[Relational v/s Non-Relational]

1. Data Model

The **data model** of a database defines how data is stored, organized, and related.

SQL

SQL databases use a **relational data model** where data is stored in **tables** (often referred to as relations).

Each table has **rows** (tuples) representing individual records, and **columns** representing attributes of those records.

The **primary key** uniquely identifies each record, while **foreign keys** link tables together, allowing for relational queries.

Example:

Let's consider a **user management system** with two tables: `Users` and `Orders`. The `Users` table contains user details, and the `Orders` table stores order details linked to specific users.

Users Table

UserId	Name	Email	Age
1	John	john@email.com	28
2	Mike	mike@email.com	31
3	Ron	ron@email.com	26

Orders Table

OrderId	UserId	Product	Price
101	1	Laptop	$1200
102	2	Smartphone	$800
103	3	Headphones	$150

The **UserID** in the `Orders` table is a **foreign key** that references the `Users` table, establishing a relationship between users and their orders.

This structured approach is ideal for applications requiring **complex queries** and **joins** between tables.

NoSQL databases use **flexible, non-relational** data models, allowing for various ways of storing and managing data.

a) Key-Value Model (e.g., Redis)

The key-value model is the simplest NoSQL model, where data is stored as **key-value** pairs. This model works well for applications that need **fast lookups** by a unique key.

For the same **user management system**, user data can be stored as key-value pairs where the key is the `UserID`, and the value is the associated user information.

```
Key: 1
Value: { "name": "John", "email": "john@email.com", "age": 28 }

Key: 2
Value: { "name": "Mike", "email": "mike@email.com", "age": 31 }
```

This model is very efficient for **simple lookups**, but it doesn't support complex querying or relationships between data.

b) Document Model (e.g., MongoDB)

In the document model, data is stored as **documents** in formats such as **JSON** or **BSON**.

Each document contains a **unique identifier** (**key**) and a set of **key-value pairs** (**attributes**). Documents can have varying structures, making the document model schema-less or flexible.

Let's model the same **user management system** using a document database.

```
{
  "_id": 1,
  "name": "John",
  "email": "john@email.com",
  "age": 28,
  "orders": [
    {
      "orderId": 101,
      "product": "Laptop",
      "price": 1200
    },
    {
      "orderId": 104,
      "product": "Smartphone",
      "price": 800
    }
  ]
}
```

In this document model, each user document contains an embedded array of orders, allowing for hierarchical storage within a single document.

c) Column-Family Model (e.g., Cassandra)

In the column-family model, data is organized into rows and columns, but unlike the relational model, each row can have a **variable number of columns.** It is optimized for **fast querying** and **large-scale distributed storage.**

d) Graph Model (e.g., Neo4j)

In the graph model, data is stored as **nodes**, **edges**, and **properties.** This model is ideal for applications where data relationships are complex and highly interconnected (e.g., social networks).

2. Schema

SQL

In SQL databases, the schema must be **defined upfront** before inserting any data.

Each table has a specific set of columns with defined data types, constraints, and relationships. The database enforces this schema, ensuring that every row adheres to the predefined structure.

This **schema enforcement** ensures data integrity, making SQL databases ideal for applications where consistency and accuracy are critical.

However, this rigidity can make it challenging to adapt to changing requirements.

In SQL databases, changing the schema can be a complex process.

If you need to add a new column, modify a data type, or change relationships, it often requires **schema migrations**.

This can lead to **downtime** or careful planning in production systems to avoid disruptions.

NoSQL

In NoSQL databases, there is **no fixed schema** that must be defined upfront.

This allows for flexible and dynamic data structures, where different records can have different attributes.

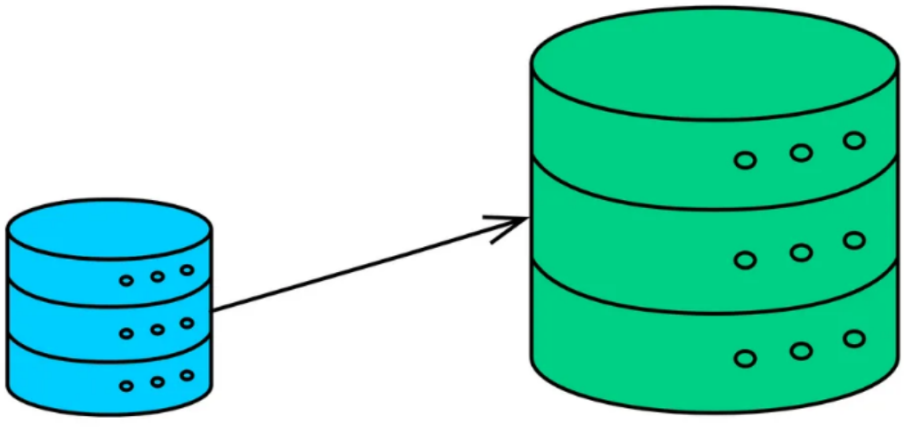
This flexibility makes NoSQL databases suitable for applications where data formats may evolve over time.

In NoSQL databases, schema changes are much simpler because the schema is **dynamic**. You can add new fields to individual records without affecting other records or requiring a schema migration.

3. Scalability

SQL

SQL databases are typically designed to scale **vertically** (also known as **scale-up**).



blog.algomaster.io

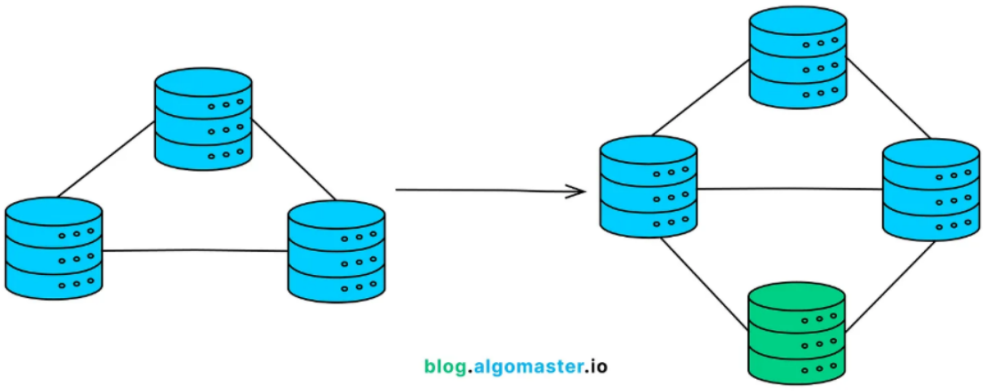
This means improving performance and capacity by adding **more power** (e.g., CPU, RAM, or storage) to a single server.

This approach works well for moderate loads but becomes limiting when the application scales to high levels of traffic or data growth.

SQL databases rely on maintaining **ACID** (Atomicity, Consistency, Isolation, Durability) properties, which makes horizontal scaling challenging due to the complexity of distributed transactions and joins.

NoSQL

NoSQL databases are designed to scale **horizontally** (also known as **scale-out**).



This means increasing capacity by adding **more servers** or nodes to a distributed system.

This distributed architecture allows NoSQL databases to handle massive volumes of data and high traffic loads more efficiently.

Each node handles a portion of the data, allowing for better load distribution and fault tolerance.

④ Query Language

SQL → proper structured language

NOSQL → No proper standard language.

5. Transactions

SQL → Have Robust support for ACID

NOSQL → Doesn't prioritize ACID for transactions.

Instead, many NoSQL databases follow the **BASE** model:

- **Basically Available**: The system guarantees availability, meaning that data can always be read or written, even if some nodes in the distributed system are unavailable.
- **Soft state**: The system may be in a temporarily inconsistent state, but eventual consistency will be reached over time.
- **Eventually consistent**: Over time, the system will become consistent, though it may not happen immediately. This trades immediate consistency for higher availability.

The BASE model is designed for scenarios where strict consistency is not required, and **performance** and **availability** are more important, such as real-time data analytics, social media platforms, or large-scale distributed applications.

While some NoSQL databases offer ACID-like features, they are generally less robust than those in SQL databases.

6. Performance

SQL

SQL databases are optimized to handle **complex queries** involving **multiple joins**, **aggregations**, and **transactions**.

For **small datasets**, SQL databases perform well, as the query optimizer can efficiently execute joins and filter data.

As the dataset grows, **performance may degrade** due to the complexity of joining large tables, especially if **indexing** is not optimized.

Their performance can be excellent for **read-heavy applications** with well-defined schemas and where data integrity is paramount.

However, they may struggle with **write-intensive** operations at scale without appropriate **indexing** and **optimization**.

Transaction overhead can also reduce performance when multiple queries are executed in a single transaction.

NoSQL

NoSQL databases are optimized to offer high performance at scale, especially for **large volumes of unstructured or semi-structured data.**

They prioritize **horizontal scalability** and are optimized for **high-throughput read/write operations**, making them ideal for real-time applications, big data, and large-scale distributed systems.

For **large datasets**, performance remains high as additional nodes are added to the cluster, distributing the workload.

NoSQL databases generally have **faster write performance** compared to SQL because:

- **Eventual consistency**: In distributed NoSQL systems, data does not have to be immediately consistent across all nodes, reducing the need for locks and increasing write speed.
- **Denormalized data model**: NoSQL databases often store related data together in a single document, which means fewer write operations compared to the normalized SQL model.

SQL

SQL databases are ideal for applications that require:

- **Structured data** with predefined schemas.
- **Complex queries** involving joins, aggregations, and transactions.
- **Strong consistency** and **ACID** (Atomicity, Consistency, Isolation, Durability) properties.
- **Relational data** where relationships between tables are important.

They are commonly used in industries like finance, healthcare, and government, where data integrity and relational structures are paramount.

NoSQL

NoSQL databases are ideal for use cases requiring:

- **Horizontal scalability** to handle large amounts of distributed data.
- **High-performance reads and writes** for real-time applications.
- **Flexible schema** to store unstructured or semi-structured data.
- **Eventual consistency** and high availability in distributed systems.

They are popular in industries like social media, IoT, and big data analytics, where flexibility and scalability are more important than strict consistency.