Welcome to Lecture 38!

Topic: Databases

Databases: The Mental Model

What a database is, what it solves, and how to choose SQL vs NoSQL

Your App

Create users
Place orders
Search data

→

Database

Stores data safely
Answers questions fast
Handles many users

Question: If your server restarts right now, what happens to user data?

Why do we need a database?

Attempt 1: In memory

Data in variables / arrays.

✅ Fast
❌ Restart wipes everything

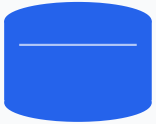
“Deploy → data = gone”

Attempt 2: A file (JSON/CSV)

✅ Survives restart
❌ Searching = scan whole file
❌ Two writes can clash
❌ Partial write can corrupt data

Attempt 3: Database

✅ Persistent
✅ Fast search
✅ Safe concurrent updates
✅ Recovery tools

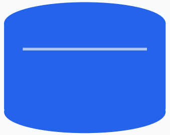


Databases exist because apps need data to be: persistent, searchable, and correct.

What is a Database?

Database vs DBMS (the software that runs it)

Database



An organized store of data built for reliable reads/writes and fast questions (queries).

DBMS (Database Management System)

The software that stores, indexes, secures, and serves your data.
Examples: MySQL, PostgreSQL, MongoDB.

Think: DB = data • DBMS = engine

What does a Database give you?

- **Persistence**

  Data survives restarts/crashes.

- **Querying**

  Ask questions like: “marks > 70”.

- **Indexing**

  Fast lookup without scanning everything.

- **Concurrency**

  Many users can write safely.

- **Transactions**

  All-or-nothing updates.

- **Constraints**

  Keep bad data out (unique, not null).

- **Backup & recovery**

  Undo accidents / restore after failure.

Tip: when choosing a *type* of database for different applications, you’re mostly choosing trade-offs across these features.

Persistence

"If the server restarts… do we still have the data?"

In-memory

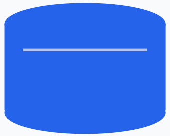
Fast
But disappears on restart

File

Survives restart
But hard to update safely

Database

Survives restart
+ safe updates



Think like an engineer:

"If a request is in-flight and the server crashes mid-write, what state is the data in?"

Querying

"Ask questions about data without writing loops every time"

Typical app questions:

- "Find student whose id=42"
- "Students with marks > 70"
- "Orders from last 7 days"
- "Top 5 best-selling products"

Key idea:

Your app asks a question and DB returns matching rows/documents.

Without a DB (file scan):

```
for each record:
  if matches → keep it
```

With a DB:

```
Execute query -> results
```

Indexing

A fast lookup structure that avoids scanning everything

Analogy: book index

Without an index:

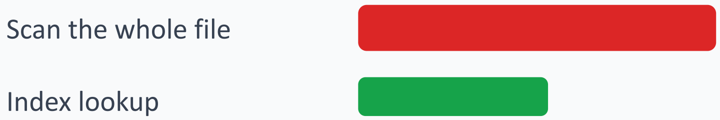
You flip every page until you find the word.

With an index:

You jump directly to the page.

```
Ex: Finding names in a phonebook
Asha -> p. 12
Sucheta -> p. 41
Zoya -> p. 58
```

Conceptual performance



Takeaway:

Indexes speed up reads, but add work on writes (the index must be updated).

Concurrency Control

When multiple users read/write at the same time

Scenario: “Last seat available”

Two users click “Book” at the same time.

Problem: both users book 1A at the same time will lead to double booking

User A	User B
View seats	View seats
Decide to book seat 1A	Decide to book seat 1A
Book seat 1A	Book seat 1A
Confirm booking	Confirm booking

Databases provide safe concurrent updates so you don’t “lose” writes.

Transactions

All-or-nothing updates (to prevent half-finished state)

Scenario: placing an order

These steps must stay consistent:

- 1) Create order record
- 2) Reduce inventory
- 3) Record payment

What can go wrong?

Order created ✅
Inventory reduced ✅
Payment recorded ❌ (crash)

Transaction guarantee:

Either all 3 happen, or none happen.

In one sentence: A transaction prevents the database from ending up in a “half-updated” state.

Different ways to store data

Same information, different shapes

Tabular (tables)

```
Students
+----+-------+------+
| id | name  | mark |
+----+-------+------+
| 1  | Asha  | 85   |
| 2  | Rahul | 62   |
+----+-------+------+
```

Great for structured data + relationships.
Eg: MySQL, Postgres

Document (JSON-like)

```
{
  "id": 1,
  "name": "Asha",
  "mark": 85,
  "courses": ["Math","CS"]
}
```

Great when data is naturally nested.
Eg: MongoDB, CouchDB

Key-Value

```
"student:1"  →  { ... }
"student:2"  →  { ... }
```

Great for super-fast lookups by key.
Eg: Redis

Relational databases

Tables + IDs + relationships

Example: Users & Orders

Users

```
+----+--------+
| id | name   |
+----+--------+
| 1  | Asha   |
| 2  | Rahul  |
+----+--------+
```

Orders

```
+------+---------+
| id   | user_id |
+------+---------+
| 501  | 1       |
| 502  | 1       |
| 503  | 2       |
+------+---------+
```

Key ideas:

- “id” uniquely identifies a row (primary key)
- user_id links Orders ⟶ Users (relationship)

Why this is powerful

You can ask questions across tables.

Example question:
“Show order ids with user names”

Joins (concept)

Combine related rows from multiple tables

A join answers questions that span tables by matching IDs.

Users

id	name
1	Asha
2	Rahul

Orders

match on Users.id = Orders.user_id

id	user_id
501	1
502	1
503	2

Join result

order_id	name
501	Asha
502	Asha
503	Rahul

Takeaway: Relational databases shine when you need relationships + cross-table questions.

Non-relational (NoSQL) databases

Data stored in shapes beyond tables

NoSQL is a family of storage models, often chosen to match access patterns.

Document DB

Store a whole “thing” as a JSON-like document.

```
{
  "id": 1,
  "name": "Asha",
  "orders": [501, 502]
}
```

Good fit when your app typically fetches the whole object at once.

Key-Value store

A huge map: key → value

```
"session:abc" → { userId: 42 }
"student:1"   → { ... }
```

Good fit for caching, sessions, and ultra-fast lookups.

NoSQL types (quick map)

Different tools for different access patterns

Document

JSON-like documents.
Good for user profiles, catalogs.

Key-Value

key -> value.
Good for caching & sessions.

Wide-column

Optimized for huge write throughput.
Good for logs/time-series.

Graph DB

Nodes + edges + traversals.
Good for recommendations / relationships.

Key idea: pick the model that matches your most common queries and reads/writes.

SQL vs NoSQL (high-level)

Not “which is better,” but which fits the problem

SQL (commonly relational)

- Data in tables + relationships
- Joins are a first-class idea
- Strong correctness tooling (constraints, transactions)
- Great for reporting & cross-entity questions

NoSQL (non-relational models)

- Data in documents / key-value / graph / ...
- Often model around app access patterns
- Flexible structure (varies by system)
- Great when you mostly fetch by key / whole document

Two myth-busters:

- NoSQL ≠ “no schema” (structure can still exist)
- SQL ≠ “can’t scale” (many SQL DBs scale well)

Constraints

Rules the database enforces to keep data correct

Why constraints exist:

App code can have bugs. Constraints are guardrails at the data layer.

Common constraints (examples)

- UNIQUE: email must be unique
- NOT NULL: name cannot be empty
- CHECK: marks >= 0
- FOREIGN KEY: order.user_id must exist in users

Bad data attempt

Constraint: UNIQUE(email)

```
INSERT user(email="asha@x")
INSERT user(email="asha@x")
```

Result: second insert is rejected

When to use SQL (relational)

Choose SQL when relationships + correctness are central

SQL is a strong default when:

- Your data has clear structure (tables make sense).
- Relationships matter (users <-> orders <-> payments).
- You need cross-entity questions (joins, reports), where entities can be users, orders, etc
- Correctness is critical (constraints, transactions)

Examples: Payments, e-commerce orders, HR systems, analytics/reporting tables

When to use NoSQL

Choose a model that matches how your app reads/writes

NoSQL is a strong fit when:

- Data is naturally nested (JSON-like objects).
- Schema changes frequently or varies across records.
- You mostly fetch by key or by whole document.
- You want a specialized model (cache, graph traversal, time-series).

Examples: User profiles, product catalogs, caching, logs/events, social graphs

Decision guide: SQL vs NoSQL

A repeatable checklist you can apply to any scenario

Ask these 4 questions (in order):

1) Do relationships & joins matter?	Often SQL
2) Do you need multi-step correctness (transactions)?	Often SQL
3) Is the data naturally nested JSON?	Often Document DB
4) Is it mostly key lookups / caching / managing user sessions?	Often Key-Value

Quick exercise (30 sec): pick SQL or NoSQL for each and justify with the checklist.

A) E-commerce orders B) Cache for recent searches C) User profile page