Jahong Liu

System Design Basics 16: Replication and Sharding

Replication and Sharding

Replication and sharding are two techniques commonly used together in a distributed system to achieve high availability and throughput.

Replication

When a single database cannot handle all incoming requests, replication comes into pay. Replication involves creating a copy of the database called a replica. The replica(s) is hosted on a separate machine or server, and it is keep in sync with the original database.

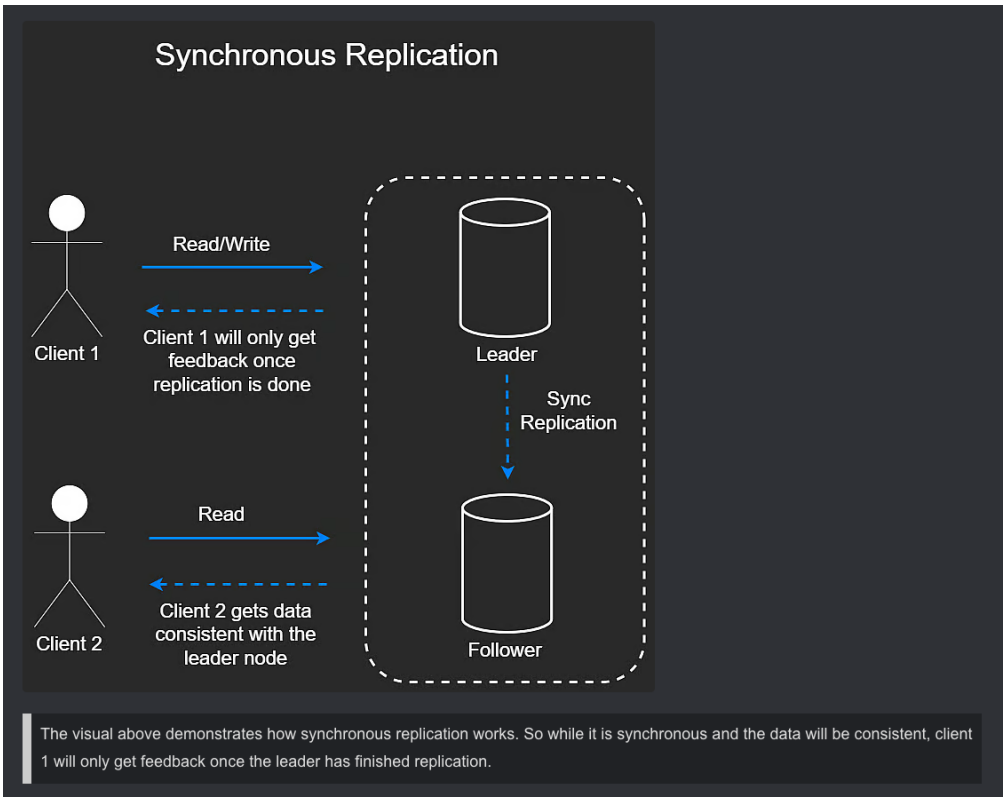
The original database is the leader or master, while the replica is the follower or slave.

In a leader/follower architecture, data replication flows from the leader to the follower, and the leader is responsible for updating the follower. If replication were attempted from the follower to the leader, the leader would not be fully updated.

There are two ways in which replication occurs: asynchronously and synchronously.

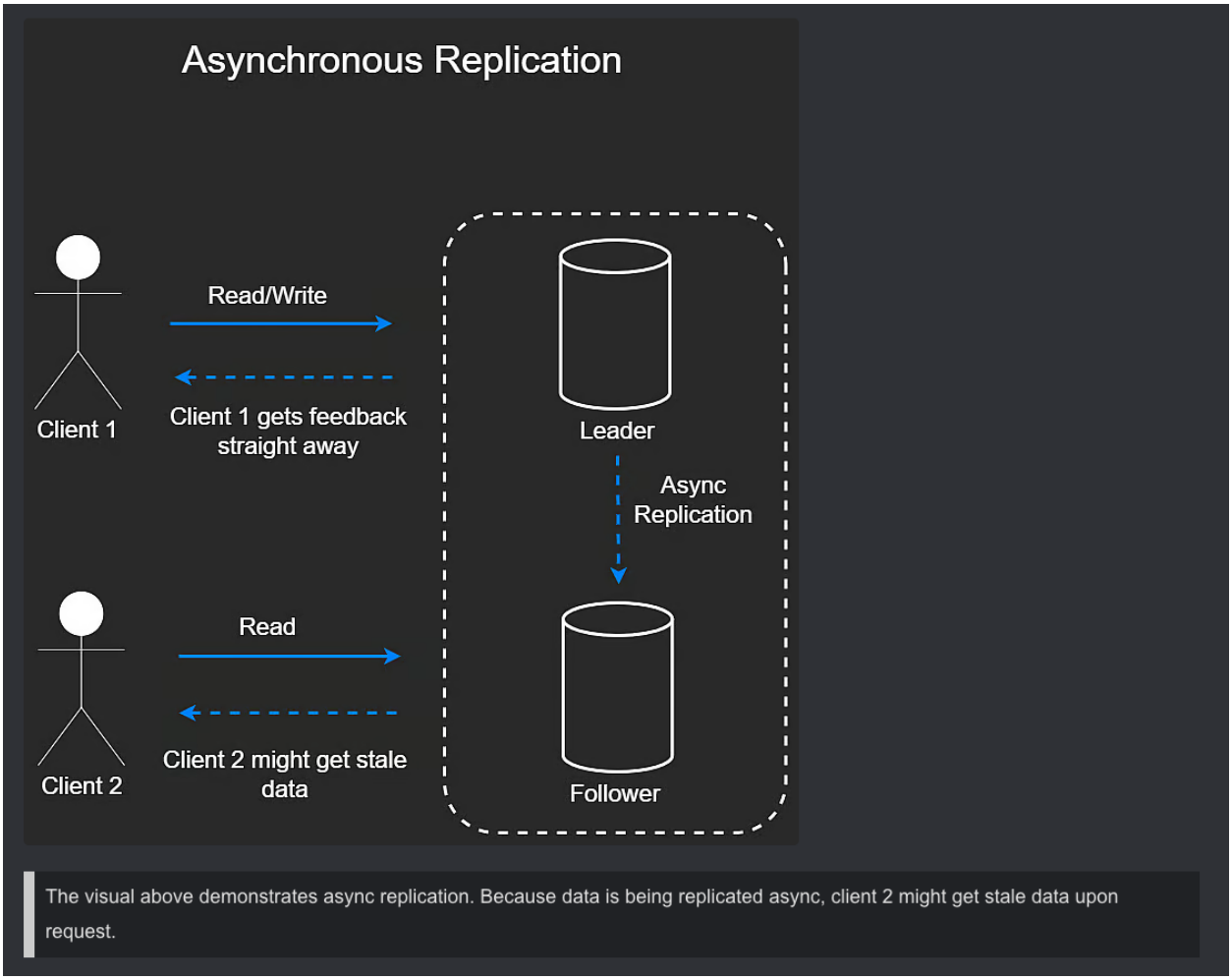
Synchronous Replication

In synchronous replication, every write transaction on the leader is immediately replicated on the follower, ensuring consistency between the two replicas. However, this approach introduces latency. The benefit is that if the leader goes down, the mostly updated follower can take its place, providing high availability.



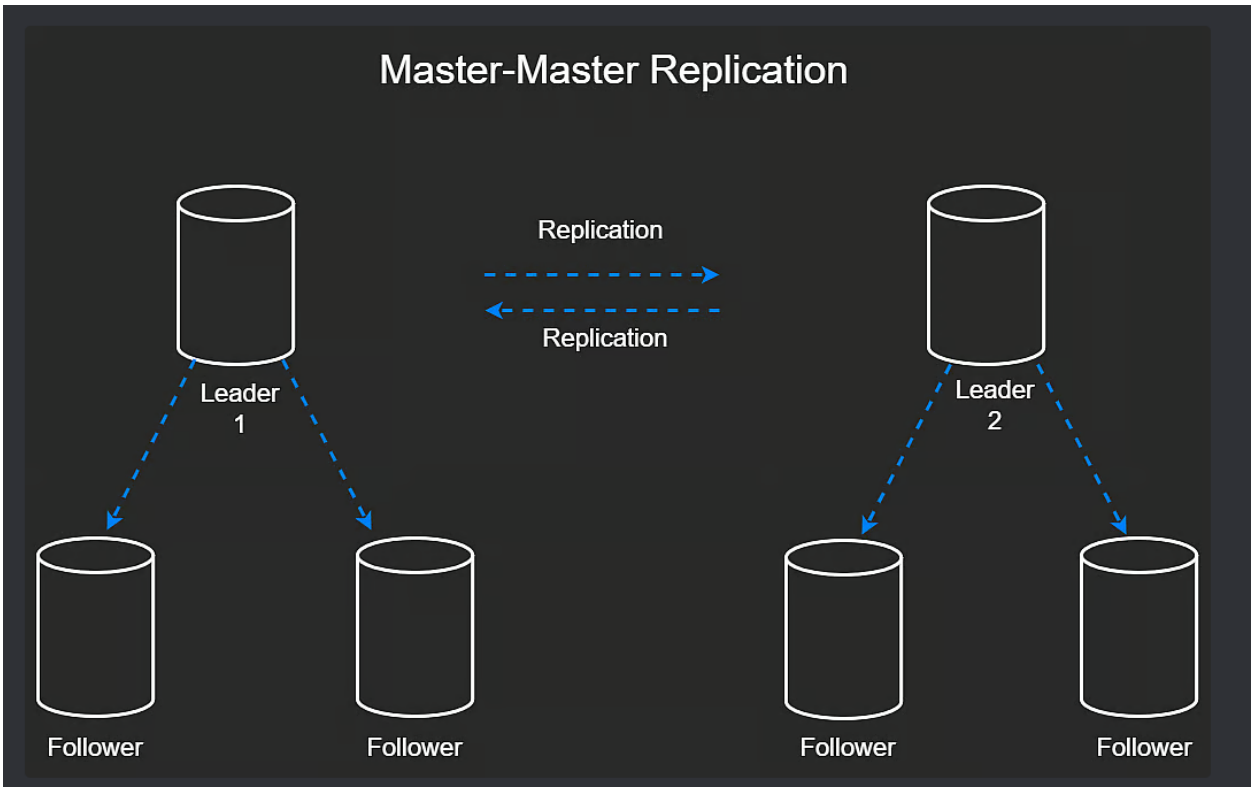
Asynchronous Replication

Asynchronous replication involves a delay in data replication. The leader database commits the transaction and sends replication data to the follower without waiting for the follower to acknowledge or apply the changes immediately. This reduces latency, but it means that if a client makes a request to the follower before the leader has updated it, the data might be stale until the leader updates it. This is the trade-ff made for increased availability.



Master-Master (Multi-Master) Replication

Master-Master replication is used when data needs to be served in different regions. For example, one leader might serve the west while another serves the east. Both leaders can be written to and read from, making it ideal for distributing data across different parts of the world. However, synchronation latency between the leaders can be a challenge, and measures like periodic updates are needed to keep them in sync.



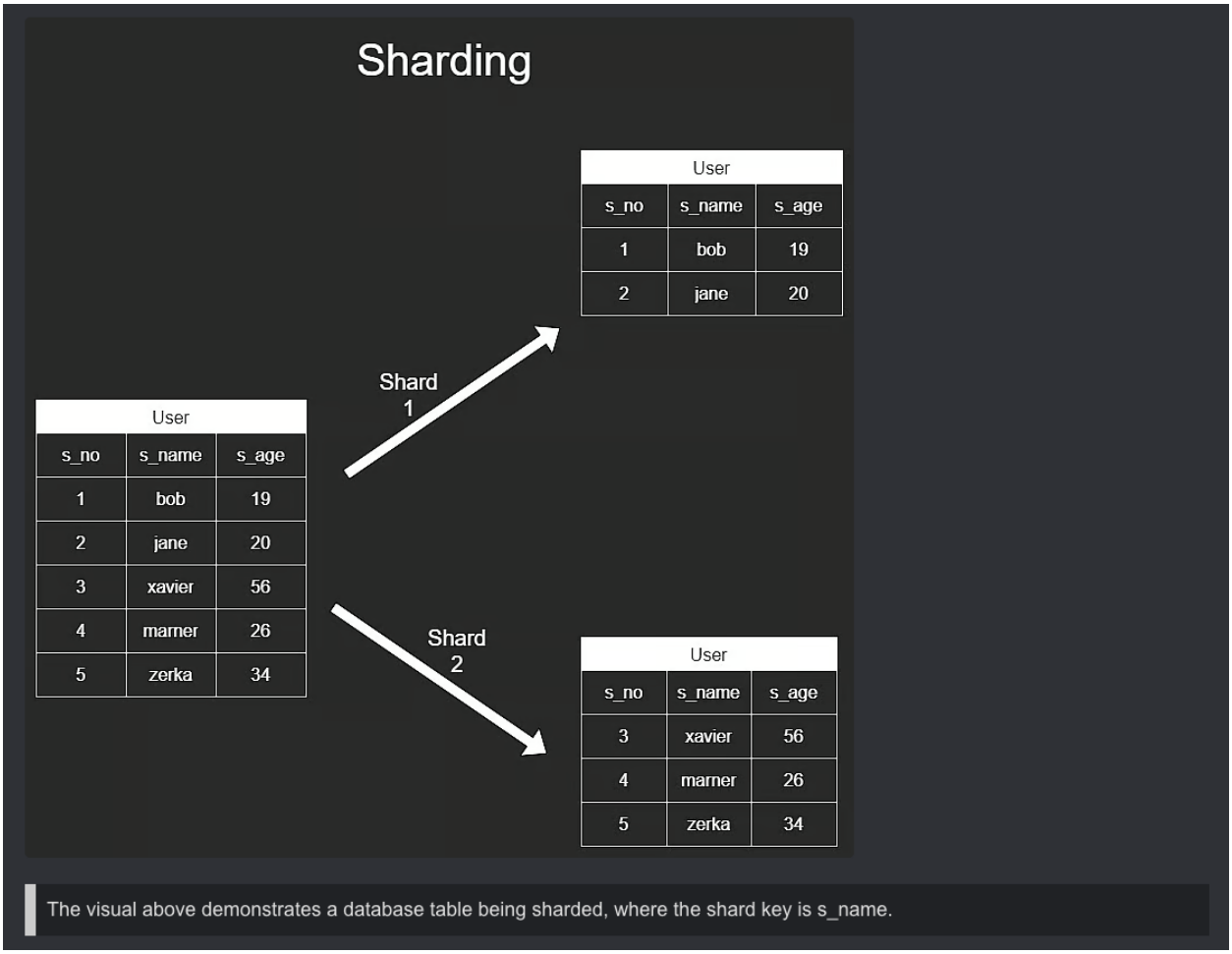
Sharding

Sharding is used when replication alone is insufficient to handle the high traffic volume on a single database. It involves dividing the database into smaller shards, each hosted on a separate machine or server. By distributing data and workload across multiple shards, the system achieves improved performance, scalability, and availability. Each shared contains only a subset of the entire dataset, and they do not have a complete copy of the original database.

Now, because we have multiple shards, what decides how data gets partitioned?

One approach is the range-based approach. Under this approach, data is split according to ranges. If we apply this to our 100 row approach with our four shards, we might split it by having 1-25, 26-50, 51-75 and 76-100, where these numbers are the IDs.

Determing how data is partitioned among shards is done using a shard key. The shard key is a chosen criterion or attribute that determines which shard each data belongs to. For example, in a relational database, the shard key might be based on sex, such as splitting data between male and female. Alternatively, it could be based on a first name/last name basis, such as dividing names from A-L and M-Z into separate shards.



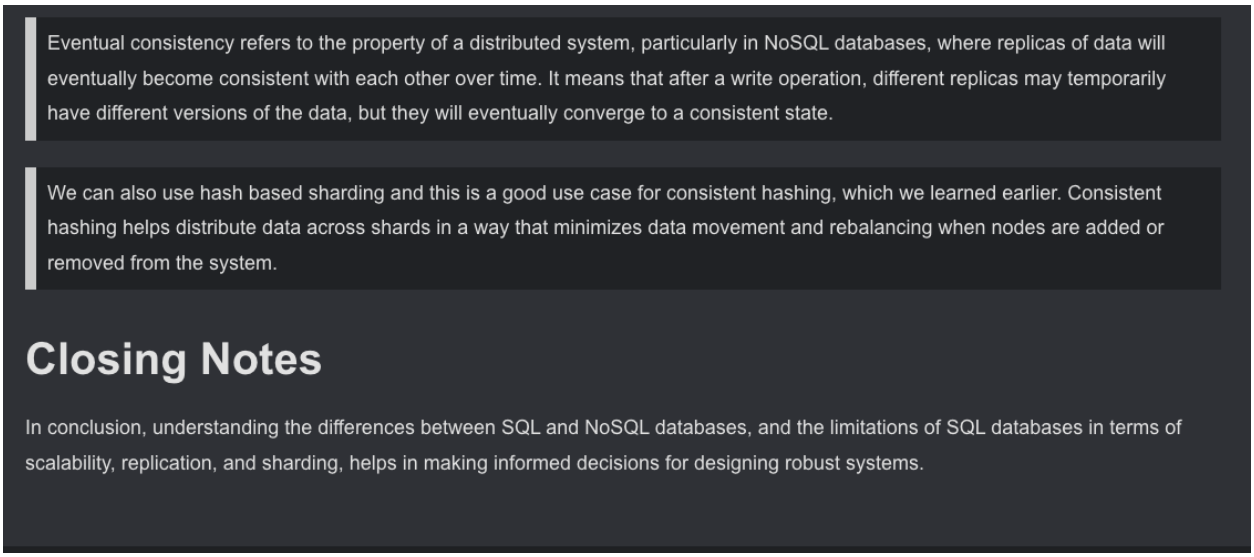
Challenges with sharding

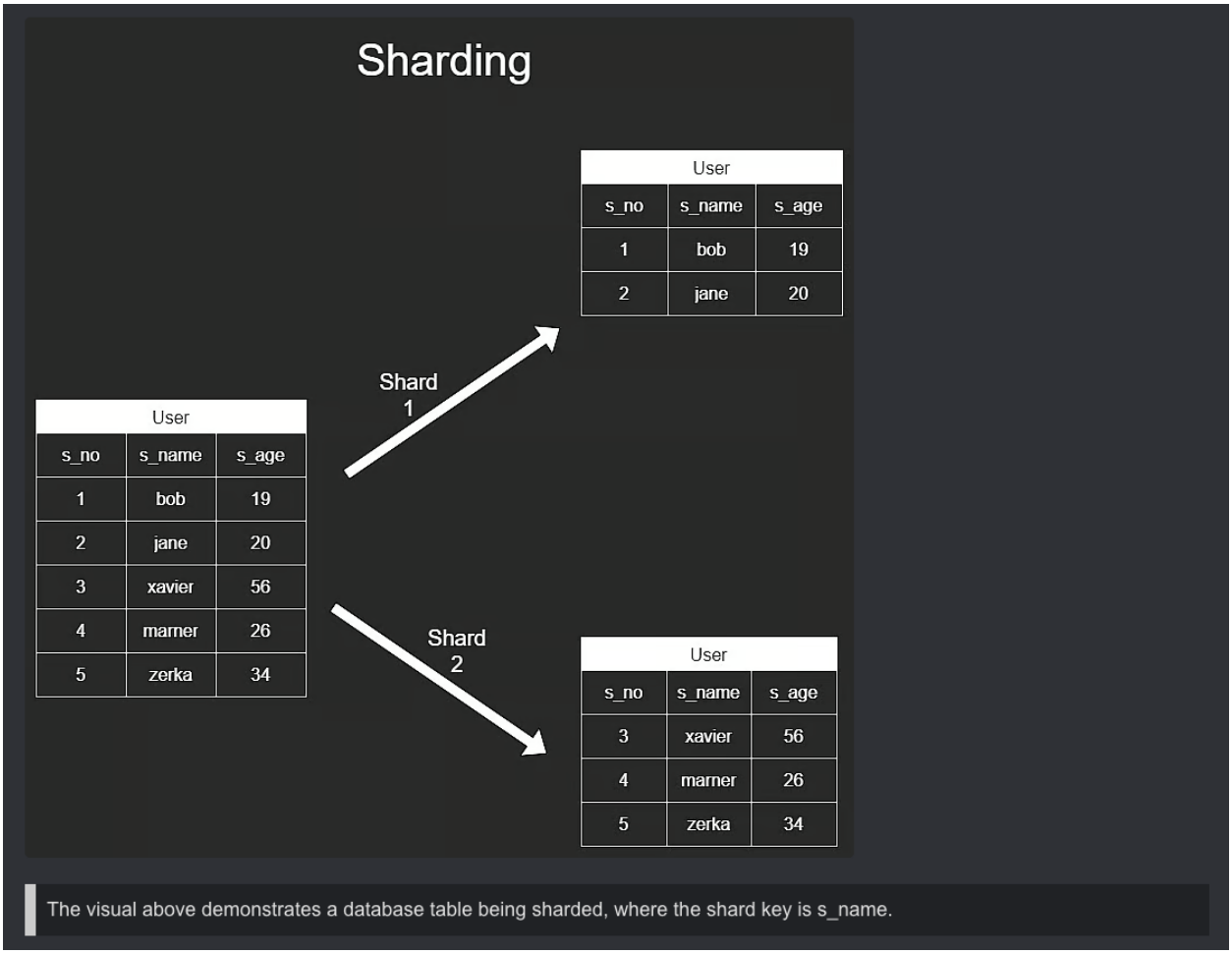
While sharding is effective for handling high traffic, it comes with its own challenges. For instance, when dealing with hundreds of tables, ensuring related tables with related data end up in the same shard can be complex. Additionally, maintaining the ACID (Atomicity, Consistency, Isolation, Durability) properties poses challenges in sharding relational databses, as they are not designed for distribution. SQL databases like MySQL and PostgreSQL do not inherently support sharding, requiring developers to implement sharding logic at the application level, which can become complicated.

On the other hand, NoSQL databases, designed with horizontal scaling in mind, are better suited for sharding, as they do not have the same constrains as relational databases. They offer eventual consistency, where data consistency across nodes is achieved over time.

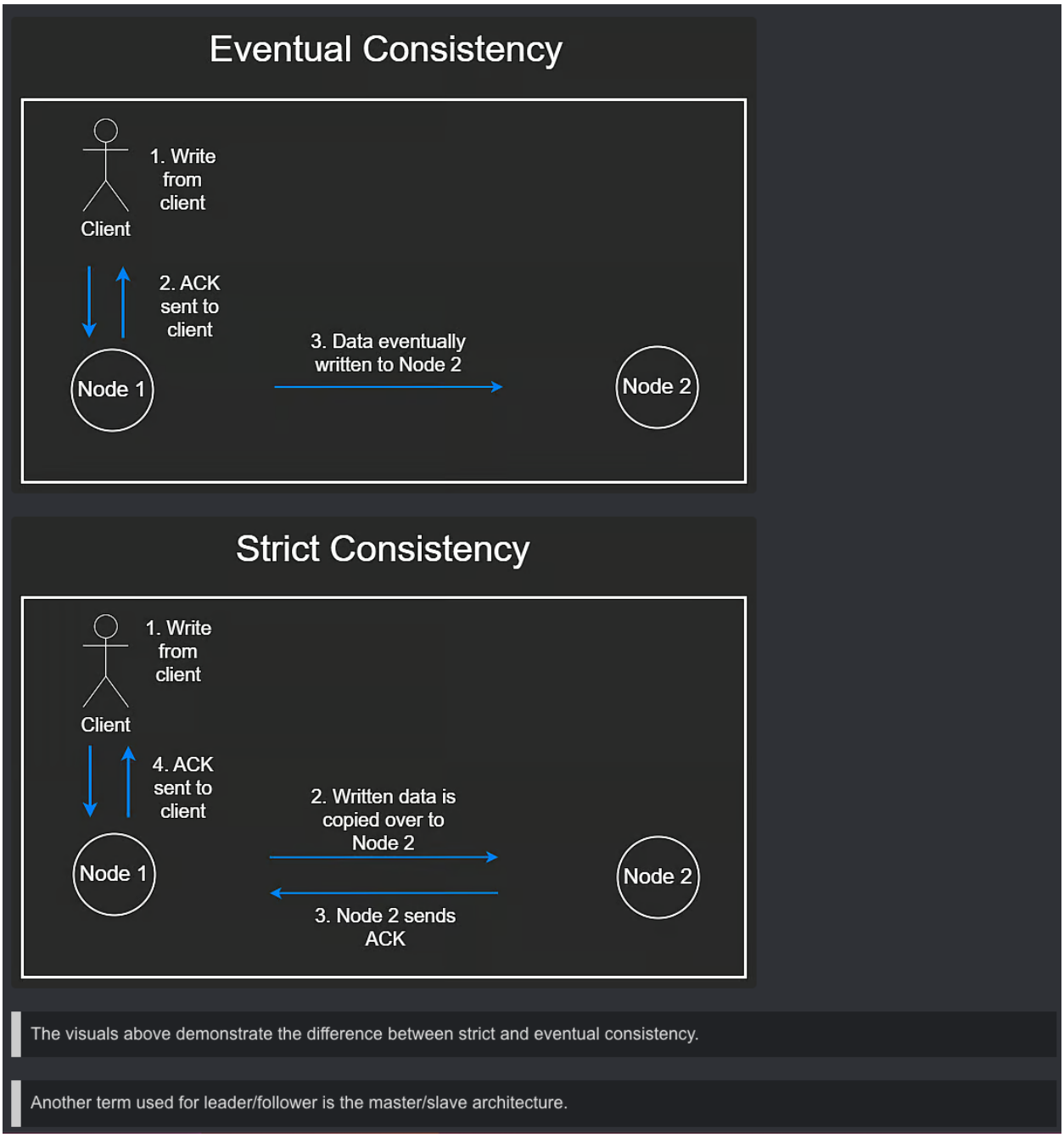
Because our data is non-relational in NoSQL databases, we don’t have the same constrains that we have when we are using relational databses. NoSQL databases were designed with horizontal scaling in mind so these limitations don’t apply as much to non-

relational databases. The data does not even need to be consistent from the nodes – it just needs to be eventually consistent.





The visual above demonstrates a database table being sharded, where the shard key is s_name.



The visuals above demonstrate the difference between strict and eventual consistency.

Another term used for leader/follower is the master/slave architecture.