Session 1

1 - Difference Between DML and DDL in Databases.

🔷 DDL – Data Definition Language

Used to **define**, **create**, or **modify** the structure of database objects (tables, views, indexes, schemas).

✔ Common DDL Commands:

Command	Meaning
CREATE	Create new tables, databases, views
ALTER	Modify an existing table structure
DROP	Delete tables or databases
TRUNCATE	Remove all data (reset table)

📝 Notes:

- DDL changes affect **schema**.
- Changes are usually **auto-committed** (cannot be rolled back in some DBs).

🔷 DML – Data Manipulation Language

Used to **manipulate the data** stored inside tables.

✔ Common DML Commands:

Command	Meaning
SELECT	Retrieve data from tables
INSERT	Add new data
UPDATE	Modify existing data
DELETE	Remove specific records

📝 Notes:

- DML affects **data**, not structure.
- DML changes can be **rolled back** using transactions.
  (e.g., ROLLBACK, COMMIT)

---

2 - What is a View?

A **view** is like a saved SQL query that you can treat as a table.

- It looks like a table
- You can `SELECT` from it
- But it **does not store data** (unless materialized view)
- Data inside a view updates automatically when the underlying tables change

📌 Example

Suppose you have a table:

employees

id	name	salary	department

Create a view:

```sql
CREATE VIEW high_salary AS
SELECT name, salary
FROM employees
WHERE salary > 5000;
```

Query the view:

```sql
SELECT * FROM high_salary;
```

This returns only employees with salary > 5000.

---

3 - Primary Key vs Foreign Key — What's the Difference?

✅ Primary Key (PK)

A **primary key** uniquely identifies each record (row) in a table.

✔ Rules:

- Must be **unique**
- Cannot be **NULL**
- Each table can have **only one** primary key
  (but it can be made of multiple columns → composite key)

✔ Example:

Table: **Students**

student_id (PK)	name
1	Ali
2	Sara

Here, `student_id` uniquely identifies every student.

✅ Foreign Key (FK)

A **foreign key** is a field that **links a record in one table to the primary key in another table**.

✓ Purpose:

- Creates **relationships** between tables
- Maintains **referential integrity** (cannot reference non-existing data)

✓ Example:

Table: **Enrollments**

enrollment_id	student_id (FK)	course
1	1	Math
2	2	Physics

`student_id` in this table references the `student_id` from **Students** table.

---

4 - What Normalization Does ?

Normalization breaks large, messy tables into **smaller, related tables**.

It helps to:

- Remove **duplicate data**
- Avoid **update anomalies**
- Avoid **insert/delete problems**
- Ensure **data consistency**

🔥 Example (Simple)

❌ Bad Table (Not normalized)

student_id	student_name	course1	course2

Problems:

- Repeated columns (course1, course2)
- Redundant data
- Hard to add/remove courses

✓ Normalized (Better Design)

Table 1: Students

student_id	student_name

Table 2: Enrollments

student_id	course

No duplication, easy to add unlimited courses.

5 - what is the difference between semi structured data and unstructured data ?

📌 Unstructured Data

Data that **has no predefined format or organization**.

✔ Characteristics:

- No fixed schema
- Hard for machines to understand directly
- Needs processing (NLP, image processing, etc.)

📚 Examples:

- Images (JPEG, PNG)
- Videos
- Audio files
- Word/PDF documents
- Emails (raw body text)
- Social media posts
- Sensor logs (raw)

🧠 In short:

> **Unstructured = free-form data, no structure, hard to query.**

📌 Semi-Structured Data

Data that **does not follow a rigid table structure**, but **has some organizational tags or markers**.

✔ Characteristics:

- No fixed table format
- But contains **metadata**, tags, or a hierarchy
- Easier to search and parse than unstructured data

📚 Examples:

- **JSON**
- **XML**
- **HTML**
- **YAML**
- **NoSQL documents (MongoDB)**
- **Logs with structured fields**
  - Example: `timestamp=2025-01-01 user=Ali action=login`

🧠 In short:

> **Semi-structured = not fully tabular but has tags/metadata that give structure.**

6 - Why NoSQL Databases?

1️⃣ To Handle Large-Scale Data (Big Data)

Modern applications generate massive amounts of data:

- Social media feeds
- Logs
- Sensors
- E-commerce events
- Streaming data

SQL databases struggle when the data becomes extremely large and grows quickly.

➡️ **NoSQL can store millions/billions of records efficiently.**

2️⃣ To Support High Performance & Real-Time Apps

Applications today require:

- Very fast reads/writes
- Low latency
- Real-time responses

SQL DBs slow down when traffic is high.

➡️ NoSQL uses distributed storage → **faster performance under high load**.

3️⃣ Flexible Schema (Schema-less)

SQL requires a fixed table structure:

bash

```bash
CREATE TABLE users (id, name, age...)
```

But modern apps change quickly.

NoSQL allows **dynamic schema**:

Example in MongoDB (NoSQL):

json

```json
{
  "name": "Ali",
  "email": "ali@test.com",
  "skills": ["Python", "DevOps"]
}
```

No need to ALTER tables → **agile development**.

4️⃣ Easy Horizontal Scalability

SQL scaling = expensive hardware (vertical scaling)

NoSQL scaling = add more machines (horizontal scaling)

Like:

- MongoDB
- Cassandra
- DynamoDB

➡️ Allows cloud-scale applications.

5️⃣ Good for Semi-Structured & Unstructured Data

SQL works best with **structured data**.

But modern data is often:

- JSON
- XML
- Logs
- Images (metadata)
- IoT data

➡️ NoSQL can store these formats **natively**.

6️⃣ Different Types for Different Needs

NoSQL has 4 main types:

Type	Examples	Best For
Document DB	MongoDB, CouchDB	JSON-like data
Key-Value Store	Redis, DynamoDB	Cache, sessions
Column-family	Cassandra, HBase	Big Data analytics
Graph DB	Neo4j	Relationships (social networks)

➡️ SQL cannot handle these specialized requirements efficiently.

---

7 - What Is a Schema?

A **schema** defines:

- What **tables** exist
- What **columns** each table has
- Data types (INT, VARCHAR, DATE...)
- **Constraints** (primary key, foreign key, unique)
- Relationships between tables
- Views, indexes, stored procedures, etc.

It is the **overall design** of the database.

🔧 Example of a Simple Schema

Students Table

Column	Type	Constraint
id	INT	Primary Key
name	VARCHAR	-
age	INT	-

---

8 - main types of Entities:

🔷 1. Strong Entity (Independent Entity)

- Exists **on its own** without depending on another entity.
- Has its **own primary key**.

Example

- Student
- Employee
- Product

These do NOT rely on another table for identification.

🔷 2. Weak Entity

- **Cannot exist without another entity**.
- Depends on a **strong entity**.
- Does **not have a full unique key** by itself.
- Identified by a **partial key + primary key of parent**.

Example

- OrderItem depends on Order
- Room depends on Building

Example Table

OrderItem:

```scss
order_id (FK)
item_number (partial key)
```

🔷 6. Recursive Entity

An entity that has a **relationship with itself**.

Example

Employee → Manager (Employee)

It's the same entity relating to itself.

---

9 - Main types of attributes:

🔷 1. Simple Attribute

A basic attribute that **cannot be divided** into smaller parts.

✔ Examples:

- `name`
- `age`
- `price`
- `email`

These are atomic (single value).

🔷 2. Composite Attribute

An attribute that **can be broken down** into smaller sub-attributes.

✔ Example:

Address → made of:

- street
- city
- country
- zip_code

Composite attributes are used when the value has meaningful parts.

🔷 3. Single-Valued Attribute

Holds **only one value** for a record.

✔ Examples:

- `date_of_birth`
- `salary`
- `national_id`

One employee → one salary.

🔷 4. Multi-Valued Attribute

Can store **multiple values** for a single entity.

✔️ Examples:

- `phone_numbers` → employee may have many
- `skills` → user may have many

In ERD — shown with **double oval**.

🔷 5. Derived Attribute

An attribute that is **calculated** from other attributes.

✔️ Examples:

- `age` → derived from date_of_birth
- `total_price` → derived from quantity × price
- `duration` → from start_date and end_date

Doesn't need to be stored physically.

---

10 - Three types of relationships in ER diagrams

🔥 1️⃣ One-to-One (1:1) Relationship

One entity is related to **one and only one** of another entity.

✔️ Example:

Person ↔ Passport

- Each person has **one** passport
- Each passport belongs to **one** person

Diagram:

```scss
Person (1) ———————— (1) Passport
```

🔥 2️⃣ One-to-Many (1:M) Relationship

One entity is related to **many** of another entity.

✔️ Example:

Department → has many → **Employees**

- One department → many employees
- Each employee → belongs to one department

Diagram:

```scss
Department (1) ————————< (M) Employee
```

🔥 3️⃣ Many-to-Many (M:N) Relationship

One entity has a relationship with **many** of another, and vice versa.

✔ Example:

Students ↔ Courses

- A student can enroll in many courses
- A course can have many students

To represent this, we need a **bridge table** (Associative Entity).

Diagram:

```scss
Student (M) >───────< (N) Course
            |        |
            └── Enrollment ──┘
```

---

11 - What Is the Relational Model?

The **Relational Model** organizes data into **tables** that are related to each other using **keys**.

Each table consists of:

- **Rows** → called *tuples*
- **Columns** → called *attributes*

Relationships between tables are formed using **primary keys** and **foreign keys**.

---

12 – What is the Database Transaction?

✅ Definition

A **transaction** is a sequence of operations that must be **executed completely or not at all** to keep the database consistent.

💡 Example

Transferring money between two bank accounts:

1. Withdraw 100 from Account A
2. Deposit 100 into Account B

These two operations **must both succeed**, otherwise the system ends up with wrong data.

So they are wrapped in **one transaction**.

🎯 ACID Properties (very important in interviews)

Transactions follow **ACID**, which ensures reliability.

1️⃣ Atomicity

"All or nothing"
If one step fails, the whole transaction is canceled.

2️⃣ Consistency

Transaction must leave the DB in a **valid state**.

3️⃣ Isolation

Multiple transactions must **not affect each other**.

4️⃣ Durability

Once committed, the data is **permanently saved**, even if power goes off.

Consistency (the "C" in ACID) means:

> **After a transaction finishes, all the data in the database must still follow the rules, constraints, and relationships defined in the database.**

In simple words:
The database should not become "corrupted" or contain invalid or impossible data after the transaction.

---

13 – What is MariaDB ?

✅ In Simple Words

> **MariaDB = MySQL but faster, open-source, and with extra features.**

You can use almost the same SQL commands, same connectors, same tools.

🏗️ Why Was MariaDB Created?

When Oracle bought MySQL, many developers feared:

- MySQL would become closed-source
- Development would slow down

So the original MySQL creators started **MariaDB** to keep a free alternative.

14 – Difference between Row and Column Based BDs

Feature	Row-Based	Column-Based
Storage	Row by row	Column by column
Best for	OLTP (transactions)	OLAP (analytics)
Read performance	Fast for full row queries	Fast for columnar/aggregated queries
Write performance	Faster for inserts/updates	Slower for single-row inserts
Compression	Low	High (similar values stored together)
Example	MySQL, PostgreSQL	Cassandra, Redshift, ClickHouse

---

15 - What Is a Column-Oriented Database?

A **column-oriented database** stores data **column by column** rather than row by row.

- All values of a single column are stored together.
- Columns can be accessed independently of other columns.
- Ideal for **read-heavy, analytical queries**.

🔷 3. Advantages of Columnar Storage

1️⃣ Fast Analytical Queries

- Aggregations like `SUM`, `AVG`, `COUNT` only scan **relevant columns**.
- Less I/O than scanning whole rows.

2️⃣ Better Compression

- Similar values stored together → higher compression ratios.
- Reduces disk usage and improves performance.

3️⃣ Efficient for OLAP

- Data warehouses and BI tools often only need a few columns from millions of rows.

4️⃣ Parallel Processing

- Columns can be read independently → great for distributed systems.

🔷 4. Disadvantages

1️⃣ Slower for Row Operations

- Inserting/updating a single row is slower because values are split across columns.

2️⃣ Not ideal for OLTP

- Frequent single-row writes are inefficient.

3️⃣ Complex Updates

- Updating multiple columns of a single row can require multiple disk operations.