SQL Databases

SQL databases (relational databases) store data in tables, just like neatly arranged Excel sheets.

Each table stores one type of information

Each row is one record

Each column is one property of that record

Primary keys uniquely identify rows

Foreign keys link tables together

SQL databases work best when your data has clear structure and relationships.

Example: Online Store (Orders System)

Customers Table

customer_id (PK)	name	email
1	Aditi Kumar	aditi@mail.com
2	Varun Singh	varun@mail.com
3	Ishita Rao	ishita@mail.com

Products Table

product_id (PK)	product_name	price
101	Headphones	1500
102	Mouse	700
103	Keyboard	1800

Here:

A customer can place many orders
An order belongs to one customer
Each order contains one product (simple example)

Foreign keys maintain these links.

Orders Table

order_id	customer_id (FK)	product_id (FK)	order_date
5001	1	101	2024-05-01 10:00:00
5002	3	103	2024-05-01 10:10:00
5003	1	102	2024-05-01 10:20:00

When to use sql

Your data is structured

Tables have relationships (users → orders, products → categories)

You need correct, consistent data (payments, inventory, bookings)

You need complex queries (joins, filters, analytics)

Common Misconception: SQL Doesn't Scale

People often assume SQL doesn't scale,
but modern relational systems scale surprisingly well using:

Read replicas
Sharding/table partitioning
Connection pooling
Caching layers (Redis, Memcached)

Companies like Uber, Flipkart, PayPal,
Spotify rely heavily on SQL foundations.

Scalability is more about the system
architecture than the database engine alone.

NoSQL Databases

NoSQL databases are designed for cases where traditional SQL tables
don't fit well—especially when your data is flexible, nested, huge in volume, or accessed in unusual ways.
Instead of strict tables and fixed schemas,
NoSQL gives you freedom in how you store and structure data.

Different NoSQL systems solve different problems,
so NoSQL is not "one thing"—it's a collection of different database styles.

There are four major types:

Document databases

Key-value stores

Wide-column stores

Graph databases

Document Databases (e.g., MongoDB, Firestore)

Document databases store data as JSON-like documents.
Each document can have its own structure, so fields
don't need to be identical across records.

When useful?
When your data shape keeps changing
When you want to store nested, hierarchical data
When avoiding joins improves performance
When different users have different types of details

When to use:

Product catalog
User profiles with variable fields
Blog/News posts with flexible structure
Content-heavy apps

example: Product Catalog (E-commerce)

```
{
  "_id": "p101",
  "name": "Running Shoes",
  "brand": "Puma",
  "sizes": [6,7,8,9],
  "images": ["img1.jpg", "img2.jpg"],
  "specs": {
    "material": "Mesh",
    "weight": "280g",
    "warranty": "6 months"
  }
}
```

```
{
  "_id": "p102",
  "name": "Bluetooth Speaker",
  "brand": "JBL",
  "battery_life": "12 hours",
  "waterproof": true,
  "features": ["Bass Boost", "USB-C", "Aux Support"]
}
```

Key–Value Stores (Redis, DynamoDB, Memcached)

These are the simplest form of databases:
You store values that can be fetched using a single key.

When useful?

Caching
Sessions
Quick lookups
High-write systems
Feature flags

No filtering, no querying, no relationships.

Trade-off:

Extremely fast
But no joins, no complex queries
You often duplicate data

Often used alongside SQL (not instead of SQL).

example:

Key → Value

```
"session:uid_401" → {
     "token": "ab2910xjq0pq",
     "expires_at": 1710001220
}
```

```
"product:101" → "Running Shoes | Rs. 2999 | Puma"
```

Wide-Column Stores (e.g., Cassandra, HBase)

These databases are optimized for huge write loads.
Data is stored in rows, but each row can have different columns
—unlike SQL where every row must follow the same structure.

Best suited for time-series and log-like data.

When useful?

Analytics
Logging events
IoT sensor readings
High-speed writes
Massive scale

example:

Row key → user ID
Columns → timestamp : event data

```
user_2001: {
  "2024-06-10T10:01": "login",
  "2024-06-10T10:03": "view:item_110",
  "2024-06-10T10:05": "add_to_cart:item_110",
  "2024-06-10T10:08": "checkout"
}
```

Every new reading is just an append, not an update.

Graph Databases (e.g., Neo4j, Amazon Neptune)

Graph DBs store information as nodes and relationships.
Perfect for exploring connections like:

"Who knows whom?"
"What product is similar to what?"
"How are these entities connected?"

When useful?

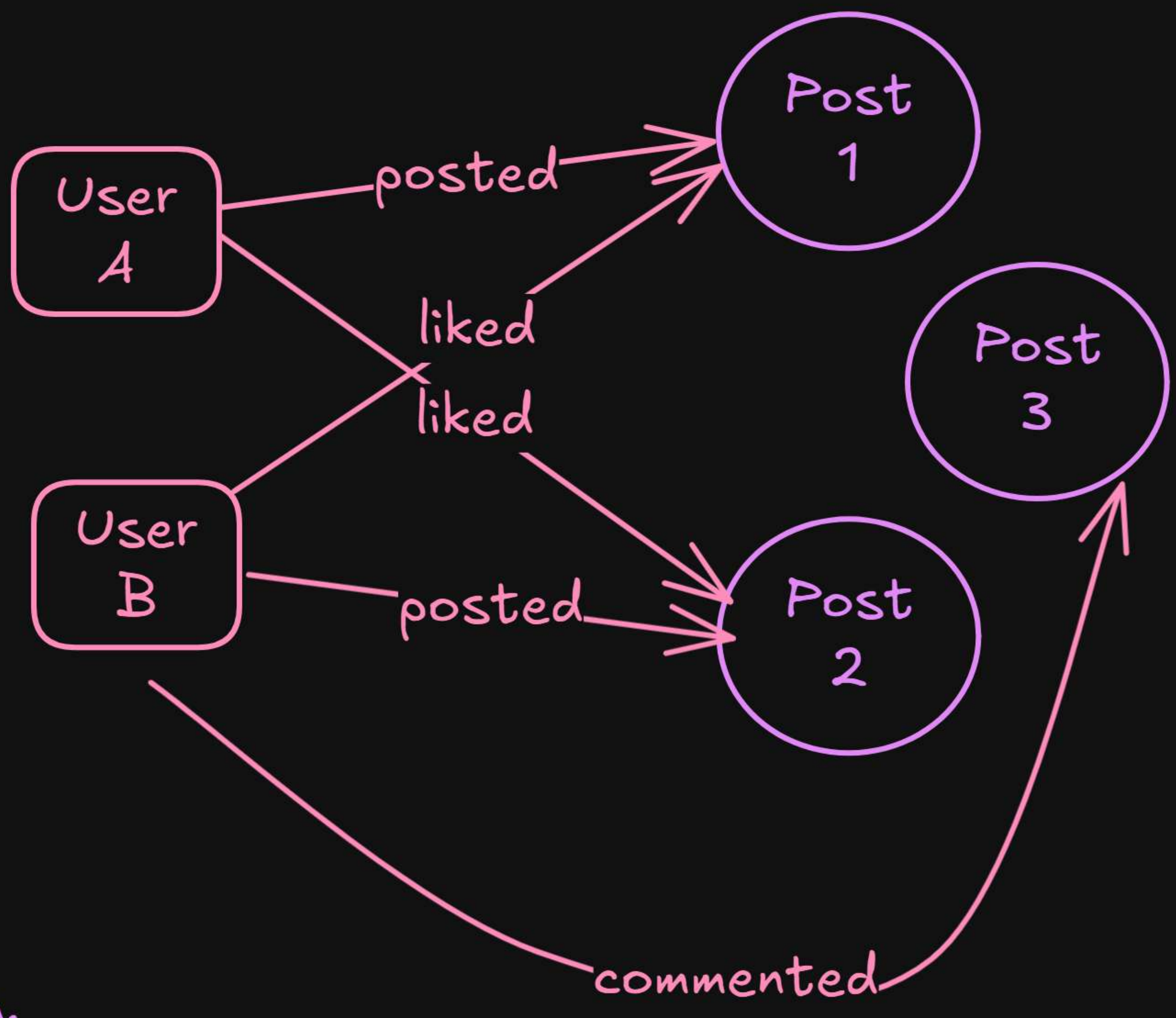
Rarely in system design interviews unless
the question specifically needs deep relationship traversal.

But they make sense when relationships themselves are the main data.

Trade-off:

Great for deep relationship queries
But operationally heavy

Most companies still use SQL for relationships



Aspect	SQL Databases	NoSQL Databases
Data Model	Tables with rows & columns	Documents, key-value, wide-column, graphs
Schema	Fixed, structured	Flexible, fields can vary
Relationships	Excellent support (joins, FK)	Limited or manual (embedding/denormalization)
Consistency	Strong, ACID compliance	Often eventual (but improving)
Query Power	Very powerful (joins, aggregations)	Depends on type; simpler in KV/document
Best Use-Cases	Orders, payments, inventory, core business data	Product catalog, caching, logs, time-series, flexible data
Scalability	Vertical + horizontal (modern SQL scales well)	Horizontal by design (especially Cassandra, DynamoDB)
When to Choose	Data is structured, predictable, relational	Schema keeps changing, high write load, nested data
Examples	PostgreSQL, MySQL, MariaDB, SQL Server	MongoDB, Redis, DynamoDB, Cassandra, Neo4j
Role in Modern Systems	Source of truth, reliable state	Specialized workloads (cache, logs, documents)
Modern Overlap	Supports JSON, full-text search, KV patterns	Many support SQL-like queries & transactions

Why SQL vs NoSQL is NOT a meaningful debate anymore

Modern databases have evolved so much that
the old boundaries have become blurry.

1. SQL databases now support NoSQL features

PostgreSQL can store:

JSON
JSONB
Key-value style storage
Document-style operations
Full-text search

Many companies store entire documents inside SQL.

So SQL is no longer "strict tables only."

2. NoSQL databases now support SQL-like querying

Modern NoSQL systems offer:

Aggregations
Indexing
Joins (limited but available)
Transactions (MongoDB has ACID transactions now)

So NoSQL is no longer "schema-lite only."

3. Real production systems use BOTH

Almost every modern architecture uses:

SQL as the source of truth (correct, reliable)
NoSQL for speed or special workloads

Example:

SQL for orders

Redis for caching

Cassandra for logs

MongoDB for product catalog

Postgres for payments

This mix is normal and expected

PostgreSQL is powerful enough to handle many NoSQL-style workloads
— JSON documents, key-value patterns, and full-text search.
That's why it's a great general-purpose default."

"NoSQL databases still exist because they are optimized for very specific extremes:
Redis for ultra-fast caching,
Cassandra for massive write-heavy time-series data,
and Neo4j for deep graph traversal.
Postgres can do these,
but it won't match a purpose-built NoSQL system at those limits."

"For high-workload document use-cases, Postgres JSONB works well
until you hit true extremes like huge documents, millions of writes per second,
or constantly changing schemas. At that point, something like MongoDB is a better fit."

"And if your current system is already using SQL and it's working well,
you typically stick with SQL unless the workload clearly demands a specialized NoSQL engine."

"So it isn't SQL vs NoSQL anymore.
It's simply: what does the workload require? Modern systems use both.