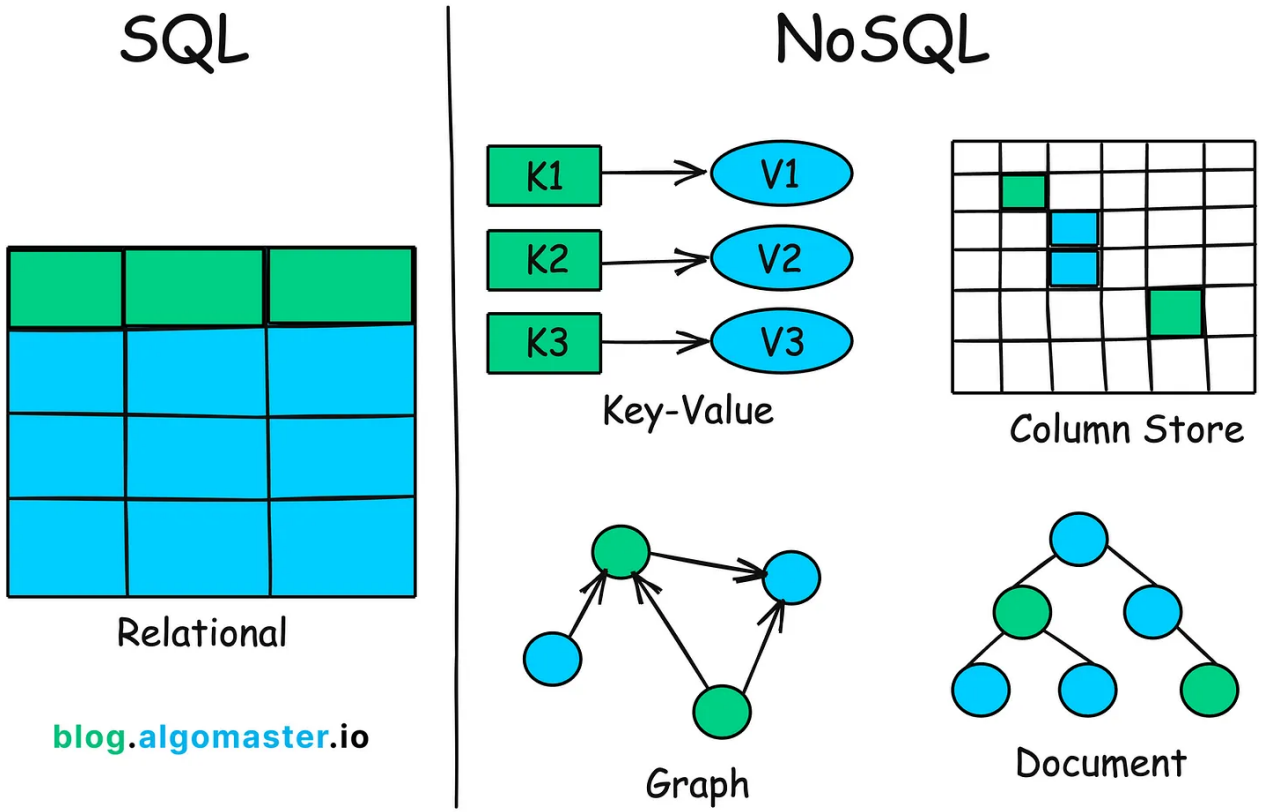
SQL vs NoSQL - 7 Key Differences You Must Know

ASHISH PRATAP SINGH

SEP 20, 2024

One of the biggest decisions we make while designing a system is **choosing between a relational (SQL) or non-relational (NoSQL) database.**



Both have their strengths and use cases, but they differ significantly in their approach to **data storage** and **retrieval**.

This article will explore **7 key differences** between SQL and NoSQL databases to help you understand which might be best suited for your specific needs.

If you’re enjoying this newsletter and want to get even more value, consider becoming a paid subscriber.

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all premium courses on algomaster.io.

---

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

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**.

Example:

Let’s assume we're building a **user activity tracking system** that stores the actions users take on a website, such as page views and purchases.

Each user has a unique `UserID`, and their activity (page views and purchases) is stored in a column-family.

Column Family: UserActivity

Row Key (UserID)	Page1	Page2	Purchase1	Purchase2
1	Viewed: Home Page	Viewed: Products	Product: Laptop ($1200)	Product: Phone ($800)
2	Viewed: Home Page	Viewed: Contact Us	Product: TV ($1500)	
3	Viewed: Products		Product: Headphones ($150)	

Row Key (**UserID**): The unique identifier for each user.

Page1, **Page2**, **Purchase1**, **Purchase2**: Columns representing the user's activity, which can vary from row to row.

- In **row 1**, user 1 has viewed two pages and made two purchases.
- In **row 2**, user 2 has viewed two pages and made one purchase.
- In **row 3**, user 3 has viewed one page and made one purchase.

Each user (row) can have a **variable number of columns**, and different users may have different activities stored in each row.

No predefined schema is required, which means new columns (such as additional page views or purchases) can be added dynamically for each user

This model allows for high write throughput and distributed storage but doesn't enforce strict relationships like the SQL relational model.

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).

In our **user management system**, users can be represented as nodes, and relationships between them (e.g., friendships or orders) can be represented as edges.

Graph Representation:

- Nodes: `Users`, `Orders`
- Edges: `PLACED_ORDER`

```
(John) --PLACED_ORDER--> (Laptop)
(Mike) --PLACED_ORDER--> (Smartphone)
(Ron) --PLACED_ORDER--> (Headphones)
```

In this model, querying relationships (e.g., finding all orders placed by a user) is highly efficient, especially for applications with complex interconnected data.

---

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.

Example: User and Orders Tables in SQL

Let's take the example of a **user management system**. In SQL, we first define the structure (schema) of the `Users` and `Orders` tables before adding data.

```
CREATE TABLE Users (
    UserID INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(100),
    Age INT
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    UserID INT,
    Product VARCHAR(100),
    Price DECIMAL(10, 2),
    FOREIGN KEY (UserID) REFERENCES Users(UserID)
);
```

In this schema:

- The `Users` table has fixed columns: `UserID`, `Name`, `Email`, and `Age`.

- The `Orders` table has fixed columns: `OrderID`, `UserID`, `Product`, and `Price`.
- **Foreign keys** are used to establish relationships between tables.

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.

Example: Adding a new column in SQL

```
ALTER TABLE Users ADD COLUMN Address VARCHAR(255);
```

This operation modifies the schema to include an `Address` field. Every record will need to be updated, and default values may be necessary for existing data.

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.

Example: User and Orders in a Document Database (e.g., MongoDB)

In a document-based NoSQL database, such as MongoDB, you store user and order data in a single document.

```
{
    "_id": 1,
    "name": "John",
    "email": "john@email.com",
    "age": 28,
    "orders": [
        {
            "orderId": 101,
            "product": "Laptop",
```

```
            "price": 1200
        },
        {
            "orderId": 104,
            "product": "Smartphone",
            "price": 800
        }
    ]
}
```

In another document, the structure can differ:

```
{
    "_id": 3,
    "name": "Ron",
    "email": "ron@email.com",
    "age": 26,
    "orders": [
        {
            "orderId": 103,
            "product": "Headphones",
            "price": 150
        }
    ],
```

```
    "loyaltyPoints": 500
}
```

In this flexible schema:

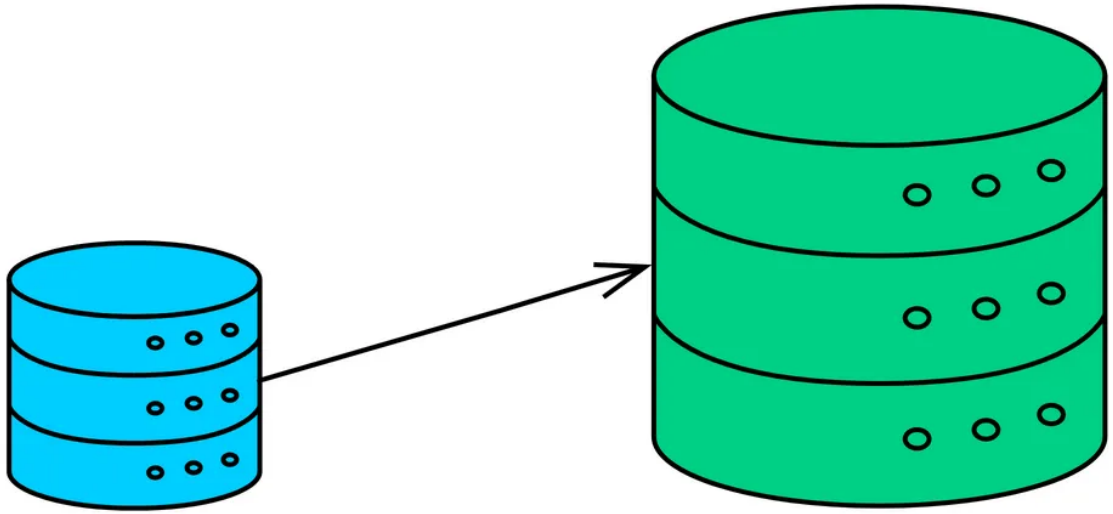
- The first document has user details along with two orders.
- The second document has user details, one order, and an additional attribute (`loyaltyPoints`), which is not present in the first document.

---

3. Scalability

SQL

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



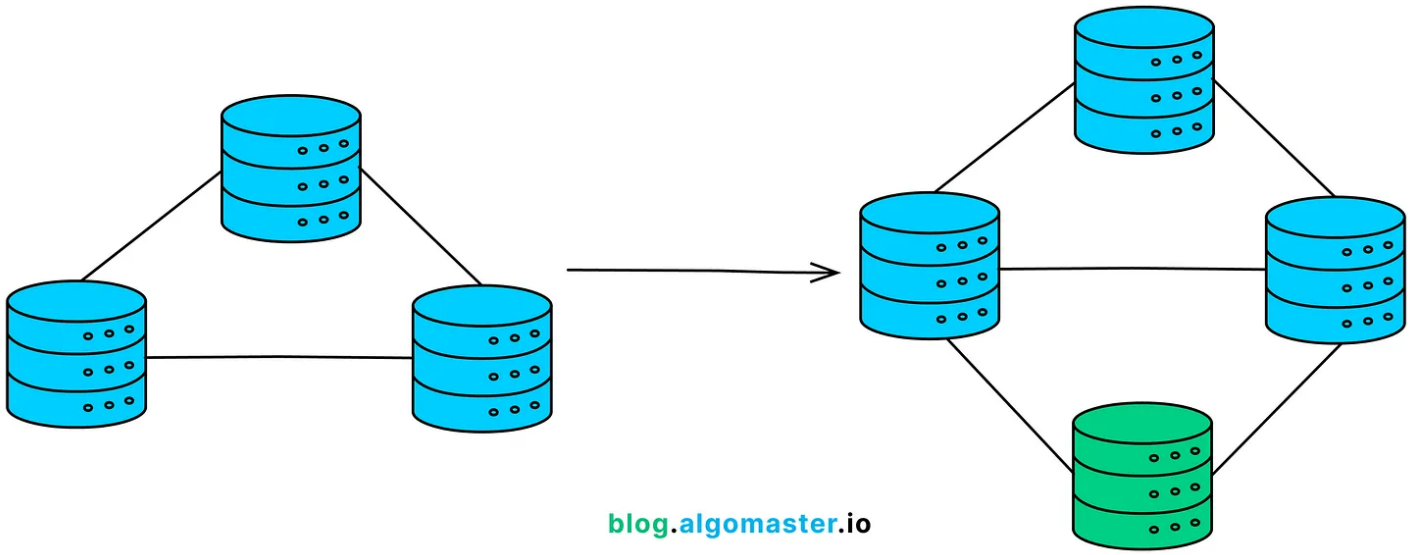
This means improving performance and capacity by adding **more power** (e.g., CPU, RAM, or storage) to a single server.

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

One of the most significant differences between SQL and NoSQL databases is the **query language** used to interact with the data.

SQL

SQL (**Structured Query Language**) is the de facto standard language used to interact with relational databases to perform operations such as **data retrieval**, **insertion**, **update**, and **deletion**.

It is **declarative**, meaning you specify what data you want, and the database engine determines how to retrieve it.

SQL allows for powerful data retrieval, aggregation, filtering, and manipulation, making it ideal for handling complex relationships between tables in a relational database.

Example:

Fetch users above age 25 and their orders

```
SELECT Users.Name, Users.Email, Orders.Product, Orders.Price
FROM Users
JOIN Orders ON Users.UserID = Orders.UserID
WHERE Users.Age > 25;
```

In this query:

- The `JOIN` operation combines the `Users` and `Orders` tables based on the `UserID`.
- The `WHERE` clause filters out users whose age is less than 25.
- The result is a list of users and their corresponding orders.

NoSQL

NoSQL databases do not have a standard query language. Each NoSQL database may have its query syntax or API, depending on its data model.

While this can provide more flexibility, it also means a steeper learning curve and potential limitations in querying capabilities.

Example:

In a document-based NoSQL database like **MongoDB**, data is stored in **JSON-like documents**. MongoDB provides its own query language, which uses JSON syntax to query the documents.

Queries in MongoDB allow for powerful filtering, sorting, and aggregation, but they work on the individual document level rather than across multiple tables.

In MongoDB, users and their orders are stored together in a single document.

Example:

Fetch users above age 25 and their orders

```
db.Users.find(
    { "age": { $gt: 25 } },
    { "name": 1, "email": 1, "orders.product": 1, "orders.price": 1 }
);
```

In this MongoDB query:

- The `find` operation searches for documents where the user's age is greater than 25.

- The second parameter is a projection specifying which fields to return: the user's name, email, and the `product` and `price` of their orders.

Same query in a **graph database** may look like:

```
MATCH (u:User)-[:PLACED_ORDER]->(o:Order)
WHERE u.age > 25
RETURN u.name, o.product, o.price;
```

---

5. Transaction Support

Transactions in databases ensure that a series of operations are executed in a reliable, consistent manner.

Transactions are particularly important in applications where multiple operations must be completed together, such as transferring money between bank accounts or ensuring that a group of database updates either all succeed or all fail.

SQL

SQL databases are known for their robust support of **ACID transactions**.

- **Atomicity**: A transaction is treated as a single, indivisible unit. Either all operations within the transaction succeed, or none of them are applied (all-or-nothing).
- **Consistency**: Transactions take the database from one valid state to another, maintaining all predefined rules such as constraints, triggers, and data types.
- **Isolation**: Multiple transactions can occur concurrently without interfering with each other, ensuring that one transaction's intermediate state is not visible to other transactions.
- **Durability**: Once a transaction is committed, its results are guaranteed to persist, even in the event of a system crash.

This makes SQL databases ideal for applications where **data integrity** and **consistency** are critical, such as financial systems.

Example:

Bank Transfer in SQL

Consider a banking system where you need to transfer $500 from User A's account to User B's account.

This operation requires two steps: debit User A's account and credit User B's account.

Both steps must succeed or fail together to ensure the system remains consistent.

```
START TRANSACTION;

-- Debit User A's account
UPDATE Accounts SET balance = balance - 500 WHERE user_id = 'A';

-- Credit User B's account
UPDATE Accounts SET balance = balance + 500 WHERE user_id = 'B';

COMMIT;
```

If either the debit or credit operation fails (e.g., insufficient funds), the entire transaction will be rolled back, leaving both accounts unchanged.

NoSQL

NoSQL databases typically do not prioritize full ACID transactions due to the need for high **availability** and **scalability** in distributed environments.

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.

Example:

Cassandra Conditional Update (Lightweight Transaction):

Cassandra does not support full ACID transactions across multiple rows or tables. Instead, it offers **lightweight transactions** for operations requiring limited

consistency.

```
BEGIN TRANSACTION;
UPDATE Users SET balance = balance - 500 WHERE user_id = 'A' IF balance
>= 500;
UPDATE Users SET balance = balance + 500 WHERE user_id = 'B';
APPLY BATCH;
```

Cassandra ensures atomicity for a batch of updates but does not support complex multi-table transactions like 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.

---

7. Use Cases

The choice between SQL and NoSQL databases often depends on the specific use case, as each type of database excels in different scenarios.

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.

Conclusion

Here's a final table comparing **SQL** and **NoSQL** databases across different key aspects:

Aspect	SQL (Relational)	NoSQL (Non-Relational)
Data Model	Structured, table-based with fixed schemas and relationships	Flexible data models (document, key-value, column-family, graph)
Schema	Rigid; must be predefined with strict structure	Schema-less or flexible; allows dynamic and unstructured data
Scalability	Vertical scaling (scale-up)	Horizontal scaling (scale-out) across distributed nodes
Consistency	Strong consistency with ACID transactions	Eventual consistency, though tunable in some systems (e.g., MongoDB)
Query Language	SQL (standardized and powerful for complex queries)	Varies (e.g., MongoDB query language, Cassandra Query Language)
Transaction Support	Full ACID (Atomicity, Consistency, Isolation, Durability)	BASE model (Basically Available, Soft state, Eventually consistent); some support ACID in limited contexts
Performance	Optimized for complex queries and relationships, slower for large-scale writes	Optimized for high-throughput reads/writes, especially in distributed systems
Use Cases	Best for structured data and applications needing strong consistency (e.g., financial systems, ERP, CRM)	Best for unstructured/semi-structured data and large-scale distributed systems (e.g., big data, real-time analytics, social media)
Scaling Challenges	Difficult to shard or partition across servers	Built to shard and distribute data easily

Data Integrity	High; enforces strong data integrity through constraints and relationships	Varies; integrity is often managed at the application level
Joins and Relationships	Supports complex joins between tables and foreign key relationships	Relationships are either embedded (documents) or handled differently (e.g., graph databases)
Examples	MySQL, PostgreSQL, Oracle, SQL Server	MongoDB, Cassandra, DynamoDB, Redis, Neo4j

To summarize, both SQL and NoSQL databases have their strengths and weaknesses, and the choice between them depends on your application's specific needs.

If your application requires structured data, complex queries, and transaction management, an SQL database is likely the best choice.

However, if your application demands scalability, flexibility, and the ability to handle unstructured data, a NoSQL database may be more suitable.

Thank you for reading!

If you found it valuable, hit a like ❤️ and consider subscribing for more such content every week.

If you have any questions or suggestions, leave a comment.

This post is public so feel free to share it.

P.S. If you're enjoying this newsletter and want to get even more value, consider becoming a **paid subscriber**.

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all **premium courses** on **algomaster.io**.

There are group discounts, gift options, and referral bonuses available.

Checkout my **Youtube channel** for more in-depth content.

Follow me on **LinkedIn**, **X** and **Medium** to stay updated.

Checkout my **GitHub repositories** for free interview preparation resources.

I hope you have a lovely day!

See you soon,
Ashish

193 Likes • 12 Restacks

← Previous

Next →

Discussion about this post

Comments Restacks

Write a comment...

Soumya Ranjan Mishra 23 Aug

Liked by Ashish Pratap Singh

The best resource in the world to learn.

LIKE (1) REPLY SHARE

adilreza 24 Sep 2024

Liked by Ashish Pratap Singh

best!

LIKE (1) REPLY SHARE

3 more comments...