DATABASE & STORAGE

Fundamentals

* **Relational Databases**

If we store everything randomly in files, it becomes:

- Hard to query
- Hard to maintain consistency
- Easy to corrupt

So we need structured storage.

- Relation DB store data in tables (rows & columns)
- Each table represents an entity.
- **Why relational**

  Because tables are connected using relationships.

- Relational DBs provides:
  - structured storage
  - Data consistency
  - strong querying capability
  - Transaction support

Core Features

1) Tables → Data organised in rows and columns.
2) Primary Keys → Each row must have unique identifier.
3) Foreign Keys → Used to create relationships between tables.
4) SQL (Allows powerful querying)
5) ACID Properties (a Imp)
   - Atomicity → Either 100% or abort (no middle point)
   - Concistency → Data stays same for all.
   - Isolation → Separate flow of processing per query.
   - Durability → Permanently stored, even if system turns off.

SQL vs NoSQL

- **SQL DB** :- • stores data in tables
  • Use rows and colums
  : Supports ACID transactions
  • Enforce schema
  • Maintain relationships.

- **NoSQL** :- Not only SQL, Types :- 1) Document (MongoDB)
  2) Key-value (Redis)
  3) Column-family (Cassandra)
  4) Graph (Neo4j)
  : Scale horizontally easily
  • Optimize for speed & flexibility

When they are needed?

SQL

- In systems like banking, Hospital, E-commerce orders, We need :
  • Guaranteed consistency
  • Transaction safety
  • Complex Joins
  • Foreign Key enforcement.
  • Scales vertically (traditional)
  • Read replicas for read scaling.

NoSQL

- Modern systems may need :
  • Massive scaling
  • Flexible data structures
  • Rapid iteration
  • Large volumes of data.

Ex:- social media posts.
- Each post may have different fields
- Nested structure

While designing system, ask/think :

- Do you need strong transactions?
- Is data highly relational?
- Is consistency critical?
- Do you expect Massive scale?
- Data structure will change frequently?.

* If consistency & relationships matter → SQL
* If scale & flexibility → NoSQL

Often → Hybrid approach.

No SQL Types & Usecase

- Different System needs different performance patterns :-
  - Fast Key lookup
  - Massive write scaling.
  - Flexible JSON storage

1) Key-Value DB → 'user_123' → {name : "John", age : 30}

They are designed for fast reads & writes, simple access patterns & high performance.

Use Case :- caching, session storage, rate limiting counters, Temporary tokens.

2) Document DB → { 'user_id' : 123,
'name : Jonh',
'orders' : [{'id' : 1, 'amt' : 100}]
}

- For semi structured data, which changes frequently.

Use case :- User Profile, Product catalog, Logs, Configuration store.

3) Column-family → store data by columns instead of rows.

- Optimised for large scale distributed storage.
- Horizontally scalability.

4) Graph Database → store : Nodes (entities) & edges (relation ship)

- Allow efficient graph queries

Uses case :- Social network, Fraud detection, Knowledge graphs.

* Use different databases for different needs.

Transaction

- A group of database operations that must succeed or fail together. Either all completes or none applies.
- Transaction ensure -
  - Data remains correct
  - System stays consistent
  - Failures don't corrupt data.
- critical in :-
  - Banking
  - Payments
  - Order processing
  - Inventory systems.

ACID Properties (Written behind)

Why Isolation levels are needed ?

- Imagine two users update the same data at same time.
- Without Isolation -
  - one user may read half-completed data.
  - Updates may overwrite each other.
  - Inconsistent results may occur.

Problems Solved By Isolation

1) Dirty read → 'A' reads data that transaction B updated but not committed. If 'B' rolls back, 'A' saw invalid data.

2) Non-Repeatable read →
   - A reads a row.
   - B updates that row
   - A reads again & sees different value.
   - Same query → different result.

3) Phantom read
   - A reads rows matching condition.
   - B inserts new matching row.
   - A reads again & sees extra row.