Q1. What is Impedance Mismatch? Explain with example.

- Impedance mismatch is a term used in computer science to describe the problem that arises when two systems or components that are supposed to work together have different data models, structures, or interfaces that make communication difficult or inefficient.(in general)
- Relational databases provide many advantages, but they are by no means perfect.
- For application developers, the biggest frustration has been what's commonly called the **impedance mismatch: the difference between the relational model and the in-memory data structures.**
- Impedance mismatch in databases refers to the difference between how data is represented in a database (tables, rows, columns) and how data is represented in programming languages (objects, classes, structures).
- Databases use the relational model → data is stored in tables (rows & columns).
- Programming languages (Java, Python, C#, etc.) use the object-oriented model → data is stored in objects (attributes & methods).
- This mismatch creates extra effort when moving data between the two worlds.

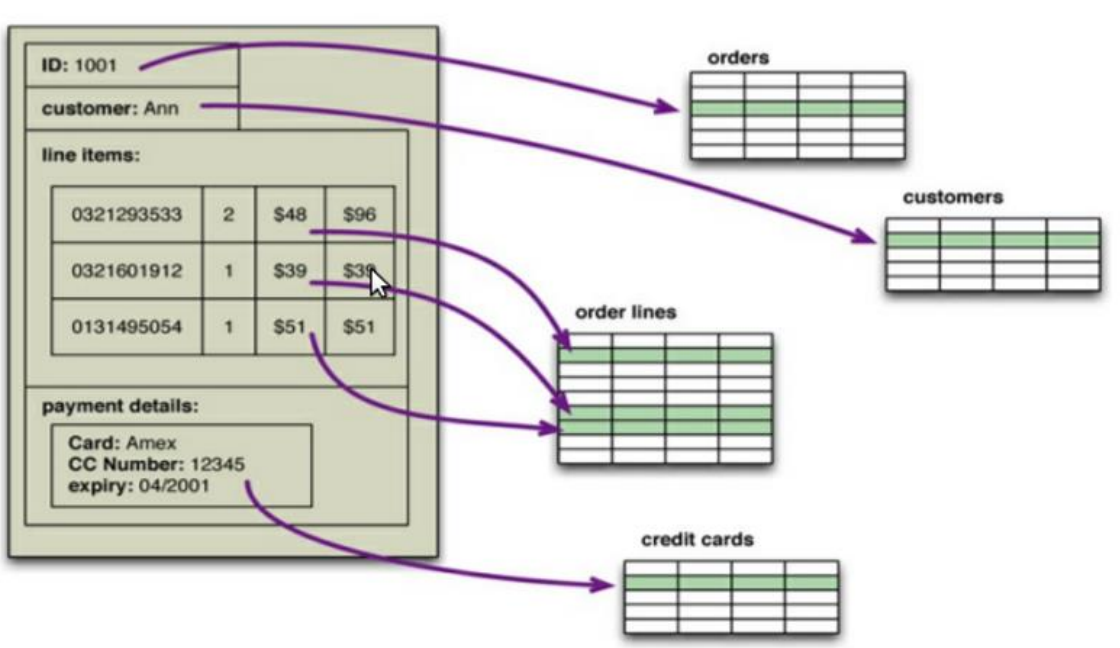


Figure 1.1. An order, which looks like a single aggregate structure in the UI, is split into many rows from many tables in a relational database

Q2. Explain Aggregate Data Models with example.

- Aggregate data models → group related data into a **single unit (aggregate)** for easy storage & retrieval in NoSQL.
- Each aggregate treated as **atomic** → stored/queried together.
- Each NoSQL solution has a different model that it uses, which we put into four categories widely used in the NoSQL ecosystem: **key-value, document, column-family, and graph**. Of these, the first three share a common characteristic of their data models which we will call **aggregate orientation.**
- Benefits:
    1. Reduces JOINs → faster queries.
    2. Flexible schema → can evolve over time.
    3. Naturally fits into JSON, BSON formats.
- Types:
    - **Document databases:** MongoDB stores entire customer order as one document {custId, orderItems, address}.
    - **Key-Value stores:** Value contains complex aggregate (JSON/XML).
    - **Column-family stores:** Cassandra groups related columns in families.

```json
{
  "orderId": 101,
  "customer": "John",
  "items": [
    {"product":"Laptop","qty":1},
    {"product":"Mouse","qty":2}
  ]
}
```

Q3. Explain Sharding and Replication concepts in detail.

Distribution models

- The primary driver of interest in NoSQL has been its ability to run databases on a large cluster. As data volumes increase, it becomes more difficult and expensive to scale up—buy a bigger server to run the database on
- A more appealing option is to scale out—run the database on a cluster of servers
- Aggregate orientation fits well with scaling out because the aggregate is a natural unit to use for distribution.
- Broadly, there are two paths to data distribution: **replication and sharding.**
    - Replication takes the same data and copies it over multiple nodes.
    - Sharding puts different data on different nodes.
- Replication and sharding are orthogonal techniques: You can use either or both of them.
    - **Single-server**
    - **Master-slave replication**
    - **Sharding**
    - **Peer-to-peer replication.**

Single server

- The first and the simplest distribution option is the one we would most often recommend—no distribution at all. Run the database on a single machine that handles all the reads and writes to the data store.
- Although a lot of NoSQL databases are designed around the idea of running on a cluster, it can make sense to use NoSQL with a single-server distribution model if the data model of the NoSQL store is more suited to the application
    - Graph databases are the obvious category here—these work best in a single-server configuration
- If your data usage is mostly about processing aggregates, then a single-server document or key-value store may well be worthwhile because it's easier on application developers.

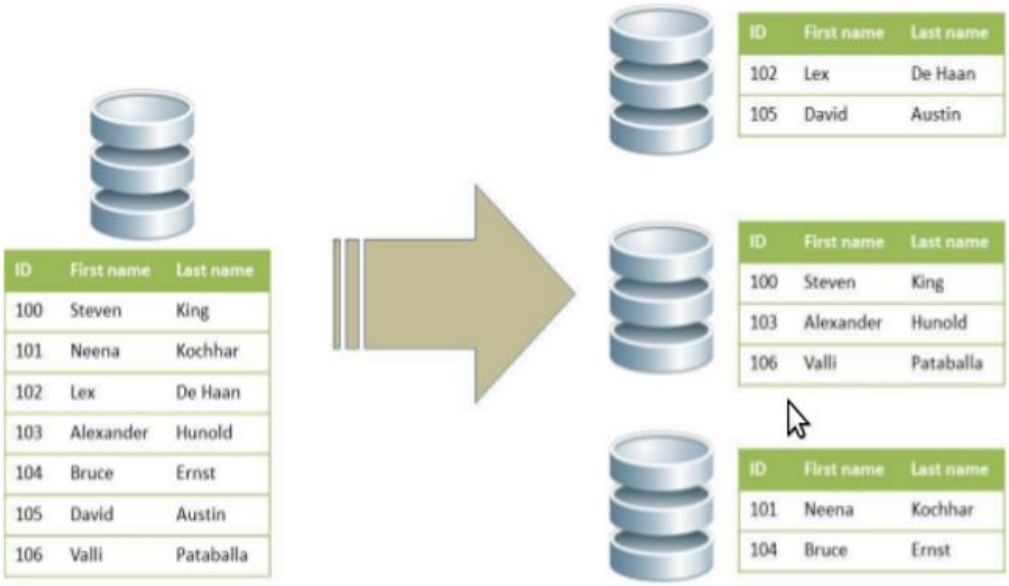
Sharding

- Often, a busy data store is busy because different people are accessing different parts of the dataset
- In these circumstances we can support horizontal scalability by putting different parts of the data onto different servers—a technique that's called sharding (see Figure 4.1).
- Breaks data into smaller **chunks (shards)** stored across multiple servers.

- Benefits:

    1. Improves performance and scalability → add more servers.
    2. Better load balancing.
    3. Handles **Big Data** efficiently.

- Many NoSQL databases offer **auto-sharding**, where the database takes on the responsibility of allocating data to shards and ensuring that data access goes to the right shard.
- Sharding is particularly valuable for performance because it can improve both read and write performance.
- If one node fails only that portion of data is affected.



ID	First name	Last name
100	Steven	King
101	Neena	Kochhar
102	Lex	De Haan
103	Alexander	Hunold
104	Bruce	Ernst
105	David	Austin
106	Valli	Pataballa

ID	First name	Last name
102	Lex	De Haan
105	David	Austin

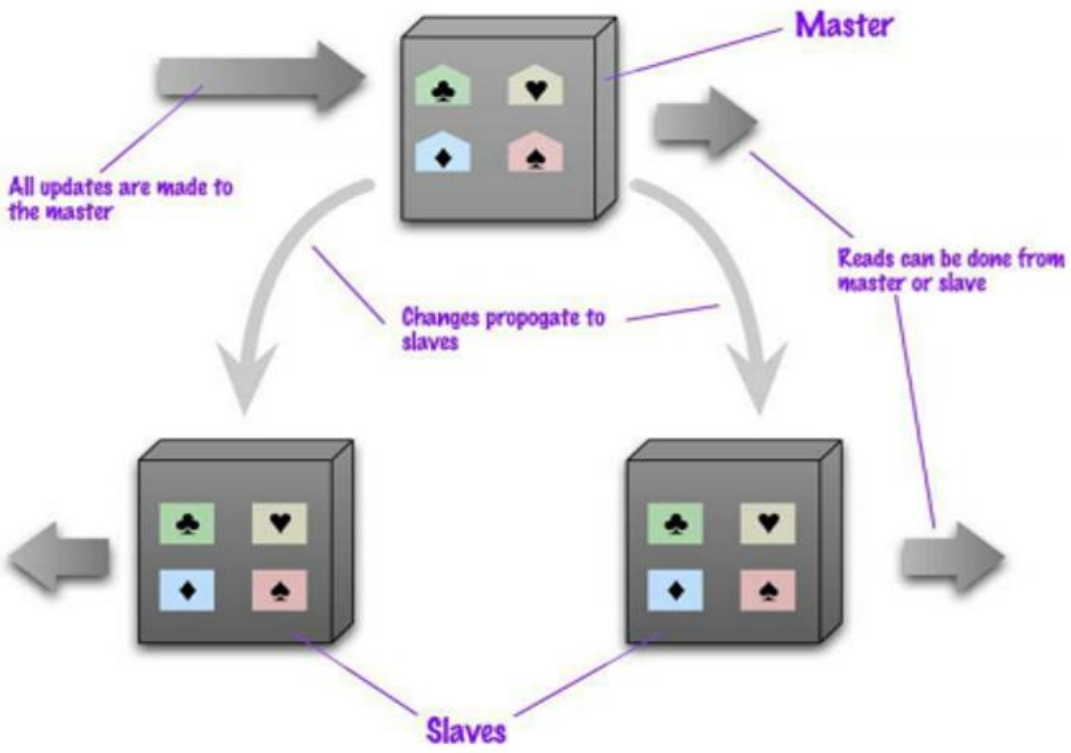
ID	First name	Last name
100	Steven	King
103	Alexander	Hunold
106	Valli	Pataballa

ID	First name	Last name
101	Neena	Kochhar
104	Bruce	Ernst

- **Replication:**
    - Copies of same data maintained on multiple servers.
        - Ensures **availability, fault tolerance**.
    - Types:
        1. Master-Slave → one primary handles writes, slaves handle reads.
        2. Peer-to-Peer → every node equal, all accept reads/writes.

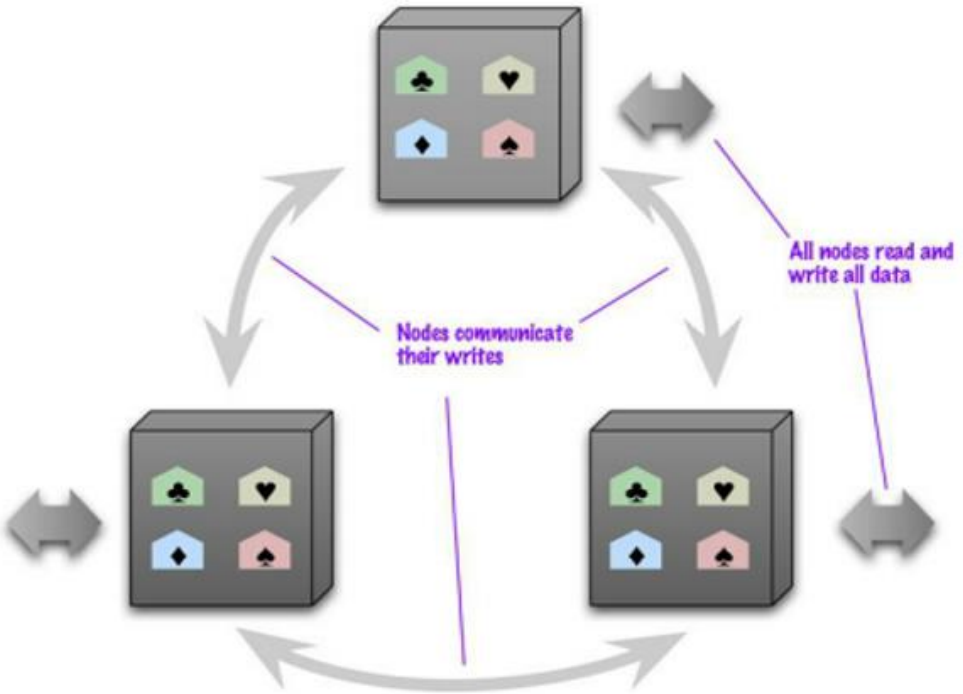
Master Slave Replication

- Data is replicated from master to slaves
- With master-slave distribution, you replicate data across multiple nodes.
- One node is designated as the master, or primary.
- This master is the authoritative source for the data and is usually responsible for processing any updates to that data.
- The other nodes are slaves, or secondaries. A replication process synchronizes the slaves with the master.
- You can scale horizontally to handle more read requests by adding more slave nodes and ensuring that all read requests are routed to the slaves.
- Masters can be appointed manually or automatically.
- Manual appointing typically means that when you configure your cluster, you configure one node as the master



Peer to Peer Replication

- Master-slave replication helps with read scalability but doesn't help with scalability of writes.
- It provides resilience against failure of a slave, but not of a master.
- Essentially, the master is still a bottleneck and a single point of failure
- Peer-to-peer replication (see Figure 4.3) attacks these problems by not having a master.
- All the replicas have equal weight, they can all accept writes, and the lossof any of them doesn't prevent access to the data store.



Master-Slave Replication

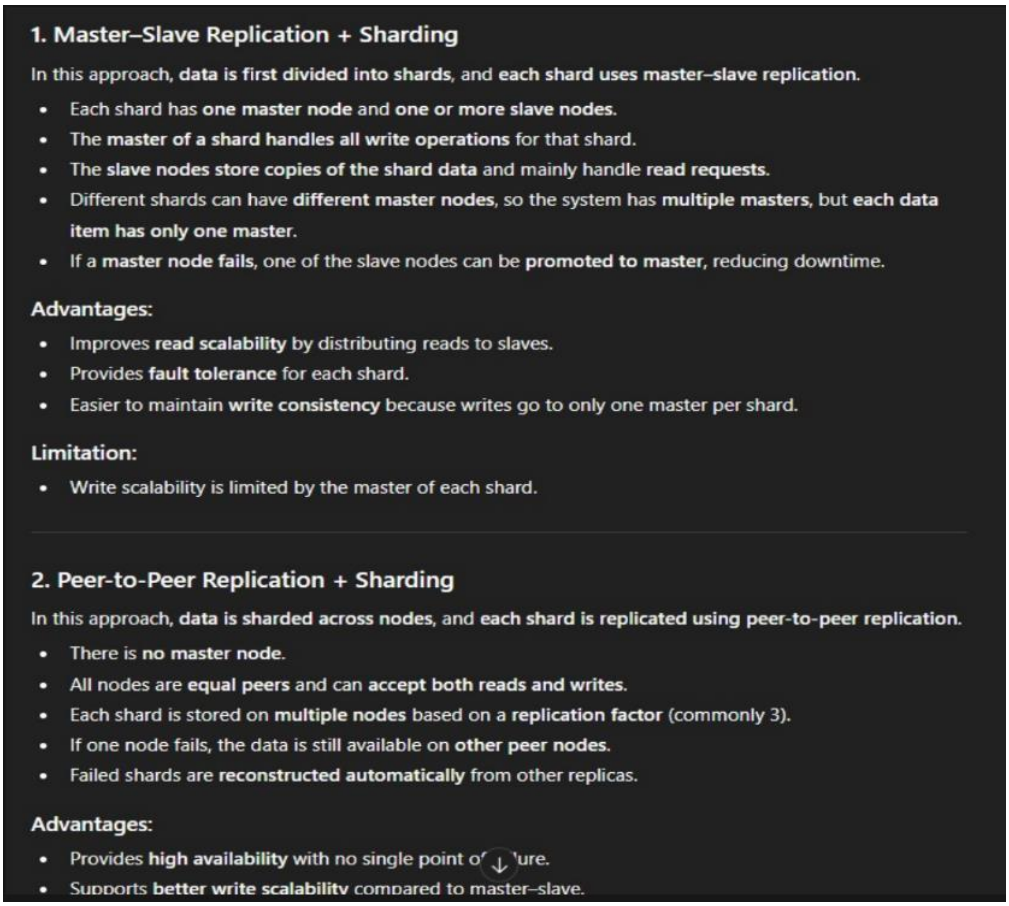
- Master handles all writes & slaves cannot write.
- Master is a single point of failure
- Provides strong consistency as master controls writes
- No conflicts, as only the master writes

VS

Peer-to-Peer Replication

- All peers can handle both reads and writes.
- No single point of failure
- Can have temporary changes between peers.
- Requires mechanisms to handle write conflicts between peers.

Combining Sharding and Replication



1. Master–Slave Replication + Sharding

In this approach, **data is first divided into shards**, and **each shard uses master–slave replication**.

- Each shard has **one master node** and **one or more slave nodes**.
- The **master of a shard handles all write operations** for that shard.
- The **slave nodes store copies of the shard data** and mainly handle **read requests**.
- Different shards can have **different master nodes**, so the system has **multiple masters**, but **each data item has only one master**.
- If a **master node fails**, one of the slave nodes can be **promoted to master**, reducing downtime.

Advantages:

- Improves **read scalability** by distributing reads to slaves.
- Provides **fault tolerance** for each shard.
- Easier to maintain **write consistency** because writes go to only one master per shard.

Limitation:

- Write scalability is limited by the master of each shard.

---

2. Peer-to-Peer Replication + Sharding

In this approach, **data is sharded across nodes**, and **each shard is replicated using peer-to-peer replication**.

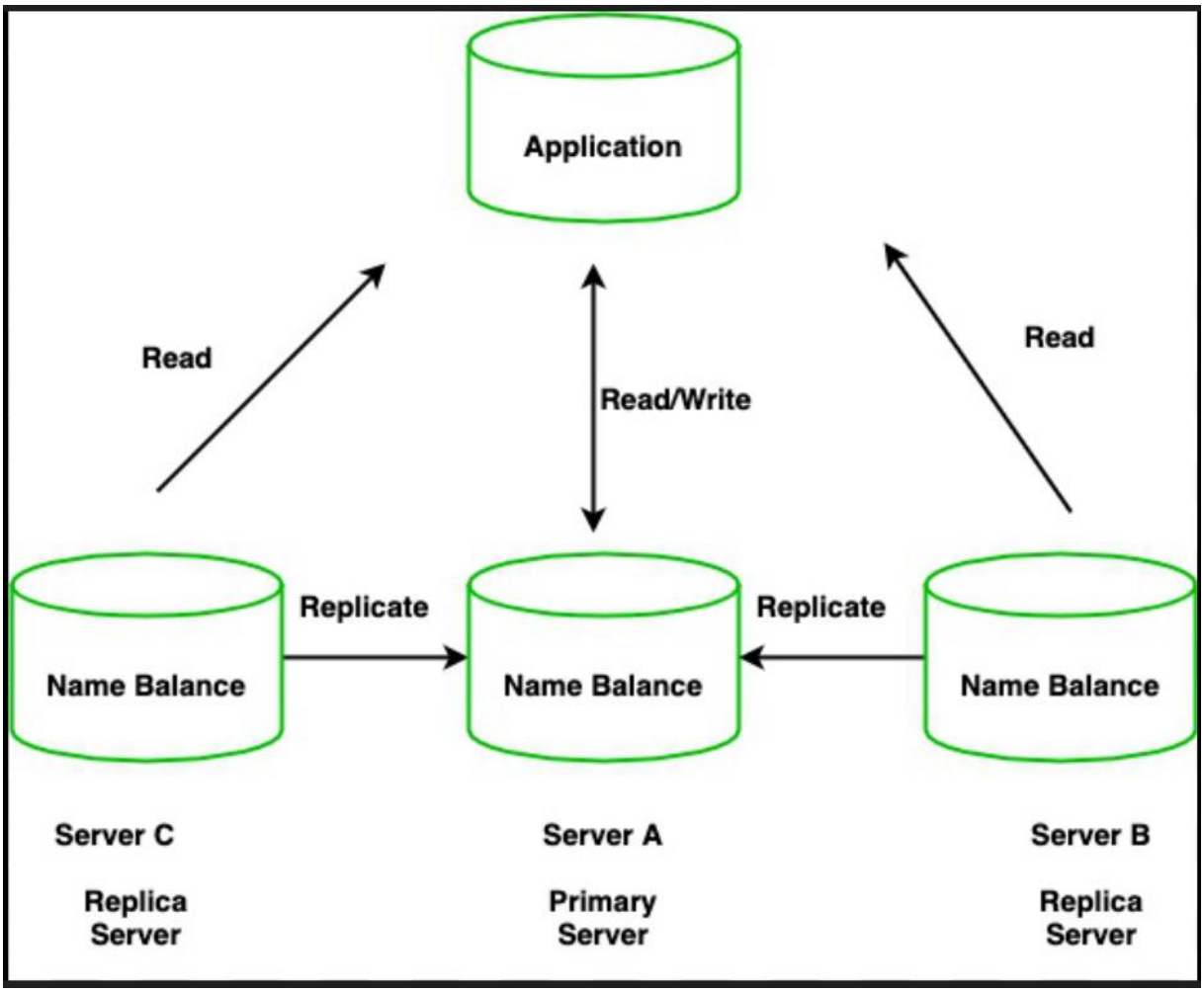
- There is **no master node**.
- All nodes are **equal peers** and can **accept both reads and writes**.
- Each shard is stored on **multiple nodes** based on a **replication factor** (commonly 3).
- If one node fails, the data is still available on **other peer nodes**.
- Failed shards are **reconstructed automatically** from other replicas.

Advantages:

- Provides **high availability** with no single point of ↓ ure.
- Supports **better write scalability** compared to master–slave.

Benefits of Combination

1. **Scalability** – Sharding distributes data, enabling millions of records to be stored across servers.
2. **Reliability** – Replication ensures data is not lost during node failures.
3. **Load Balancing** – Queries spread across shards and replicas.
4. **High Availability** – Failover supported via replicas.



Q5. Explain the concept of sharding in NoSQL databases. How does it help in managing large datasets?

- In NoSQL, sharding is **built-in** (auto-sharding).
- Database automatically partitions data across nodes.
- Application doesn't need to know which node stores which data.
- **Helps manage large datasets by:**
    1. **Load distribution** – avoids single-node bottleneck.
    2. **Performance boost** – queries limited to relevant shard.
    3. **Elastic scalability** – servers can be added/removed dynamically.
    4. **Cost-effective** – runs on commodity hardware.

Q8. What is Update Consistency? Explain with example.

Answer:
Update consistency ensures that **after an update operation, all subsequent reads reflect the latest value of the data**.

1. Definition

- When a write/update operation is acknowledged, every future read should return the updated value, regardless of which replica the read comes from.

2. Importance

- Prevents stale data reads.
- Ensures application correctness.
- Critical in domains like:
    - **Banking:** Balance updates must be consistent.
    - **E-commerce:** Stock reductions must reflect instantly.

3. Example

- We'll begin by considering updating a telephone number. Coincidentally, **Martin and Pramod** are looking at the company website and notice that the phone number is out of date.
- Implausibly, they both have update access, so they both go in at the same time to update the number.
- To make the example interesting, we'll assume they update it slightly differently, because each uses a slightly different format.
- This issue is called a **write-write conflict**: two people updating the same data item at the same time.
- When the writes reach the server, the server will **serialize** them—decide to apply one, then the other.
- Let's assume it uses alphabetical order and picks Martin's update first, then Pramod's.
- Martin's update would be applied and immediately overwritten by Pramod's.
- In this case Martin's is a **lost update**.

- Here the lost update is not a big problem, but often it is. We see this as a failure of consistency because Pramod's update was based on the state before Martin's update, yet was applied after it.

4. Challenges

- Hard to maintain in distributed systems due to network delays and replication lag.
- Often traded off in favour of availability (CAP theorem).

Q9. What is Eventual Consistency? Explain.

Answer:
Eventual consistency is a **weaker form of consistency** used by many NoSQL systems to balance performance and availability.

1. Definition

- Guarantees that if no new updates are made, **all replicas will eventually converge to the same value**.
- Does not guarantee immediate consistency.

2. Importance

- Improves availability and system responsiveness.
- Users may see slightly stale data, but system guarantees convergence.

3. Example

- A user posts a comment on a social network.
- Some servers may display it instantly, while others may take seconds.
- Eventually, all servers reflect the comment.

4. Systems Using Eventual Consistency

- Amazon DynamoDB, Cassandra, CouchDB, Riak.

Q11. Explain Read Consistency in NoSQL databases and the challenges in achieving it.

Answer:

Read consistency ensures that **data read by clients is correct and up-to-date** with recent writes.

1. **Levels of Read Consistency**

- **Strong Consistency:**
    - Every read returns the latest value.
    - Example: Google Spanner.
- **Eventual Consistency:**
    - Reads may be stale but eventually converge.
    - Example: DynamoDB, Cassandra.
- **Read-Your-Writes Consistency:**
    - A client always sees its own updates.
    - Example: Social media posts → you always see your latest post.
- **Causal Consistency:**
    - Preserves cause-effect relationships.

2. Challenges

- **Network latency:** Delay in propagating updates to replicas.
- **Partition tolerance:** During network failure, replicas may diverge.
- **Load balancing:** Reads routed to stale replicas.
- **Concurrency:** Multiple clients updating same record may lead to anomalies.

3. Example

- Let's imagine we have an order with line items and a shipping charge.
- The shipping charge is calculated based on the line items in the order.
- If we add a line item, we thus also need to recalculate and update the shipping charge. In a relational database, the shipping charge and line items will be in separate tables.

- The danger of inconsistency is that Martin adds a line item to his order, Pramod then reads the line items and shipping charge, and then Martin updates the shipping charge.
- This is an inconsistent read or read-write conflict

Q13. What are NoSQL databases, and how do they differ from traditional relational databases?

Answer:

1. NoSQL Databases – Definition

- "Not Only SQL" → family of non-relational, schema-less, distributed databases.
- Handle **structured, semi-structured, and unstructured** data.
- NoSQL databases are non-relational and are designed to handle large volumes of data, high user loads, and flexible schema structures.

2. Key Characteristics

- Schema-less: Data can be stored in varied formats (JSON, BSON, etc.).
- Horizontal scalability: Easily spread across multiple servers. **Horizontal scalability** (also called **scale-out**) is the ability of a system to **increase capacity by adding more machines or nodes**, rather than upgrading the existing hardware.
- High performance: Optimized for high throughput and low latency.
- Support for unstructured, semi-structured, or structured data.
- BASE properties (Basically Available, Soft state, Eventual consistency).

Q14. Why did NoSQL databases emerge, and what problems do they aim to solve?

Answer:

1. Reasons for Emergence

- Explosion of **Big Data** (social media, IoT, e-commerce).
- Need for **scalability beyond RDBMS**.
- Flexible schema for fast-changing apps.
- Global systems demanding 24×7 **availability**.

2. Problems Solved by NoSQL

1. **Handling large volumes:** Can manage petabytes of data across clusters.
2. **Support for high velocity:** Can handle millions of requests per second.
3. **Schema flexibility:** JSON/XML storage, no rigid schema.
4. **High availability:** Through replication and partitioning.
5. **Complex data types:** Supports multimedia, nested structures.

Q15. What is NoSQL? Explain its advantages.

Answer:

1. **Definition**

- A type of database that provides **non-relational, distributed, and schema-flexible storage**.

2. **Advantages**

1. **Scalability:** Horizontal scaling using sharding.
2. **Flexibility:** NoSQL does not require any adherence to pre-defined schema.
3. **Relaxes the data consistency requirement:**

NoSQL databases have adherence to CAP theorem (Consistency, Availability, and Partition Tolerance). Most of the NoSQL databases compromise on consistency in favor **Performance:** Faster reads/writes compared to RDBMS for large datasets.

4. **Variety of data:** Supports text, JSON, logs, images, videos.
5. **Cost-effective:** Runs on commodity hardware and cloud platforms.
6. **Designed for Big Data:** Handles large volumes, variety, and velocity.
7. **Data can be replicated to multiple nodes and can be partitioned:**

   There are two terms that we will discuss here:

   A) Sharding

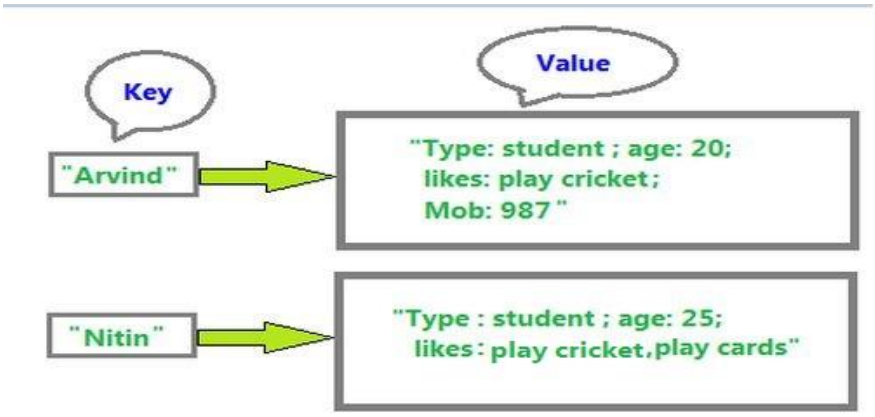
   B) Replication

8. Eventual consistency : Eventual consistency in NoSQL databases is a consistency model where updates to the database may not be visible to all nodes immediately, but given enough time (and no new updates), all nodes will eventually have the same data.

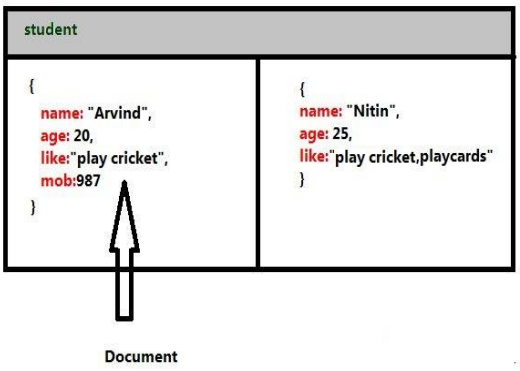
Q16. Explain the different types of NoSQL databases with example.

Answer:

1. **Key-Value Stores:**
    - The simplest type of NoSQL database is a key-value store.
    - It maintains a big hash table of keys and values.
    - Every data element in the database is stored as a **key-value** pair consisting of an attribute name (or "key") and a value. In a sense, a key-value store is like a relational database with only two columns: the key or attribute name (such as state) and the value (such as Alaska).
    - For example, Dynamo, Redis, Apache Cassandra

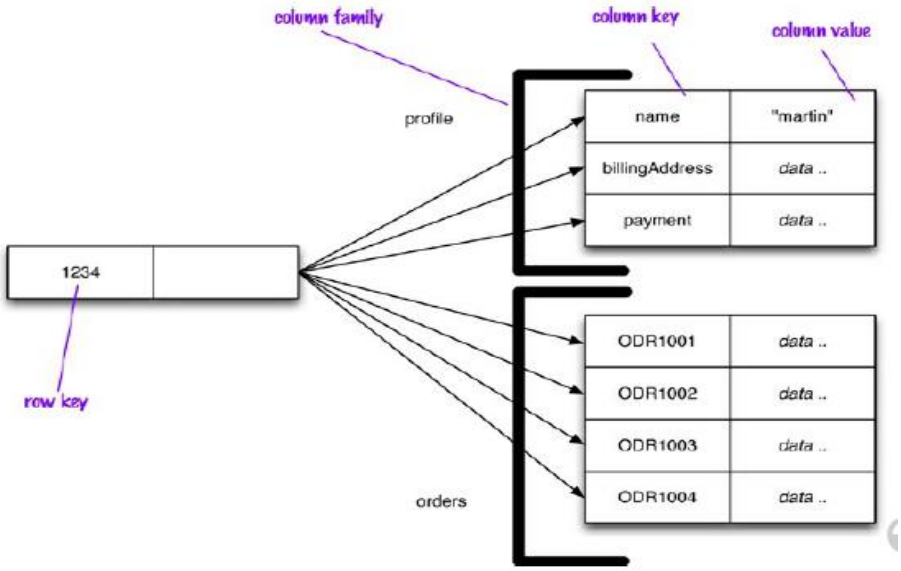


2. **Document Stores:**
    - It maintains data in collections constituted of documents.
    - A document database stores data in JSON, BSON , or XML documents.
    - In a document database, documents can be nested. Particular elements can be indexed for faster querying.
    - Example: MongoDB, CouchDB.



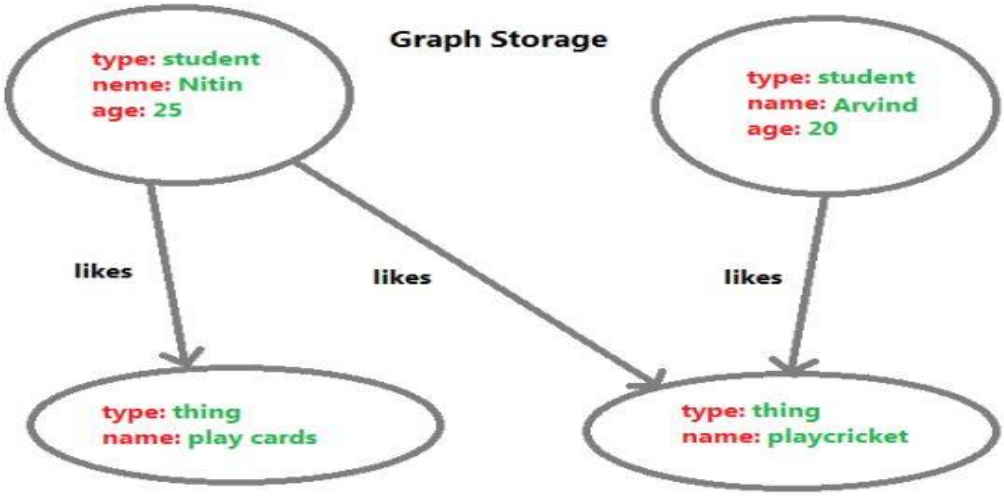
3. **Column-Family Stores:**

   - While a relational database stores data in rows and reads data row by row, a column store is organized as a set of columns.
   - This means that when you want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data. Columns are often of the same type and benefit from more efficient compression, making reads even faster.
   - Columnar databases can quickly aggregate the value of a given column
   - Example: Cassandra, HBase.



4. **Graph Databases:**

   - A graph database focuses on the relationship between data elements.
   - Each element is stored as a node (such as a person in a social media graph).
   - The connections between elements are called links or relationships.
   - They are also called network database. A graph stores data in nodes.
   - Example: Neo4j,infinite graph



Q18. Explain CAP theorem in detail.

Answer:

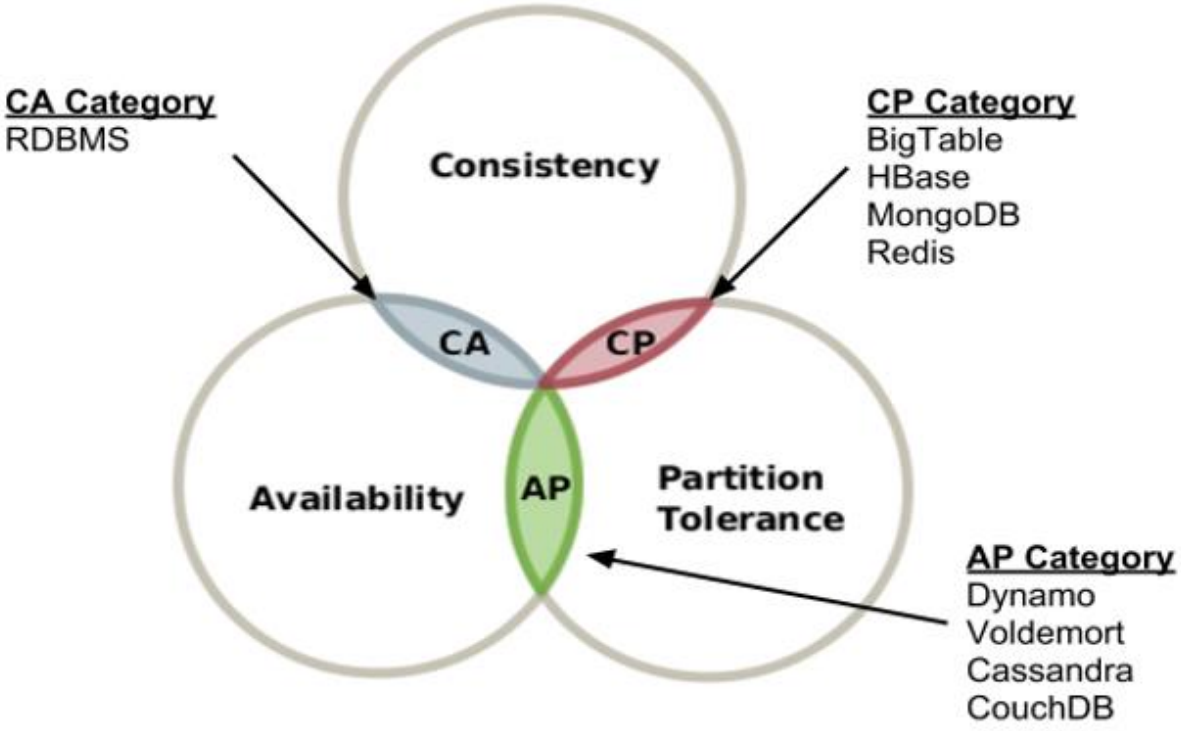
The basic statement of the CAP theorem is that, **given the three properties of Consistency, Availability, and Partition tolerance, you can only get two.**

1. Definition

- In distributed systems, it is **impossible to simultaneously provide**:
    - **Consistency (C):** Every read gets the latest write or an error.
    - **Availability (A):** Every request receives a response.
    - **Partition Tolerance (P):** System functions despite network failures.

2. Scenarios

- **CP systems (Consistency + Partition):** Prioritize correctness, may sacrifice availability. Example: HBase.
- **AP systems (Availability + Partition):** Prioritize uptime, may sacrifice consistency temporarily. Example: Cassandra, DynamoDB.
- **CA systems (Consistency + Availability):** Possible only in non-distributed databases.



3. Trade-off Example

In presence of network partitions, a distributed system must choose between:

- **Consistency (C)** or
- **Availability (A)**

Trade-off Explanation

- **If consistency chosen:** System rejects requests during partition to avoid stale data. → High correctness but reduced availability.
- **If availability chosen:** System serves requests even with stale data. → High uptime but weaker correctness.

Example

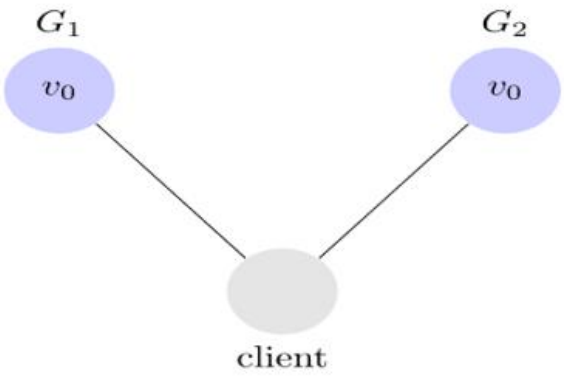
- Banking system (CP): Ensures correct balance but may deny transactions when partitioned.
- Social media (AP): Posts visible immediately, even if not consistent across all replicas.

4. Proof (Simplified)

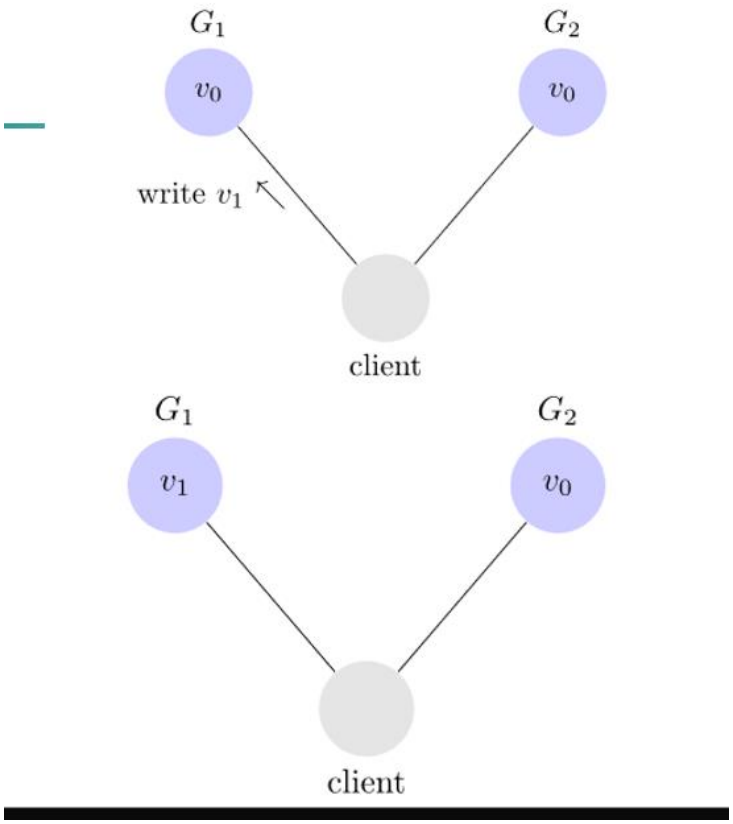
The Proof

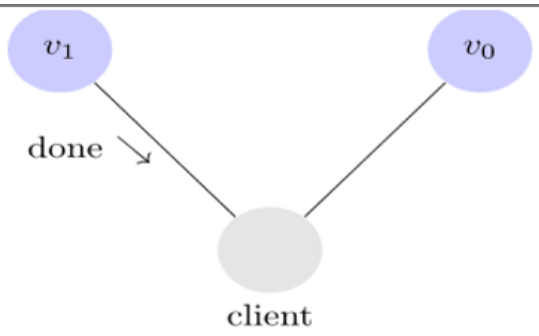
Now that we've acquainted ourselves with the notion of consistency, availability, and partition tolerance, we can prove that a system cannot simultaneously have all three.

Assume for contradiction that there does exist a system that is consistent, available, and partition tolerant. The first thing we do is partition our system. It looks like this.

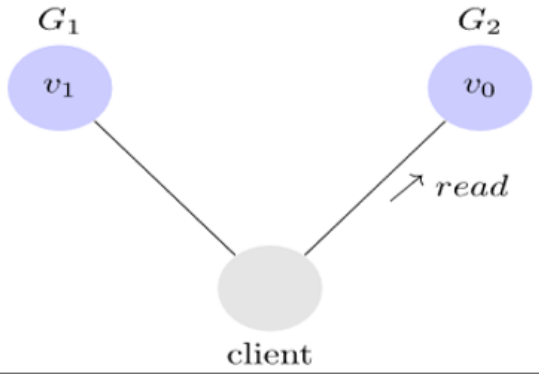


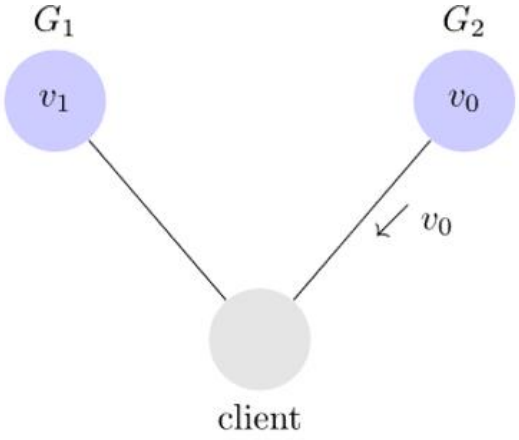
Next, we have our client request that v_1 be written to G_1. Since our system is available, G_1 must respond. Since the network is partitioned, however, G_1 cannot replicate its data to G_2. Gilbert and Lynch call this phase of execution α_1.





Next, we have our client issue a read request to G_2. Again, since our system is available, G_2 must respond. And since the network is partitioned, G_2 cannot update its value from G_1. It returns v_0. Gilbert and Lynch call this phase of execution α_2.





G_2 returns v_0 to our client after the client had already written v_1 to G_1. This is inconsistent.

We assumed a consistent, available, partition tolerant system existed, but we just showed that there exists an execution for any such system in which the system acts inconsistently. Thus, no such system exists.

Q19. Explain Basic MapReduce with example.

Answer:

1. Definition

- MapReduce = a **programming model** for large-scale distributed data processing.

2. Phases

1. **Map Phase:** Input data split into (key, value) pairs.
2. **Shuffle Phase:** Keys grouped across nodes.
3. **Reduce Phase:** Values aggregated per key.

3. Example: Word Count

- Input: “Hello world Hello”
- Map: (“Hello”,1), (“world”,1), (“Hello”,1)
- Shuffle: (“Hello”, [1,1]), (“world”, [1])
- Reduce: (“Hello”,2), (“world”,1)

Q20. Why Partition and Combiners are used? Explain with example.

Answer:

1. Partitioning

- Splits intermediate data among reducers.
- Ensures balanced workload.
- **Example:** Word count → partition by first letter: A–M to Reducer1, N–Z to Reducer2.

2. Combiners

- Mini-reducers that run on mapper outputs.
- Reduce data sent over the network.
- **Example:** If Mapper emits (“Hello”,1), (“Hello”,1), combiner aggregates locally to (“Hello”,2).

3. Benefits

- Less network traffic.
- Improved processing efficiency.

Q21. Explain the process of partitioning in MapReduce. How does it improve processing efficiency?

Answer:

1. Process

- After the **Map phase**, data is partitioned based on key.
- A partition function decides which reducer a key goes to.
- Default = hash partitioning.

2. Benefits

- **Load balancing:** Ensures no reducer is overloaded.
- **Locality:** Related keys sent to the same reducer.
- **Parallelism:** Enables all reducers to work simultaneously.

3. Example

- If keys are customer IDs, partition function ensures same customer's data goes to one reducer.

Q22. How does the combining phase work in MapReduce, and what is its purpose?

Answer:

1. Definition

- Combiner = "mini reducer" running on Mapper's output.

2. Purpose

- Reduces volume of intermediate data before sending to reducers.
- Saves network bandwidth.

3. Example

- Word count: Mapper emits ("Hello",1) 1000 times.
- Combiner aggregates locally → ("Hello",1000).
- Instead of sending 1000 pairs over the network, only 1 pair sent.

Q24. How can MapReduce calculations be composed to handle more complex queries or data processing tasks? Explain with example.

Answer:

1. Concept

- Complex queries require **multiple MapReduce jobs chained together**.
- Output of one job becomes input for the next.

2. Example: Join Operation

- Job 1: Map → emit (key=customerID, value=order). Reduce → groups all orders by customer.
- Job 2: Map → merge with customer info. Reduce → produces final joined dataset.

3. Benefits

- Enables multi-step workflows like joins, sorting, grouping, analytics.

Q25. Explain Two-Stage MapReduce with example.

Answer:

1. Definition

- A process where one MapReduce job's output feeds into another.

2. Example: Average Word Length

- **Stage 1:** Map → emit (word, length). Reduce → total length & count per word.
- **Stage 2:** Map → calculate average length. Reduce → final averages.

3. Benefits

- Handles problems that cannot be solved in one pass.

Q28. What is replication and session consistency?

Answer:

- **Session Consistency:** Guarantees that within a user's session, they always see their own writes.
  - **Read-your-writes consistency** which means that, once you've made an update, you're guaranteed to continue seeing that update.

- One way to get this in an otherwise eventually consistent system is to provide **session consistency**: Within a user's session there is read-your-writes consistency
- This does mean that the user may lose that consistency should their session end for some reason.
- There are a couple of techniques to provide session consistency. A common way, and often the easiest way, is to have a
- **Sticky session**: a session that's tied to one node
- A sticky session allows you to ensure that as long as you keep read-your-writes consistency on a node, you'll get it for sessions too.
- The downside is that sticky sessions reduce the ability of the load balancer to do its job.

- **Example:** If a user updates profile picture, they must see the new picture immediately in their session, even if other replicas lag.

Q27. Write the difference between SQL and NoSQL.

Table 4.3 SQL versus NoSQL

SQL	NoSQL
Relational database	Non-relational, distributed database
Relational model	Model-less approach
Pre-defined schema	Dynamic schema for unstructured data
Table based databases	Document-based or graph-based or wide column store or key-value pairs databases
Vertically scalable (by increasing system resources)	Horizontally scalable (by creating a cluster of commodity machines)
Uses SQL	Uses UnQL (Unstructured Query Language)
Not preferred for large datasets	Largely preferred for large datasets
Not a best fit for hierarchical data	Best fit for hierarchical storage as it follows the key-value pair of storing data similar to JSON (Java Script Object Notation)
Emphasis on ACID properties	Follows Brewer's CAP theorem
Excellent support from vendors	Relies heavily on community support
Supports complex querying and data keeping needs	Does not have good support for complex querying
Can be configured for strong consistency	Few support strong consistency (e.g., MongoDB), few others can be configured for eventual consistency (e.g., Cassandra)
Examples: Oracle, DB2, MySQL, MS SQL, PostgreSQL, etc.	MongoDB, HBase, Cassandra, Redis, Neo4j, CouchDB, Couchbase, Riak, etc.