ellipse → derived attribute (e.g., age), Underlined name → primary key).



Diamonds → Relationship sets.

Lines → Connections, cardinality marks appear on these lines.
Cardinality is marked as:

- 1:1: |—| (e.g., Student ↔ Library Card).
- 1: N: |—< (e.g., Department → Students).
- M: N: >—< (e.g., Students ↔ Courses).

4. Cardinality and Participation

Cardinality Symbol	Definition	Example
1 : 1	Each entity links to exactly one on the other side.	1:1 with total participation for Library Card (must belong to a student), but partial for Students (may not have a card if lost).
1 : N	One entity may relate to many on the other side.	One **Instructor** teaches many **Courses**, but each course has exactly one instructor for a semester.
M : N	Many entities relate to many others.	M: N in ER becomes an **associative table** (e.g., ENROLLMENT) with composite key (StudentID, CourseID).

Participation
Total: Every entity must participate (e.g., every Library Card must be assigned to a student).
Partial: Partial participation means not all entities in the entity set participate in the relationship (e.g., not every student joins a Club).

5. Weak Entities

A **weak entity** lacks a primary key of its own and depends on a **strong owner entity** plus a partial key.
Example:
OWNER entity set: Hostel Room (room No).
WEAK entity set: Occupancy uses dashed underline for partial key (bed Number) and double diamond for identifying relationship with Hostel Room.
Associated occupancy records should be automatically deleted when the owner entity is deleted (cascading delete).

6. Generalization (ISA Hierarchies)

Generalization creates a superclass–subclass hierarchy, capturing shared attributes in the superclass.
Example:

```
          [PERSON]
           /     \
[STUDENT]       [INSTRUCTOR]
```

Subclasses **inherit** superclass attributes (e.g., STUDENT gets name from PERSON).
STUDENT adds programme, roll No; INSTRUCTOR adds department, salary.
Queries requiring all people (e.g., campus email list) consult only the superclass, while programme-specific queries use **STUDENT** alone.

7. Specialization

Specialization divides a superclass into exclusive or overlapping subclasses. It may be total (every entity must belong to a subclass) or partial (some entities may not belong to any subclass).
Example:

```
[STUDENT]       [INSTRUCTOR]
        \          /
         [PERSON]
```

PERSON holds common attributes (name, address).
STUDENT has programme, roll No; INSTRUCTOR has department, salary.
Queries requiring all people (e.g., campus email list) consult only the PERSON, while programme-specific queries use STUDENT alone.

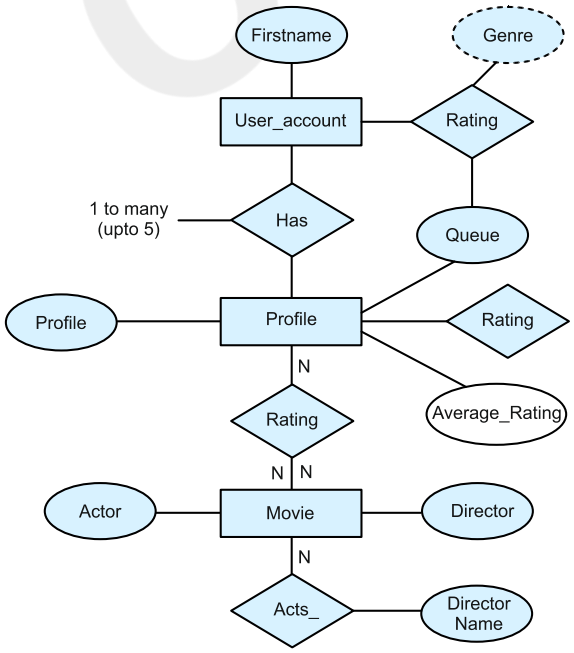
- Generalization combines multiple lower-level entities into a higher-level entity by collecting their common attributes (e.g., STUDENT and INSTRUCTOR into PERSON).
- Specialization pushes **distinct features downward** ("one becomes many").

8. Aggregation

Aggregation means combining smaller parts into one big meaningful unit, just like making a full project using different modules. In an ER diagram, aggregation is used when a relationship itself needs to be connected to something else. We group the relationship + its entities and treat them as one single unit.

Example:
Exam Cell wants to track which projects are submitted. But to submit a project, we need Student, Project and a Faculty Guide. These 3 can be connected by a relationship: WORKS_ON (Student, Project, Faculty Guide). Now, instead of linking all 3 separately to Submission Status, we group them like this:
Wrap WORKS_ON and its entities in a **dashed box** labelled Project Submission, then link to Submission Status.

9. ER Diagram for Netflix



i. **User_Account**
- **Attributes:** USER_ID (Primary Key), Firstname, Lastname, Credit_Card_No.
  This is like your Netflix account. One account per user.
- **Relationships:**
  - User_Account ─ (1: 5) ─ PROFILE with crow's foot at PROFILE.
  - Double line from PROFILE to HAS relationship.

ii. **Profile**
- **Attribute:** PROFILE_ID (Primary Key), PROFILE_NAME, CREATION_DATE
  Each account can have up to 5 profiles (like for you, your dad, your sibling).
- **Relationships:**
  - USER_ACCOUNT ─ (1: 5) ─ PROFILE (one account owns 1-5 profiles).
  - Has preferences for GENRE (User Genre).
  - Has QUEUE and Rental History with movies.

iii. **Genre**
- **Attribute:** GENRE_ID (Primary Key), GENRE_NAME
  Types of movies (like action, comedy, horror, etc.).
- **Relationships:**
  - PROFILE ─ (M: N) ─ GENRE through USER_PREFERENCE (associative entity with weight/priority).
  - Linked to MOVIE (Movie Genre).

iv. **Movie**
- **Attributes:** MOVIE_ID (Primary Key), MOVIENAME, YEAR, PRODUCER, AVERAGE RATING, DURATION (minutes), STREAMING_QUALITY (HD/4K), LICENSE_EXPIRY (date)
  Each movie has these details stored.
- **Relationships:**
  - Appears in QUEUE and Rental History of a PROFILE.
  - Linked with GENRE through Movie Genre.
  - MOVIE ─ (M:N) ─ ACTORS via STARRED_BY (with ROLE attribute).

v. **Queue (Relationship)**
- **Attribute:** RANKING
  A profile's to-watch list with order (e.g., 1st movie to watch, 2nd…).
- **Connects:** PROFILE ↔ MOVIE (1 profile ↔ many movies).

vi. **Rental History (Relationship)**
- **Attribute:** RATING (1 to 5), WATCH_DATE, DEVICE_TYPE (mobile/TV)
  What movies a profile has watched and rated.
- **Connects:** PROFILE ↔ MOVIE.

vii. **Actors**
- **Attributes:** ACTOR_ID (Primary Key), FIRSTNAME, LASTNAME
  ACTOR ─ (1: N) ─ AWARD (optional participation).
- Connected to MOVIE with Starred_by.

viii. **Starred by (Relationship)**
- **Attributes:** CHARACTER_NAME (e.g., 'Tony Stark').
  Shows which actors acted in which movies.
- **Connects:** ACTORS ↔ MOVIE.

Relational Model

1. Relational Model in DBMS

Every relational database-whether it runs your college ERP or the canteen app-follows the same four-part "grammar." If you understand these parts and the rules that protect them, you can design tables that never lose marks, fees, or attendance records.

2. Fundamental Building Blocks

Building Block	Definition	Example
Domain	A set of valid values with constraints (e.g., RollNo INT CHECK (RollNo BETWEEN 21000000 AND 21999999)).	Roll Number Domain = "any 8-digit number from 21000000 to 21999999."
Attribute	A single column in a table; its values must come from one domain.	RollNo INT, Name VARCHAR(30), Programme CHAR(10), MobileNo CHAR(10).
Tuple	One complete row that talks about exactly **one** real-world item.	Atomic values only (e.g., split Name into FirstName, LastName for 1NF compliance).
Relation	The whole table-many tuples under the same set of columns.	The STUDENT table that contains every student's detail this year.

Keep this ladder in mind: **Domain** → **Attribute** → **Tuple** → **Relation**.

3. Relational Schema vs. Instance

- **Schema (also called "intention")**
  A stable structure defining tables, attributes, domains, and constraints (e.g., STUDENT (RollNo INT PK, Name VARCHAR (30)). It lists the table name, each attribute, its domain, and any rules (like which column is the primary key).
  Example: STUDENT (roll_no INT PRIMARY KEY, name VARCHAR(30), programme CHAR(10), mobile_no VARCHAR(10))
- **Instance (also called "extension")**
  The actual data right now. It is just the collection of rows currently sitting in the table. Tomorrow's instance may have new students or fewer if someone drops out.

Think of the schema as the empty worksheet template and the instance as the filled worksheet after everyone's marks are entered.

Sample Relation:

Roll_No	Name	Programme	Mobile_No
22104567	Parikshit Sharma	BTech CSE	9876543210
22104612	Rajat Mehta	BTech ECE	9123456780
22104755	Sakshi Bansal	MBA	9001122334

This table represents the **STUDENT** relation in the database. Each row is a tuple, and each column is an attribute with its own domain.

4. Integrity Constraints

- **Key Constraint**
  Rule: Every table must have at least one candidate key — a column (or set of columns) whose values uniquely identify each row and never repeat.
  Example: Roll No is unique for every student; if your script tries to insert the same roll number twice, the DBMS blocks it. Keys can be single-column (RollNo) or composite (StudentID, CourseID).
- **Entity Integrity**
  Rule: A primary-key column cannot contain NULL values.
  Example: You cannot add a student record with an empty roll number; otherwise, nobody could tell who that row belongs to.
- **Referential Integrity**
  Rule: A foreign key must match an existing primary key value in the referenced table. If NULL is allowed, it means the relationship is optional.
  Example: The MARKS table has a foreign key column, Roll No, which references the primary key (Roll No) in the STUDENT table. If you upload marks for Roll No 22109999 but that student does not exist in STUDENT, the DBMS rejects the upload to maintain referential integrity.

Types of Keys

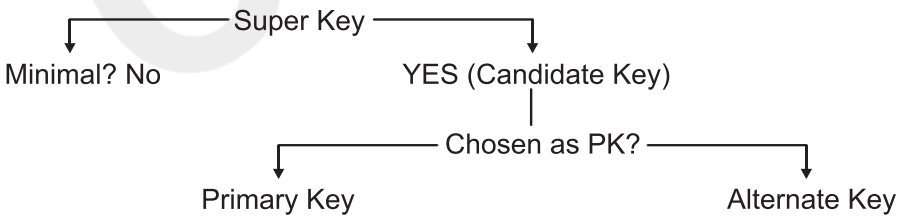
1. What are keys?

A **key** is a set of one or more columns that lets the DBMS (and you) pick out a single, exact row from a table. Without keys, duplicate data would creep in and your queries would return the wrong student, the wrong mark, or even both.

2. Quick-look Table of Key Types

Key Type	Definition	Example
Super Key	A column-set (possibly with redundant columns) that uniquely identifies rows.	{ rollNo, mobileNo }, { rollNo }, { email } — each super key is unique.
Candidate Key	A minimal super key: remove any column and it stops being unique.	{ rollNo } or { email }. Both stand alone and are minimal.
Primary Key	The single candidate key the DBA chooses as the official row identifier.	College chooses rollNo as the primary key for **STUDENT**.
Alternate Key	Any remaining candidate key not chosen as the primary key.	Alternate keys are often indexed for faster lookups (e.g., searching by email).
Composite Key	A key built from two or more columns.	In the MARKS table, a combination of { rollNo, courseCode } forms a composite key, uniquely identifying each mark entry for a student and course.
Foreign Key	A column set that points to the primary (or alternate) key of another table.	MARKS.rollNo → STUDENT.rollNo ; MARKS.courseCode → COURSE.code.

Mind Map



Remember: Every primary key is a candidate key, every candidate key is a super key, but not the other way round.

⭐ 3. Keys Examples in Detail

- **Primary Key**

Roll No (PK)	name	programme	email
22104567	Ananya Sharma	BTech CSE	ananya.sharma@college.edu.in

Example
Table: STUDENT
Roll No never repeats and is never NULL, so the DBMS can efficiently identify a unique student.

- **Composite Key**

Roll No (FK)	Course Code (FK)	grade
22104567	CS101	8.5

Example
Table: **MARKS**
Roll No alone is not unique (a student has marks in many subjects).
Course Code alone is not unique (many students take CS101).
Together they are unique, forming a composite key.

- **Foreign Key**
If someone tries to insert roll No = 22109999 in **MARKS** but that Roll No is missing from STUDENT, the foreign-key check fails and the insert is refused.

4. Checks for Keys in Exams & Interviews

- **Minimality test:** Drop one column from the key; if the remaining columns no longer uniquely identify the row, then the key was minimal.
- **Null rule:** In composite PKs, no column can be NULL (unlike foreign keys); a foreign key may be NULL if "if the relationship is optional" is allowed.
- **Composite vs. Super:**
Composite Key: A **minimal** multi-column candidate key.
Super Key: **Any** unique column-set (may include non-key columns).

Normalisation

1. What is normalisation?

Normalisation decomposes large, unstructured tables into smaller, logically related tables to minimize redundancy and dependency. This removes update anomalies, improves consistency, and ensures data integrity.

2. Functional Dependency (FD)

A **functional dependency** X → Y means: If two tuples have the same value for attribute set X, they must have the same value for attribute set Y. In this case, Y is said to be functionally dependent on X.
Example:
Roll No → Student Name
(the same roll number can never belong to two different names).

Types of Functional Dependencies (FDs)

Type of Dependency	Mathematical Relation	Example
Full Functional Dependency	X→Y where Y depends on all attributes in X, i.e., for any A⊂X, A→Y.	Like needing both student ID + exam paper code to find a grade (neither alone suffices): To know your Grade in a subject, you need both your Roll No. and the Course ID (e.g., CS101). Neither alone is enough.
Partial Functional Dependency	X→Y where X = AZ (composite key) and A→Y (A is a proper subset of X).	Partial FDs violate 2NF ⇒ split tables (e.g., move Student Name to a STUDENT table).: Your Student Name depends only on your Roll No., not the Course ID. (Roll No. → Student Name)
Trivial Functional Dependency	X→Y where Y ⊆ X.	Roll No., Name → Name: If you know the student's Roll No. and Name, you obviously know their Name.
Non-trivial Functional Dependency	X→Y where Y ⊄ X.	Roll No. → Dept_Name: Your Department Name is determined by your Roll No., but Dept Name isn't part of the Roll No. itself.
Transitive Dependency	A functional dependency A → C is said to be transitive if it can be derived from A → B and B → C.	Transitive FDs cause update anomalies: Changing a department's building requires updating all student records. You can find the building from your roll number, but it goes through your department.
Multivalued Dependency	A → {B, C} where B and C are independent.	Roll No →→ Hobbies, Roll No →→ Sports (Multivalued Dependencies): A student's Roll No can determine multiple hobbies and multiple sports independently.

3. Four Key Normal Forms

Normal Form	Simple rule	Quick college illustration
1 NF	Every cell holds one atomic (indivisible) value; no repeating groups.	1NF enforces atomic values: Each cell contains one indivisible datum (e.g., no lists/arrays). Store each course in a **separate row**, not "CS101, MA102" in one cell.
2 NF	If (Roll No, Course Code) is PK, Student Name (dependent only on Roll No) must be moved.	If the key is {roll No, course Code}, columns like student Name (depends only on roll No) must move to another table.
3 NF	2 NF **and** no *transitive* FD (A → B → C). Every non-prime attribute must depend only on a super key.	In a COURSE table, course Code → instructor ID and instructor ID → instructor Name create a chain, move instructor details out.
BCNF	For **every** FD X → Y, X must be a super key. Stronger version of 3 NF.	If instructor ID → course Code also holds, you need a separate TEACHES table so that both sides become keys. TEACHES(Instructor ID PK, Course Code PK) to satisfy BCNF.

4. Lossless Join vs. Dependency Preservation

- **Lossless Join -** After decomposing a table, joining the pieces must **exactly** recreate the original rows-no extra, no missing.
- **Dependency Preservation -** All original FDs can still be enforced **without** joining the tables back together.

Most textbook decompositions aim for **both**, sometimes BCNF forces you to choose lossless join over perfect preservation.

5. Walk-through Example - From 1 NF to BCNF

- **Un-normalised form (UNF)**
  Consider a single spreadsheet used by a hurried faculty member:

Roll No	Student Name	programme	Course List	Grade List
22104567	Ananya Sharma	BTech CSE	CS101, MA102	8.5, 7.0
22104612	Rajat Mehta	BTech ECE	EC101	8.0
22104755	Farah Khan	MBA	MG201, MG202, MG203	7.5, 8.2, 7.9

Problems: multi-valued cells, hard to insert a new course, risky to update a single grade. Ananya's, Farah's row stores multiple values in single cells-clearly not atomic.

- **First Normal Form (1 NF)**
  Rule: every cell holds one value. Split repeating lists into individual rows.

Roll No	Student Name	programme	Course Code	grade
22104567	Ananya Sharma	BTech CSE	CS101	8.5
22104567	Ananya Sharma	BTech CSE	MA102	7.0
22104612	Rajat Mehta	BTech ECE	EC101	8.0
22104755	Farah Khan	MBA	MG201	7.5
22104755	Farah Khan	MBA	MG202	8.2
22104755	Farah Khan	MBA	MG203	7.9

  Problem: Still redundant, the same student data repeats on every course row.
- **Find the FDs**
  FD1: roll No → student Name, programme.
  FD2: course Code → course Name, credits, instructor ID.
  FD3: instructor ID → instructor Name.
  FD4: (roll No, course Code) → grade.

- **Second Normal Form (2 NF)**
  (roll No, course Code) is the composite key. Columns that depend only on part of it move out.
  - STUDENT (roll No PK, student Name, programme) ← FD1.
  - COURSE (course Code PK, course Name, credits, instructor ID) ← FD2.
  - RESULT (roll No FK, course Code FK, grade) ← FD4.
  - RESULT table: Roll No FK → STUDENT, Course Code FK → COURSE.

  Rule: remove columns that depend on only part of a composite key (roll No, course Code) and create separate tables.

STUDENT

Roll No	Student Name	programme
22104567	Ananya Sharma	BTech CSE
22104612	Rajat Mehta	BTech ECE
22104755	Farah Khan	MBA

COURSE

Course Code	Course Name	credits	Instructor ID
CS101	Intro to CS	4	I01
MA102	Calculus I	3	I02
EC101	Basic Electronics	4	I03
MG201	Management 101	3	I04

INSTRUCTOR

Instructor ID	Instructor Name
I01	Dr Rao
I02	Prof Gupta
I03	Dr Khan
I04	Dr Sen

RESULT

Roll No	Course Code	grade
22104567	CS101	8.5
22104567	MA102	7.0
22104612	EC101	8.0
22104755	MG201	7.5

After creating the INSTRUCTOR table, every non-key column in each table depends on the whole primary key.

- **Third Normal Form (3 NF)**
  FD3 is transitive because instructor ID → instructor Name and both attributes are present in the COURSE table. Hence, instructor Name is transitively dependent on course Code via instructor ID.
  Create a new table:
  - **INSTRUCTOR** (instructor ID PK, instructor Name).
  - **Update COURSE:** remove instructor Name, keep instructor ID.

  Rule: remove transitive dependencies. Instructor ID → instructor Name lies inside the COURSE table.

INSTRUCTOR

Instructor ID	Instructor Name
I01	Dr Rao
I02	Prof Gupta
I03	Dr Khan

COURSE now drops instructor Name but keeps instructor ID.

Course Code	Course Name	credits	Instructor ID
CS101	Intro to CS	4	I01
MA102	Calculus I	3	I02
EC101	Basic Electronics	4	I03

All tables are in 3 NF.

- **Boyce-Codd Normal Form (BCNF)**
  Rule: in every functional dependency X → Y, X must itself be a candidate key.
  Assume one instructor can teach many courses (common in college). Since all functional dependencies in the current schema have left sides that are candidate keys, the schema satisfies BCNF.
  If the college instead enforced "an instructor teaches exactly one course," we would break BCNF. To resolve the BCNF violation where each instructor teaches exactly one course i.e., instructor ID → course Code, we split the relation by creating an associative table TEACHES (instructor ID, course Code), ensuring both resulting tables satisfy BCNF.

  Final BCNF set (for the common 'one instructor – many courses' rule):
  - **STUDENT** (roll No PK, student Name, programme).
  - **INSTRUCTOR** (instructor ID PK, instructor Name).
  - **COURSE** (course Code PK, course Name, credits, instructor ID FK).
  - **RESULT** (roll No FK, course Code FK, grade, PK = roll No + course Code).

  All joins are lossless, and the original dependencies (grade tied to student + course; instructor tied to course) are preserved without stitching tables back together during updates.

Denormalisation

1. What is Denormalisation?

Denormalization intentionally introduces controlled redundancy by:

1. Combining normalized tables vertically (fewer tables)
2. Adding derived columns horizontally (pre-computed values)
3. Creating materialized views (persisted query results)
   Always maintains normalized source of truth.

You accept some data repetition so that common queries avoid expensive joins.

2. When and Why do we Denormalise?

- **Heavy read, light write**
  Example: The placement-cell dashboard shows current CGPA for every student frequently, but updates only once per semester.
  Denormalisation helps by avoiding joins between **STUDENT and RESULT** tables at every page load.
- **Aggregates needed repeatedly**
  Example: Canteen system prints a monthly mess bill by aggregating daily swipe data; totals are reused the whole month.
  Denormalisation helps by Pre-storing the total saves 30 × daily calculations.
- **Star schema for reports**
  Example: Annual NAAC report fetches student, course, and programme data in one query.
  Denormalization helps by allowing simpler, faster analytics queries.

Common motives: **speed**, **simplified code, analytics convenience, or working** around slow networks.

3. Trade-offs - Performance vs Redundancy

More speed (Denormalized) ⟵ balance point ⟶ Cleaner data (Fully Normalised)

Metric	Normalized	Denormalized	Impact
Read Performance	100ms per join	10ms flat	10x faster
Write Performance	5ms	20ms (plus sync)	4x slower
Storage	100GB	150-300GB	+ 50-200%
Consistency	Immediate	Eventual (batch)	Staleness risk

- **Pros of Denormalization**
  Fewer joins → faster SELECTs.
  Simpler queries for BI dashboards.
  Lower CPU cost on overloaded DB servers.
- **Cons**
  Repeated data eats storage.
  Extra columns can go stale (update anomalies).
  More triggers or batch jobs are needed to keep copies in sync.

Rule of thumb: denormalise **only** after measuring that joins are your real bottleneck, not before.

4. Example - From BCNF to a Denormalised "Result View"

Recall the BCNF design from the previous class:

Table	Key Columns
STUDENT	Roll No PK
COURSE	Course Code PK
INSTRUCTOR	Instructor ID PK
RESULT	Roll No + course Code PK

Daily requirement: the exam branch must list **roll No, name, programme, course Name, credits, instructor Name, grade** for 5,000 students every hour during result season.

4.1 – Pain Point

The report currently joins **STUDENT × RESULT × COURSE × INSTRUCTOR** → if say 20 ms per row × 5,000 = 100 s.

4.2 – Denormalised Schema

Create a new table **RESULT_FLAT**, refreshed nightly:

Roll No	Student Name	programme	Course Code	Course Name	credits	Instructor Name	grade
22104567	Ananya Sharma	BTech CSE	CS101	Intro to CS	4	Dr Rao	8.5
…	…	…	…	…	…	…	…

Redundant fields: student Name, programme, course Name, credits, instructor Name
Result: Hourly report now runs as a single-table scan.

Another Mini Example - Pre-computed Mess Balance

Normalised set
SWIPE (student ID, date, amount) - thousands of rows per month
STUDENT (student ID, name, programme)

Denormalized helper table:
MESS_BALANCE (student ID PK, month, total amount)

Result: The kiosk app reads MESS_BALANCE efficiently instead of summing every swipe in real time.

5. Guidelines for Safe Denormalisation

- **Measure first** – use EXPLAIN, look at actual query times.
- **Target hot paths** only – keep the rest normalised.
- **Automate refresh –** triggers, materialised views, or nightly data syncs to keep copies fresh.
- **Document –** note which fields are duplicated and who owns the "truth."
- **Rollback plan –** ensure you can revert if storage or update costs become excessive.

Transactions & Concurrency Control

1. What exactly is a transaction?

A transaction is a **small, complete job** that the database treats as one package.
Think of it like sending a registered parcel: the courier either delivers the whole parcel or brings it back; they never deliver just part of it.

For eg. When a teacher presses **"Publish Result"** the system must do three things:

i. **Insert** the new grade for every subject.
ii. **Re-calculate** each student's CGPA.
iii. **Set** a flag that says "Result Published = YES".

These three steps **belong together**. If power fails after step 1, steps 2 and 3 never run, so the grade and CGPA would clash. The database therefore cancels step 1 as well, leaving everything exactly as it was.

2. ACID - four promises a transaction must keep

Property	Original Definition	Enhanced Definition	Example Scenario
Atomicity	All-or-nothing execution	Guarantees entire transaction succeeds or fails as a unit, including: - System crashes - Constraint violations - Deadlocks	Power fails during fee payment: • Neither account is updated • Log shows attempted transaction
Consistency	Maintains database rules	Ensures all constraints (PK, FK, CHECK) are valid before and after, even if: - Multiple transactions run concurrently - System fails mid-transaction	Student transfer between departments: • Department quotas preserved • No student exists without a department
Isolation	Transactions don't interfere	Provides isolation levels (Read Uncommitted → Serializable) with precise anomaly prevention	Two teachers grading same paper: • Serializable: One waits • Read Committed: May see intermediate state
Durability	Survives crashes	Committed changes persist via: - Write-ahead logging (WAL) - Synchronous replication - Checksum verification	Server crash after grade submission: • Grades recoverable from WAL • No "lost" submissions

PROS and CONS of ACID properties:
PROS:

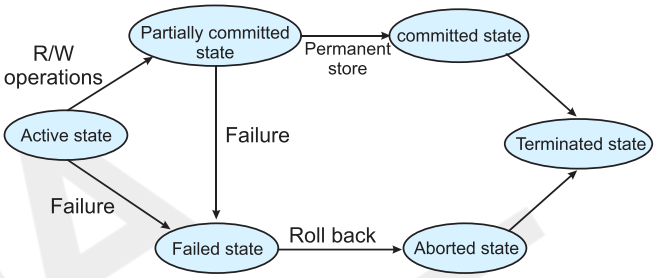
- **Keeps data correct –** no phantom marks or wrong balances.
- **Survives crashes –** quick recovery up to last commit.
- **Supports safe concurrency –** allows multiple users to execute transactions simultaneously without conflict.

CONS:

- **Extra overhead** - logging + locking can slow heavy traffic.
- **Complex to build** - code & storage engine get bigger.
- **Scalability limits** - strict ACID across many servers is tough; needs fancy tech (e.g., distributed consensus).

3. Life-cycle of a transaction

ACTIVE - SQL statements are running.
PARTIALLY COMMITTED - last statement finished; DB is flushing log records.
COMMITTED - success message sent to user; job is permanent.
ABORTED - an error happened; DB UNDOs every change and returns to the original state.



4. Different ways to schedule transactions

- **Serial schedule:** Run Transaction 1 completely, then Transaction 2.
  Benefit: always safe.
  Risk: Increased response time for users due to sequential execution.
- **Interleaved (parallel) schedule**
  Mix the steps: T1-step, T2-step, T1-step, and so on.
  Benefit: better CPU use, faster for many users.
  Risk: need rules so results still look like some serial order.

Serializable schedules – safe interleaving
Goal: To ensure that even with interleaving, the final result is equivalent to some serial execution.

Flavour	Test	Example
Conflict-serializable	Build a "who-uses-same-data" graph; if no cycle → good.	T1 writes CGPA, T2 later reads the same CGPA. Edges show T1 → T2; if graph has no loop, schedule is safe.
View-serializable	Same final **values** and same "first-read" data as a serial run. Superset of conflict - serializable.	Even if ops don't commute, as long as marks and attendance end up identical to some serial run, it passes. Blind writes (write without read) can be OK here.

Recoverability Ladder – Transaction Commit Permissions

- **Recoverable schedule**
  If T2 has read data written by T1, T2 can commit **only after** T1 commits.
  Prevents inconsistencies such as a transaction depending on data that was later rolled back.
- **Cascading (may trigger chain aborts)**
  Reading uncommitted data is allowed.
  If T1 aborts, every T2, T3 that touches T1's dirty writes must also abort → domino effect.
- **Cascade less (ACA)**
  A transaction is allowed to **read only committed data**.
  Zero domino problem, but still lets another transaction **write** over uncommitted data.

- **Strict schedule (gold standard)**
  No one may read or write a value changed by T1 until T1 has committed or aborted.
  Guarantees both no dirty reads and no dirty writes; most real systems get this via **Strict Two-Phase Locking (S2PL).**
- **Non-recoverable schedule – why you should avoid it**
  T2 reads T1's uncommitted value and then commits before T1.
  If T1 aborts later, the database is stuck with bad data from T2; no clean rollback path.
  Exam keyword: **"dirty commit."**

"Serializable decides order, Recoverability decides commit timing, Strict says 'hands off my data till I'm done.'"

5. Isolation levels (from weakest to strongest)

Level	Implementation	Lock Scope	Anomalies Prevented
Read Uncommitted	No read locks	Row-level	None
Read Committed	S-locks on reads	Row-level	Dirty reads
Repeatable Read	S-locks held	Row+index	+ Non-repeatable reads
Serializable	Range locks	Predicate	+ Phantoms

Example:
-- Set per-transaction isolation
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE grades SET score = 85 WHERE student_id = 101; COMMIT;
Stronger isolation ⇒ more safety but extra locking and slower writes.

Picking the right isolation level

Isolation choice	Why to choose	Example
Higher level (e.g., **Serializable**)	Data always stays correct, but more waiting because locks are held longer	Only one student is served at a time to avoid data conflicts, though this slows down processing.
Lower level (e.g., **Read Committed**)	Everyone works faster, but you might hit small inconsistencies	Canteen opens **two** counters. Food comes quicker, yet occasionally the same samosa is billed to two people before staff notices.

Advantages of Isolation

- **More classmates can work together:** fewer "table occupied" moments in the DB library.
- **Choose your consistency:** critical apps get stricter rules; casual ones get speed.
- **Fewer weird errors:** helps avoid dirty/non-repeatable/phantom reads.
- **Flexible tuning:** you can set the level per transaction if your DBMS allows.

Disadvantages of Isolation

- **Extra overhead:** DB wastes CPU time checking locks/versions.
- **Lower concurrency at high levels:** Serializable may line students up single-file.
- **Not universal:** Some cloud DBs skip Repeatable Read, so your code becomes less portable.
- **Mental load:** Developers must manage isolation levels per module carefully; otherwise, inconsistencies may occur.

Isolation Problems

Phenomenon	What happens in the DB	Hostel twist
Dirty read	T1 changes fees but hasn't committed → T2 reads it. If T1 rolls back, T2 sees fake data.	Senior updates Mess Bill = ₹0 (just testing). You peek before he presses **Undo**. You think owe nothing and leave.
Non-repeatable read	T1 reads a row twice, but T2 commits an update between the two reads → value differs.	You check the room-allocation list at 10 AM (Shows Room 101). After lunch the warden swaps you to Room 202. At 2 PM you re-check; list now says 202.
Phantom read	T1 runs a SELECT query with condition grade > 8. Meanwhile, T2 inserts a new qualifying row and commits. T1 reruns the query and sees an extra row.	You list "all students with 85%+ attendance" to plan a class trip. While you're booking the bus, two friends clear attendance backlogs and now also appear on the list when you refresh.

6. Serializability

Serializability is a concept that ensures the results of executing multiple transactions concurrently are the same as if they were executed one after the other, in some order. It helps in achieving a consistent database state, even when transactions overlap in time.

Conflict-serializability quick test

- Draw a node for each transaction (T1, T2, …).
- Draw an arrow $T_i \rightarrow T_j$ when T_i writes a row and T_j later reads or writes the same row.
- **If the graph has no loops**, the schedule is safe.

Example
T1 writes Ananya's grade.
T2 later reads that grade.
If the graph has an arrow from T1 to T2 and no path from T2 back to T1, there is no loop, so the schedule is safe.

7. Why do we need concurrency control?

On result-day hundreds of teachers, students, and placement staff hit the database together. Without control they could:

- read half-processed/updated data,
- overwrite each other's changes, or
- cause indefinite blocking or deadlocks.

Concurrency-control rules keep everyone safe and keep the system quick.

Real-world applications

- Banking Systems: Guarantees **accurate transaction processing** (deposits/withdrawals) even with simultaneous user access.
- E-commerce Platforms: Prevents **double bookings or inventory errors** during concurrent purchases.
- Social media: Maintains **data consistency** when users interact with content simultaneously.
- Online Reservations: Ensures **accurate availability** and prevents overbooking despite multiple simultaneous reservation attempts.

8. Popular techniques to control concurrency

- **Locking**
  Shared (S) lock – multiple users can read, but no one can write.
  (X) lock – only one transaction can both read and write; others are blocked.

- **Two-Phase Locking (2PL)**
  Growing phase - a transaction grabs all locks it needs.
  Shrinking phase - it releases locks; it cannot take new ones.
  Following this rule guarantees serializability if no lock is acquired after any lock is released (strict 2PL).
  Example:
  Teacher (T1) takes an **X-lock** on row (roll = 22104567, CS101) to change the grade. Placement office (T2) asks for an **S-lock** on the same row to show the grade on screen. T2 waits until T1 commits, then reads the final, correct mark.

- **Timestamp ordering**
  Every transaction gets the current timestamp when it starts.
  If two transactions fight for the same row, the older timestamp wins; the newer one waits or rolls back. This works well when the system does many reads and only few writes, e.g., when students keep refreshing **RESULT_FLAT** while teachers rarely update grades.

- **Optimistic control**
  Assume clashes are rare.
  Transactions read without locks; at commit, the DB checks for conflicts. If a clash is found, one transaction rolls back and restarts. Suitable for read-heavy workloads like statistical reports, where data changes are infrequent.

9. Recovery management - fixing crashes

This is all about getting a database back to normal after something goes wrong. It ensures information stays correct and accessible, even after unexpected issues like computer crashes, software glitches, or human errors.

- **Keeps data safe:** Prevents losing or corrupting important information.
- **Stops disruptions:** Minimizes time when the system is down, so things can get back to normal quickly.
- **Ensures operations continue:** Helps college systems like admission or exam results run smoothly again after a problem.

Types of Database Failures

- Transaction Failure: When a specific transaction cannot complete due to logical errors or constraint violations.
- System Failure: The whole system crashes.
- Media Failure: Physical damage to storage.
- Natural Disasters: Major events causing widespread damage.

Techniques for Database Recovery
These methods help bring the database back online with minimal data loss:

- **Backup and Restore:** Taking regular copies of the database.
  Example: The college IT department takes daily backups of all student attendance records. If the live system crashes, they can load yesterday's data.

- **Log-Based Recovery:** Maintaining a detailed log of all operations performed on the database.
  Example: The online fee payment portal keeps a log of every payment. If the system fails mid-transaction, the log helps either complete the payment fully or cancel it entirely, ensuring accurate recovery so that no payment is duplicated or lost.

- **Shadow Paging:** Maintaining a backup version while changes are made.
  Example: When you update your profile details on the college portal, a "shadow" copy of your old profile exists until the new one is perfectly saved. If something goes wrong, it ensures the old consistent state is retained if changes fail.

- **Checkpointing:** Taking snapshots at specific times.
  Example: The online course registration system saves its state every 15 minutes. If it crashes, it only needs to reprocess registrations from the last 15 minutes, not the whole day.

- **Rollback and Rollforward:** Undoing uncommitted transactions (rollback) and reapplying committed operations from logs (rollforward).
  Example: If your scholarship application fails midway, a rollback cancels any partial updates. Once fixed, a rollforward can apply all successful application data to the system.

Best Practices for Database Recovery
- **Regular Backups:** Back up data frequently (e.g., daily attendance records).
- **Off-Site Backups:** Store copies in different locations (e.g., college admission data stored on a cloud server outside the campus).
- **Test Recovery:** Regularly practice restoring data to ensure it works.
- **Automate Processes:** Automate recovery procedures using dedicated tools to reduce human intervention and errors.

Additional isolation techniques supported by many DBMSs
- Snapshot Isolation
- MVCC (Multi-Version Concurrency Control)

SQL Commands

1. SQL Command Families

Family	Original Definition	Enhanced Definition	Key Commands Added
DDL (Data Definition)	Shape the tables	Defines/modifies database schema with transactional safety	TRUNCATE, RENAME, COMMENT
DML (Data Manipulation)	Add/change rows	Manipulates data with optional returning clauses	MERGE, RETURNING clause
DCL (Data Control)	Access control	Granular permission management	DENY (SQL Server)
TCL (Transaction Control)	Save/undo work	Manages transaction boundaries	SAVEPOINT, SET TRANSACTION

Key DML Commands

- INSERT (insert data in a table)
  Example: INSERT INTO STUDENT (roll No, name, programme)
  VALUES (22105123, 'Nisha Verma', 'BTech ECE');
- UPDATE (update data in a table)
  Example: UPDATE MARKS
  SET grade = 9.1
  WHERE rollNo = 22104567 AND courseCode = 'CS101';
- DELETE (delete data from a table)
  Example: DELETE FROM
  STUDENT WHERE
  roll No
  = 22109999;
  -- Student left the college
- SELECT (select some data from a table)
  Example: SELECT rollNo, name
  FROM STUDENT
  WHERE programme = 'MBA';

Important DDL Change

- ALTER (alter/change the structure of a table)
  ALTER TABLE STUDENT
  ADD COLUMN phoneNo CHAR(10);

2. Operators & Aggregates

SQL provides **two key** categories of tools for everyday calculations:
Operators – let you compare, add, multiply, or combine conditions while the query is running.
Aggregate functions – scan a whole set of rows and give back **one** summarized number such as a count, sum, or average.

Operators (work row-by-row)

Category	Original Operators	Added Operators	NULL Handling
Arithmetic	+ - * /	% (mod), ^ (power)	COALESCE(mark, 0) + 5
Comparison	= <> > <	<=> (NULL-safe equal), IS DISTINCT FROM	WHERE grade IS NOT NULL
Logical	AND OR NOT	ANY, ALL, BETWEEN	WHERE COALESCE(attendance, 0) > 75

Sample queries

Add bonus marks to a score
SELECT score + 5 AS new_score FROM EXAM WHERE rollNo = 22104567;

Find students who scored above 90 and are in the 'CSE' batch
SELECT rollNo, name FROM EXAM WHERE score > 90 AND programme = 'CSE';

Aggregate functions (work on many rows at once)

Function	What it returns	Example question
COUNT(*)	Number of rows	How many students wrote the exam?
SUM(col)	Total of a column	What is the total mess bill this month?
AVG(col)	Average value	What is the average CGPA in IT?
MAX(col)	Highest value	Who has the highest CGPA?
MIN(col)	Lowest value	What is the lowest attendance %?

Sample queries
Count registered students
SELECT COUNT (*) AS total_students FROM STUDENT;

Total sales in the canteen
SELECT SUM(amount) AS total_sales FROM CANTEEN_SALES WHERE saleDate >= '2025-05-01' AND saleDate <= '2025-05-31'

Average CGPA for the IT department
SELECT AVG(CGPA) AS avg_IT_cgpa FROM STUDENT WHERE programme = 'IT';

Highest and lowest hostel fees paid
SELECT MAX(totalFee) AS max_fee, MIN(totalFee) AS min_fee FROM HOSTEL_ACCOUNTS;

Use **row-level operators** when you need to filter rows (e.g., WHERE salary > 5000), perform calculations inside a row (e.g., price × quantity AS lineTotal), or combine conditions (AND / OR). **Use aggregate** functions when you need one summary number such as total students, average marks by course (GROUP BY course Code), or the highest CGPA in the college.

3. SQL Clauses (the “sentence order”)

```
SELECT  column_list
FROM    table_name
WHERE   filter_condition          -- row filter
GROUP BY grouping_columns         -- group rows
by values HAVING group_filter     -- filter groups after grouping
ORDER BY column_list [ASC|DESC] -- sort results
LIMIT count OFFSET start     -- limit output (MySQL/PostgreSQL)
```

Example: Average grade per course, only if avg ≥ 8, top 5 hardest first

```
SELECT courseCode, AVG(grade) AS avgGrade
FROM   MARKS
WHERE  semester = 6
GROUP BY courseCode
HAVING AVG(grade) >= 8
ORDER BY avgGrade ASC
LIMIT 5;
```

4. Joins - putting tables together

Join Type	Original Definition	Performance Tip	Syntax Variants
INNER	Matching rows	Use indexed columns in ON clause	FROM A, B WHERE... (theta) vs A JOIN B ON... (ANSI)
LEFT	All left + matches	Add WHERE B.key IS NOT NULL to simulate INNER	LEFT OUTER JOIN
RIGHT	All right + matches	Rarely used - restructure as LEFT	RIGHT OUTER JOIN
FULL	All rows from both	Expensive - consider UNION ALL	FULL OUTER JOIN

```
-- INNER JOIN sample
SELECT s.rollNo, s.name, m.grade
FROM   STUDENT  AS s
INNER JOIN MARKS AS m
ON s.rollNo = m.rollNo;
```

5. UNION in SQL

UNION combines two SELECTs with the **same column list** and removes duplicates (UNION ALL keeps duplicates).

Example:

```
-- Girls or MBA students
SELECT rollNo FROM STUDENT WHERE gender = 'F'
UNION
SELECT rollNo FROM STUDENT WHERE programme = 'MBA';
```

6. Views in SQL

A **view** acts like a virtual table whose content comes from a query.

Example:

```
CREATE VIEW V_TOPPERS
AS
SELECT rollNo, name, CGPA
FROM   STUDENT
WHERE  CGPA >= 9.0;
```

Why are views useful?

Views encapsulate complex joins behind a virtual table name, Grants read-only access to sensitive data (RBAC) and provide a Denormalised read-helper without storing extra data.

7. Sub-Queries (queries inside queries)

- **Scalar sub-query** (returns one value)

```
-- Returns all students with the maximum CGPA
SELECT name
FROM STUDENT
WHERE CGPA = (SELECT MAX(CGPA) FROM STUDENT);
```

- **Row or table sub-query** (returns many rows)

```
-- Students who scored above course average
SELECT rollNo, grade
FROM   MARKS AS m
WHERE  grade > (
    SELECT AVG(grade)
    FROM   MARKS
    WHERE  courseCode = m.courseCode
);
```

This is a correlated subquery — supported by most DBMSs like MySQL/PostgreSQL.

- **EXISTS / NOT EXISTS** (for membership tests)

```
-- SELECT 1 is used for existence check; it doesn't return data
SELECT rollNo, name
FROM STUDENT s
WHERE NOT EXISTS (
  SELECT 1 FROM ATTENDANCE
  WHERE rollNo = s.rollNo
    AND status = 'ABSENT'
);
```

Indexing, SQL Optimisation, Sharding

1. Need for optimising SQL queries

Slow queries make the whole site feel “laggy”. Good optimisation means pages open faster, less load on the server and smaller cloud bills.

2. Practical techniques to speed up SQL

Tip	One-line idea	Example
Select only needed columns	Avoid SELECT *	The merit-list page needs roll No, name, CGPA; it does **not** need mobile numbers.
Filter early, join later	Put the WHERE clause **before** big joins	First filter to ‘students in CSE 2023 batch’, then join with placements table.
Use indexes in predicates	Put indexed columns inside WHERE or JOIN ON	WHERE roll No = 22104567 uses PK index; WHERE LOWER (name) cannot.
Avoid functions on indexed columns	Functions hide the index	WHERE name = 'Ananya' is fast; WHERE UPPER (name) = 'ANANYA' forces a full scan.
Take advantage of LIMIT / TOP	Fetch only what UI shows	The leaderboard shows top-10 CGPAs; ORDER BY cgpa DESC LIMIT 10.
Examine the execution plan	Let the DB tell you the slow step (EXPLAIN)	If EXPLAIN shows “Seq Scan”, think about adding or fixing an index.

3. Indexing - what and why?

An **index** is a tiny, sorted copy of one or more columns.
The database can **jump straight to the row** instead of scanning the whole table, just like a textbook index jumps straight to page numbers.

4. Index types

Index type	Short idea	Example
Primary (clustered)	Sorts the table itself by the primary-key column.	STUDENT table physically stored by rollNo.
Secondary (non-clustered)	Separate index file points back to rows. Table order stays unchanged.	Index on mobileNo so admin can find a student by phone.
Clustering index	Determines the physical order of data in the table (usually the primary key).	STUDENT table stored in rollNo order. Non-clustered indexes (secondary) point to rows without altering table order.

5. How B-trees and B+-trees store an index

- **B-tree:** Keys in internal nodes guide searches; leaves contain keys and row pointers.
- **B+-tree (used by most DBMSs):** All keys are copied to leaves, and leaves are linked for range scans. Leaves are linked left-to-right, so range scans (e.g., CGPA BETWEEN 8 AND 9, inclusive) are very fast.

Think of it like a library index: internal nodes act as guides, pointing to the exact drawer (leaf node) containing the record.

```
      B-tree:                    B+-tree:
    [Internal]                  [Internal]
   /     |     \                /     |     \
[Leaf] [Leaf] [Leaf]      [Leaf]→[Leaf]→[Leaf]      (linked)
```

6. Scaling a database server

- **Vertical scaling (scale-up)**
  Upgrade a single machine by adding more RAM, faster SSDs, and additional CPUs.
  Example: Upgrading the library's PC with extra RAM sticks
- **Horizontal scaling (scale-out)**
  Add more machines; split or copy data across them.
  Example: Deploying three help desks instead of one during peak fee payment periods.

7. Sharding

Sharding divides a large database into smaller, independent pieces called **shards**, each hosted separately. This boosts **performance, scalability, and availability**.

A university chain keeps **Delhi campus records** on Server-A, **Mumbai campus** on Server-B. Student look-ups stay local; only a cross-campus query touches both servers.

Key Components

- **Shards:** Independent partitions of data (e.g., each department maintains its own student records).
- **Shard Key:** A column that determines where data goes. (e.g., student's Branch determines their department's database.)
- **Shared-Nothing Architecture:** Each shard operates independently.

Sharding Methods

- **Range-Based Sharding:** Divides data by value ranges.
  Example: Student Roll No. 1-1000 → Shard 1.
  Pros: Simple. **Cons:** Can lead to uneven data.
- **Hashed Sharding:** Uses a formula on the shard key for even distribution.
  Example: A hash function applied to the Admission ID determines the target server, ensuring uniform data distribution.
  Pros: Even data spread.
  Cons: Requires rehashing (and data movement) when adding/removing shards, though consistent hashing minimizes this impact.
- **Directory Sharding:** Uses a lookup table to map keys to shards.
  Example: Course ID maps to specific department server.
  Pros: Flexible.
  Cons: This method may fail if the lookup table contains incorrect or outdated mappings.

- **Geo Sharding:** Partitions data by location.
  Example: Delhi Campus students → Delhi Server for low-latency access, with failover to Mumbai if needed.
  Pros: Geo sharding provides faster access by routing users to geographically closer servers, but may result in uneven data distribution if user density varies by location.

Optimizing Sharding

- **Cardinality:** More unique shard key values allow more shards.
- **Frequency:** Avoid popular shard keys to prevent overloaded shards ("hotspots").
- **Monotonic Change:** Avoid monotonic keys (e.g., sequential timestamps) as they can lead to unbalanced shard distribution and write hotspots.

Poor Sharding Example: Sharding student feedback by "courses completed" (e.g., 11+ courses → Shard C) can overload Shard C as students progress.
Sharding is powerful for large datasets. Choosing an appropriate shard key is crucial to ensure performance and balanced data distribution.

8. RBAC - Role-Based Access Control

Define roles: Student, Faculty, Admin.
Attach permissions to roles, not to individual users.
Student → SELECT marks, pay fees.
Faculty → INSERT / UPDATE marks for their courses.
Admin → full rights.
Assign users to roles: Roll No. 22104567 inherits “Student” rights on first login.

This keeps security rules simple and easy to manage.

9. Encryption - keeping data secret

Layer	What is protected?	Example
At rest	Files on disk are scrambled.	A stolen hard disk cannot reveal student Aadhaar numbers.
In transit	Data is encrypted during network travel.	Student portal uses **HTTPS**, so Wi-Fi snoopers cannot read passwords.
Column-level	Encryption is applied only to sensitive columns, such as bank account numbers or Aadhaar.	The bank Account No column is stored with AES encryption; queries must decrypt when needed.

10. Data masking - hiding real values in non-prod copies

Data masking (or obfuscation) hides sensitive original data by replacing it with modified, non-sensitive content. It keeps the data format, making it safe for use in testing, development, or training without risking privacy.

Key Techniques

- **Static Data Masking (SDM):** Modifies data permanently in a copy of the database. Example: The college IT department masks student names and mobile numbers when creating a test database for a new admissions portal.

- **Dynamic Data Masking (DDM):** Masks data on-the-fly as it's accessed, without changing the original data.
  Example: A college helpdesk representative sees only the last four digits of a **student's bank account number** when accessing their fee payment records.
- **Deterministic Data Masking:** Replaces the same original value with the same masked value every time.
  Example: "rollno123@college.edu" is always masked as "masked.student1@college.edu" across all test systems.
- **Non-Deterministic Data Masking:** Replaces the same original value with a different masked value each time.
  Example: A student's **parent's contact number** might be masked differently each time it's pulled for various internal reports, making it harder to link.
  Note: Avoid non-deterministic masking for columns referenced in foreign keys to maintain relational integrity.
- **Format-Preserving Masking:** Masks data while keeping its original format.
  Example: A student's university ID (e.g., AB1234567) is masked in a way that retains the two-letter prefix and the seven-digit format
- **Shuffling:** Rearranges existing data within a column.
  Example: Student **exam scores** are shuffled across different students in a research dataset, making scores realistic but delinked from individuals.
- **Redaction:** Irreversibly removes or blacks out sensitive information from view, unlike masking which keeps structure.
  Example: **Student's medical history** details are completely blacked out before a general academic advisor views their file.
- **Nulling Out:** Replaces sensitive data with SQL NULL values (i.e., empty or missing data).
  Example: For a survey on student satisfaction, student names and addresses are removed by replacing them with NULL values before sharing the data with external analysts.

  Data masking is vital for **protecting sensitive information**. By using these techniques, colleges can ensure data usability in non-production environments while maintaining **privacy and security**.

11. Speed and Safety Checklist

Goal	Tool you reach for
Fast SELECT by primary key	**B+-tree primary index**
Fast range of CGPAs	Secondary B+-tree on CGPA
High read traffic at result time	Denormalized **RESULT_FLAT** view + indexes
Sudden user surge	Add a second read-only replica (**Horizontal Scaling**)
Different campuses	**Shard** on campusID
Protect sensitive data	RBAC roles + column encryption + masking in test dumps
High write throughput	Partitioning + batch inserts

SQL Queries Practice

1. SQL Queries Practice for Interviews

S. No	Explanation	Query
1	Show every staff member's first name, naming the result column EMPLOYEE_NAME.	SQL: SELECT given_name AS EMPLOYEE_NAME FROM staff;
2	Return first names in UPPER-case.	SQL: SELECT UPPER (given_name) FROM staff;
3	List every distinct division represented in the staff table.	SQL: SELECT DISTINCT division FROM staff;
4	Grab the first three characters of each given name.	SQL: SELECT SUBSTRING(given_name,1,3) FROM staff;
5	Find where the letter **B** occurs in the name Siddarth.	SQL: SELECT INSTR(given_name, 'B') FROM staff WHERE given_name = 'Siddharth';
6	Trim spaces on the right of first names.	SQL: SELECT RTRIM (given_name) FROM staff;
7	Trim spaces on the left of division names.	SQL: SELECT LTRIM (division) FROM staff;
8	Show each unique division together with its length.	SQL: SELECT division, LENGTH(division) FROM staff GROUP BY division;
9	Replace every lowercase **a** with uppercase **A** in first names.	SQL: SELECT REPLACE(given_name, 'a', 'A') FROM staff;
10	Combine first and last names into a single column FULL_NAME.	SQL: SELECT CONCAT(given_name,' ',family_name) AS FULL_NAME FROM staff;
11	Display all staff details, ordered by first name A→Z.	SQL: SELECT * FROM staff ORDER BY given_name ASC;
12	Display all staff, ordered by first name A→Z and division Z→A.	SQL: SELECT * FROM staff ORDER BY given_name ASC, division DESC;
13	Pull the records for employees named **Arjun** or **Rohan**.	SQL: SELECT * FROM staff WHERE given_name IN ('Arjun', 'Rohan');
14	Return everyone except **Arjun** and **Rohan**.	SQL: SELECT * FROM staff WHERE given_name NOT IN ('Arjun', 'Rohan');
15	Show workers whose division starts with "Finan".	SQL: SELECT * FROM staff WHERE division LIKE 'Finan%';
16	Get staff whose first name contains the letter **a** anywhere.	SQL: SELECT * FROM staff WHERE LOWER(given_name) LIKE '%a%';

17	Fetch staff whose first name ends with **a**.	SQL: SELECT * FROM staff WHERE LOWER(given_name) LIKE '%a';
18	Fetch staff whose six-letter first name ends with **h**.	SQL: SELECT * FROM staff WHERE given_name LIKE '_____h'; AND LENGTH(given_name) = 6;
19	Show employees with pay between 100 000 and 500 000 (inclusive).	SQL: SELECT * FROM staff WHERE pay BETWEEN 100000 AND 500000;
20	List staff who joined in **February 2014**.	SQL: SELECT * FROM staff WHERE YEAR(hire_date) = 2014 AND MONTH(hire_date) = 2;
21	Count how many people work in the **Finance** division.	SQL: SELECT division, COUNT(*) FROM staff WHERE division = 'Finance';
22	Return full names of staff earning 50 000 – 100 000.	SQL: SELECT CONCAT(given_name, ' ', family_name) FROM staff WHERE pay BETWEEN 50000 AND 100000;
23	Show each division with its head-count, highest first.	SQL: SELECT division, COUNT(staff_id) AS total_staff FROM staff GROUP BY division ORDER BY total_staff DESC;
24	List employees who also appear as **Supervisor** in the role table.	SQL: SELECT * FROM staff AS s INNER JOIN role AS r ON s.staff_id = r.staff_ref_id WHERE r.role_name = 'Supervisor';
25	Find role titles held by more than one person.	SQL: SELECT role_name, COUNT(*) AS cnt FROM role GROUP BY role_name HAVING cnt > 1;
26	Return only rows with odd staff_ids.	SQL: SELECT * FROM staff WHERE MOD(staff_id, 2) != 0;
27	Return only rows with even staff_ids.	SQL: SELECT * FROM staff WHERE MOD(staff_id,2) = 0;
28	Clone the entire staff table into staff_copy.	SQL: CREATE TABLE staff_copy LIKE staff; INSERT INTO staff_copy SELECT * FROM staff;
29	Show rows common to both staff and its copy (by id).	SQL: SELECT s.* FROM staff s INNER JOIN staff_copy USING (staff_id);
30	Show staff records that are **not** in the copy.	SQL: SELECT s.* FROM staff s LEFT JOIN staff_copy USING (staff_id) WHERE staff_copy.staff_id IS NULL;
31	Display today's date and current timestamp.	SQL: SELECT CURDATE(); SELECT NOW();
32	Fetch the top 5 highest-paid employees.	SQL: SELECT * FROM staff ORDER BY pay DESC LIMIT 5;
33	Get the 5th-highest salary using LIMIT.	SQL: SELECT * FROM staff ORDER BY pay DESC LIMIT 1 OFFSET 4;

34	Find the 5th-highest salary **without** LIMIT.	SQL: SELECT pay FROM staff s1 WHERE 4 = (SELECT COUNT(DISTINCT s2.pay) FROM staff s2 WHERE s2.pay >= s1.pay);
35	List employees who share the same salary with someone else.	SQL: SELECT s1.* FROM staff s1 JOIN staff s2 ON s1.pay = s2.pay AND s1.staff_id <> s2.staff_id;
36	Return the second-highest salary overall.	SQL: SELECT MAX (pay) FROM staff WHERE pay NOT IN (SELECT MAX (pay) FROM staff);
37	Show every row twice (duplicate output).	SQL: SELECT * FROM (SELECT * FROM staff UNION ALL SELECT * FROM staff) AS dup ORDER BY staff_id;
38	List staff IDs that do **not** receive bonuses.	SQL: SELECT staff_id FROM staff WHERE staff_id NOT IN (SELECT staff_ref_id FROM incentive);
39	Select the first 50 % of rows (by id order).	SQL: SELECT * FROM staff ORDER BY staff_id LIMIT (SELECT COUNT(*) / 2 FROM staff);
40	Return divisions that have fewer than four employees.	SQL: SELECT division, COUNT (*) AS headcount FROM staff GROUP BY division HAVING headcount<4;
41	Show every division with its employee count.	SQL: SELECT division, COUNT(*) AS headcount FROM staff GROUP BY division;
42	Fetch the very last row in the table (largest id).	SQL: SELECT * FROM staff WHERE staff_id = (SELECT MAX (staff_id) FROM staff);
43	Fetch the very first row in the table (smallest id).	SQL: SELECT * FROM staff WHERE staff_id = (SELECT MIN (staff_id) FROM staff);
44	Display the last five rows, ordered naturally.	SQL: (SELECT * FROM staff ORDER BY staff_id DESC LIMIT 5) ORDER BY staff_id;
45	For each division, list the employee(s) with the top salary.	SQL: SELECT s.division, s.given_name, s.pay FROM (SELECT division, MAX(pay) AS top_pay FROM staff GROUP BY division) t JOIN staff s ON t.division = s.division AND t.top_pay = s.pay;
46	Pull the three highest distinct salaries via a correlated sub-query.	SQL: SELECT DISTINCT pay FROM staff s1 WHERE 3 >= (SELECT COUNT(DISTINCT pay) FROM staff s2 WHERE s2.pay > s1.pay) ORDER BY pay DESC;
47	Pull the three lowest distinct salaries via a correlated sub-query.	SQL: SELECT DISTINCT pay FROM staff s1 WHERE 3 >= (SELECT COUNT(DISTINCT pay) FROM staff s2 WHERE s2.pay < s1.pay) ORDER BY pay;

48	Generic formula for the *n*-th highest salary.	SQL: SELECT DISTINCT pay FROM staff s1 WHERE :n >= (SELECT COUNT(DISTINCT pay) FROM staff s2 WHERE s1.pay<=s2.pay) ORDER BY s1.pay DESC;
49	Show total payroll cost per division, highest first.	SQL: SELECT division, SUM (pay) AS total_pay FROM staff GROUP BY division ORDER BY total_pay DESC;
50	List names of the employee(s) earning the absolute max salary.	SQL: SELECT given_name, family_name, pay FROM staff WHERE pay = (SELECT MAX(pay) FROM staff);

MCQs:

1. **Which benefit of a DBMS is illustrated when a fee clerk can view only the fee status, whereas the principal can also view marks?**
   A. Growth (Scalability)
   B. Security
   C. Speed (Indexing)
   D. No Extra Copies

2. **Flipkart stores every customer's shopping cart in a key-value database primarily because key-value systems offer what advantage?**
   A. Table-level referential integrity
   B. Ultra-fast look-ups via a simple key-value structure
   C. Built-in ACID transactions for complex joins
   D. Time-series data for analytics

3. **The library's subject-based shelf organization mirrors how DBMS _____ accelerate searches by maintaining sorted references to data locations.**
   A. Full-table scans
   B. Clustered indexes (physically reorder data)
   C. Non-clustered indexes (separate sorted references)
   D. Transaction logs

4. **In a three-tier DBMS architecture, which component issues SQL queries to the database server?**
   A. Mobile client app
   B. Application (middle) server
   C. Web browser's JavaScript runtime
   D. Load balancer

5. **A personal inventory system built entirely in Microsoft Access on a single laptop exemplifies which architectural style?**
   A. Two-tier
   B. Three-tier
   C. One-tier
   D. Distributed

6. **According to the CAP-theorem examples, UPI wallet payments favor which two guarantees during a network partition?**
   A. Availability + Consistency
   B. Consistency + Partition Tolerance
   C. Availability + Partition Tolerance
   D. Durability + Consistency

7. **The hostel-warden portal, which displays only Roll No., Name, and Hostel Fee Status, operates at what level of abstraction?**
   A. Physical level
   B. Logical level
   C. View level
   D. File-system level

8. **Which scenario best illustrates physical data independence?**
   A. Splitting an Address column into Street, City, and Pin Code without updating the application code
   B. Moving attendance files from HDDs to faster SSDs and adding a B-tree index, with no change to SQL queries
   C. Creating a new Faculty table linked to Students by Roll No.
   D. Granting the examination branch access to marks via an additional view

9. **Altering the logical schema (e.g., breaking address into three columns) without rewriting other modules demonstrates:**
   A. Physical independence
   B. Logical independence
   C. No independence
   D. View-level caching

10. **Which data model organises records strictly in a top-down parent-child tree where each child has exactly one parent?**
    A. Object-Oriented Model
    B. Hierarchical Model
    C. Network Model
    D. Entity–Relationship Model

11. **A student belonging to several clubs, with each club linked back to many students, illustrates the need for a specific data model.**
    A. Relational
    B. Hierarchical
    C. Network
    D. Time-Series

12. **During conceptual design, engineers draw entities like Student and Course connected by *ENROLLED_IN*. Which data model is being used?**
    A. Entity–Relationship (ER)
    B. Object-Oriented
    C. Relational
    D. Key-Value

13. **A weak entity in an ER model MUST have:**
    A. A many-to-many relationship
    B. Total participation in its identifying relationship
    C. Double-lined relationship diamond
    D. Both B and C

14. **A relationship where one A relates to many B's, and each B to exactly one A is classified as:**
    A. M : N
    B. 1 : N
    C. 1 : 1
    D. M : 1

15. **Treating a relationship set and its participating entities as one higher-level unit so it can participate in another relationship is called:**
    A. Specialization
    B. Generalization
    C. Aggregation
    D. Composition

16. **In the relational model, a single column whose values come from one domain is called a(n):**
    A. Domain
    B. Attribute
    C. Tuple
    D. Relation

17. **Which statement correctly distinguishes a schema from an instance?**
    A. Schema changes with every insert/delete; instance is fixed
    B. Schema is the permanent structure; instance is the current rows
    C. Schema lists current primary-key values; instance lists data types
    D. Schema enforces referential integrity; instance enforces entity integrity

18. **The integrity rule that forbids NULL values in a primary-key column is called:**
    A. Key Constraint
    B. Referential Integrity
    C. Entity Integrity
    D. Domain Constraint

19. **A candidate key is defined as:**
    A. Any unique attribute set, even with extras
    B. A minimal super key—remove one attribute, and uniqueness breaks
    C. The chosen official identifier of the table
    D. A foreign-key reference to another table

20. **Which rule is mandatory for a primary-key column (or set)?**
    A. Must reference another relation's key
    B. Must allow NULLs for optional rows
    C. Must be unique and not NULL
    D. Must include more than one attribute

21. **Which statement is true of super keys, candidate keys, and primary keys?**
    A. Every composite key is a candidate key
    B. Every candidate key is also a super key
    C. Every candidate key is minimal by definition
    D. A primary key may contain duplicates if no foreign key uses it

22. **Which schema violates 2NF?**
    A. STUDENT(rollNo PK, name)
    B. MARKS(rollNo PK, courseCode PK, grade, studentName)
    C. COURSE(code PK, title, credits)
    D. All comply

23. **In BCNF, for every functional dependency X → Y, X must be:**
    A. A non-minimal super key
    B. A foreign key
    C. A candidate key
    D. A subset of Y

24. **The ability to reconstruct the original relation without data loss after decomposition is called the _____ property.**
    A. Dependency preservation
    B. Lossless (non-additive) join
    C. Partial-dependence removal
    D. Referential integrity

25. **Denormalisation refers to:**
    A. Splitting a wide table into smaller ones
    B. Combining normalized tables or including redundant data to reduce costly joins
    C. Encrypting data so only DBAs can view it
    D. Applying BCNF to all relations

26. **Denormalisation is justified when the workload is mainly:**
    A. Heavy on writes, light on reads
    B. Write-once, read-never
    C. Read-intensive with few updates
    D. CPU-bound on triggers

27. **The chief drawback of denormalising a schema is:**
    A. Higher risk of update anomalies due to duplicated facts
    B. Loss of primary-key enforcement
    C. Mandatory distributed transactions for every write
    D. Loss of the lossless-join property among the remaining tables

28. **Durability is achieved through:**
    A. Write-ahead logging (WAL)
    B. Two-phase locking
    C. Snapshot isolation
    D. All of the above

29. **The isolation level that blocks dirty reads but still allows non-repeatable reads is:**
    A. Read Uncommitted
    B. Read Committed
    C. Repeatable Read
    D. Serializable

30. **Strict 2PL improves basic 2PL by:**
    A. Releasing locks immediately after use
    B. Holding all locks until commit/abort
    C. Using timestamp ordering
    D. Eliminating the shrinking phase

31. **The SQL family used to grant or revoke privileges is:**
    A. DML
    B. DDL
    C. DCL
    D. TCL

32. **After grouping rows, which clause filters entire groups based on aggregates?**
    A. WHERE
    B. HAVING
    C. ORDER BY
    D. LIMIT

33. **Which aggregate counts all rows, including those with NULLs?**
    A. COUNT (column_name)
    B. COUNT (*)
    C. AVG (column_name)
    D. SUM (column_name)

34. **Which index type physically orders table rows by the key itself?**
    A. Secondary (non-clustered)
    B. Primary (clustered)
    C. Bitmap
    D. Hash

35. **Adding more machines and partitioning data across them to handle the load is called:**
    A. Vertical scaling (scale-up)
    B. Horizontal scaling (scale-out)
    C. Query parallelism
    D. Snapshot isolation

36. **In RBAC, permissions are first attached to:**
    A. Individual users
    B. Database triggers
    C. Roles
    D. Stored procedures

ANSWER KEY									
1.	(B)	2.	(B)	3.	(C)	4.	(B)	5.	(C)
6.	(B)	7.	(C)	8.	(B)	9.	(B)	10.	(B)
11.	(C)	12.	(A)	13.	(D)	14.	(B)	15.	(C)
16.	(B)	17.	(A)	18.	(C)	19.	(B)	20.	(C)
21.	(B)	22.	(B)	23.	(C)	24.	(B)	25.	(B)
26.	(C)	27.	(A)	28.	(A)	29.	(B)	30.	(B)
31.	(C)	32.	(B)	33.	(B)	34.	(B)	35.	(B)
36.	(C)								