Introduction to Database Replication (Types and Advantages)

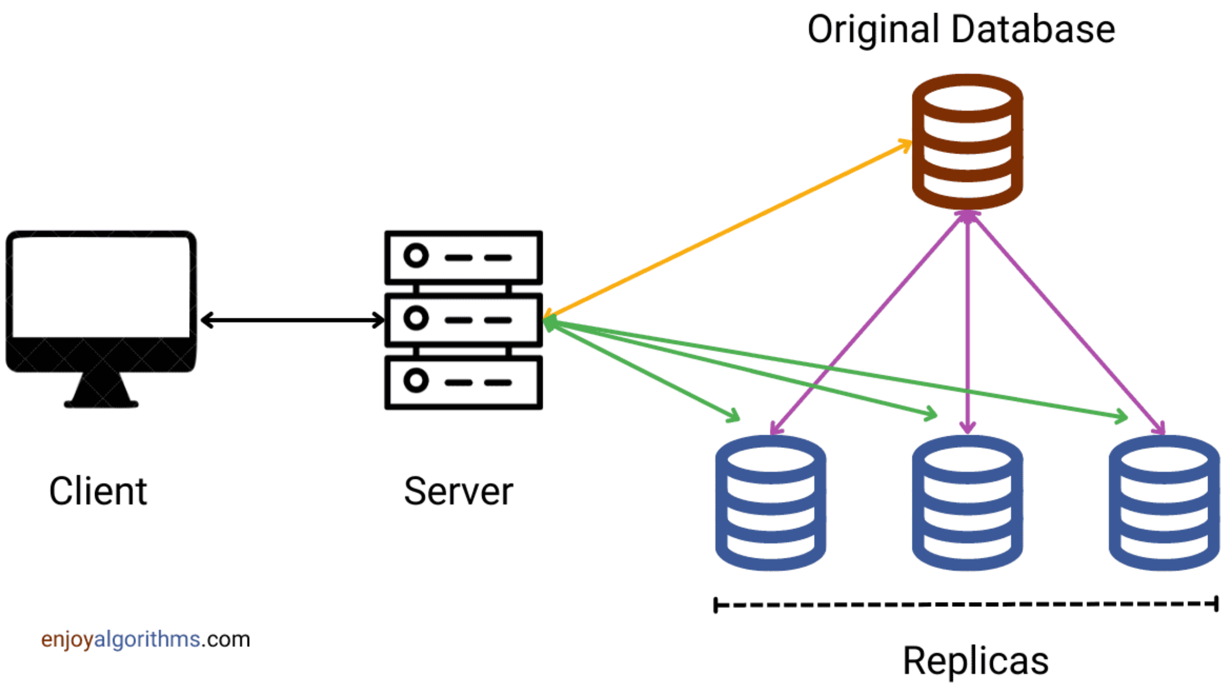
The concept of database replication has evolved in the last two decades due to the increasing use of distributed databases instead of databases with a single node. But its core principles are more and less the same over time. One of the reasons is: The fundamental constraints of distributed systems have not changed significantly.

In this blog, we will discuss the definition, types, advantages and disadvantages of database replication.

What is Database Replication?

Database replication is the process of keeping multiple copies of the same data in different servers (replicas) so that if one server goes down, other servers can continue to serve data without any interruption or downtime.

- Database replication provides several advantages like high performance, availability, and reliability.
- If the data that we are replicating does not change over time, then replication is easy: we just need to copy the data to every node once, and we are done. All the difficulty in replication lies in handling changes to replicated data.
- We can create a replica of an entire database or a subset of the database.



Understanding database replication with an example

Suppose there is an online store that has a single database server that stores all the data related to the products, orders, and customers. If this server fails due to some reason (hardware issues, software issues, etc.) the whole website will be unavailable.

To solve this problem, we can use database replication. The website owner can set up multiple database servers that replicate data from the primary database server. If the primary server goes down, other servers can take over and continue to serve requests.

Database replication techniques based on architecture

It's important to ensure that any updates made to the data on one node are reflected on all other nodes. So, we need efficient techniques that can ensure that all nodes have the most up-to-date version of the data.

There are three types of database replication techniques based on architecture: Single-leader replication, multi-leader replication, and no-leader replication. Each strategy has its advantages and disadvantages.

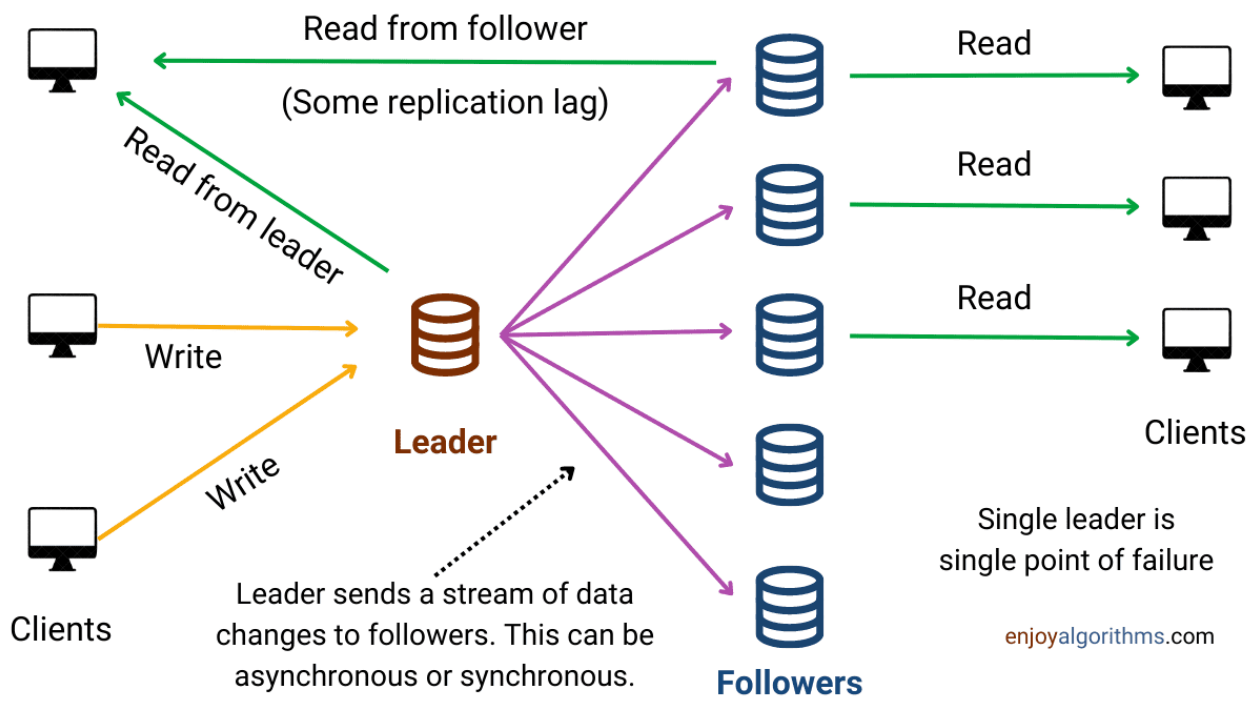
Single-leader Architecture

This is also known as **Master-Slave** or **Active-Passive replication**.

In single leader replication, there is a single leader (master) and several follower replicas (slaves). All write requests are served by the leader node and all read requests are served by the leader or any of the follower nodes. The best idea would be to use the leader only for write requests and distribute read requests across followers. This will remove some load from the leader.

- After serving the write request, the leader node sends a stream of data changes to the follower nodes to update their current state of data.
- As write operations are concentrated on a single node, if the leader node fails, the system may become unavailable until a new leader is elected.
- During the replication process, we need to handle one of the critical problems: reads from follower nodes may not reflect the latest changes as there can be some delay in the replication process (replication lag).
- This architecture is a good option when read-write ratio is very high.

We have covered this idea of replication in a separate blog: Complete overview of Master-Slave Replication.

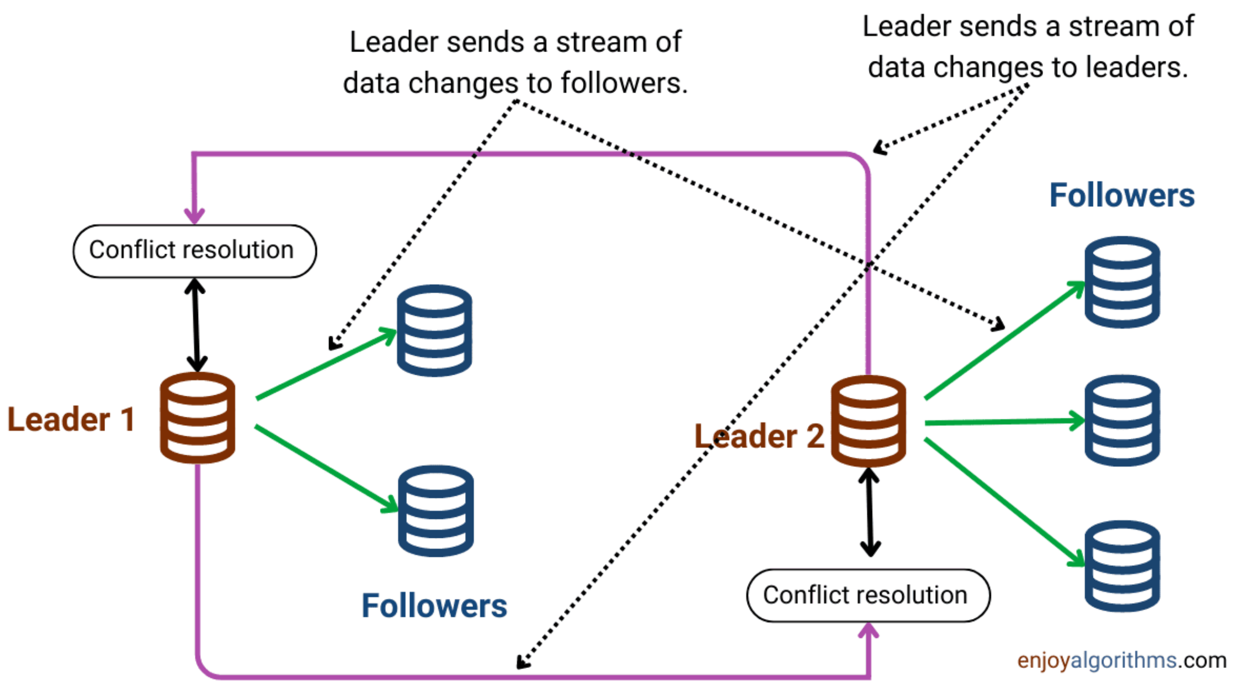

Read from follower
(Some replication lag)
Read
Read
Read
Read from leader
Write
Write
Leader
Clients
Clients
Leader sends a stream of data changes to followers. This can be asynchronous or synchronous.
Followers
Single leader is single point of failure
enjoyalgorithms.com

Multi-leader Architecture

There is a big problem with leader-based replication: If the leader is unavailable due to some reason, we cannot perform write operations to the leader until we upgrade another follower replica as a leader. So a common solution to such a problem is to use multi-leader replication. This is also called **master-master or active-active replication.**

Here, clients can send write requests to one of the leader nodes, and each leader works as a follower for the other leader. So whenever a leader node performs the write operation, it will forward streams of data change to all other leader nodes.

There are some complexities in multi-leader architecture. For example, conflicts can arise when two or more leader nodes receive conflicting write requests simultaneously. So it's important to have conflict resolution mechanisms in place to ensure data consistency. **Note:** In the near future, we will discuss multi-leader in detail in a separate blog.


Leader sends a stream of data changes to followers.
Leader sends a stream of data changes to leaders.
Conflict resolution
Leader 1
Followers
Leader 2
Followers
Conflict resolution
enjoyalgorithms.com

- When data needs to be available in multiple regions, a multi-leader architecture can help reduce latency and improve performance. The idea is simple: Each region can designate its own leader node.
- By having multiple leader nodes, a multi-leader architecture can provide redundancy and ensure that the system remains available even if one or more nodes fail. It can also enable write scaling because write requests can be distributed across multiple nodes.

No-leader Architecture

This is also known as **leader-less replication**. In this architecture, clients send write requests to several nodes and read from several nodes in parallel. There is no concept of a leader in this approach, which allows any replica to directly accept writes from clients.

Leaderless replication can provide high availability and fault tolerance. Because there is no single point of failure, the system can continue to function even if some nodes fail. It can also provide high read and write throughput because requests can be distributed across multiple nodes.

This method also poses challenges for synchronization, as it can be difficult to ensure that all nodes have the same view of the data at all times. In addition, handling conflicts that may arise from concurrent writes can be complex, and careful design is necessary to ensure data consistency.

Note: In the near future, we will discuss multi-leader in detail in a separate blog.

Asynchronous replication

In this strategy, the leader node responds immediately to the client after updating its own copy of the data, without waiting for the changes to be propagated to the followers. But there is a risk of data loss without the client's knowledge because the confirmation comes before the main replication process. The idea is: replication happens in the background and the leader asynchronously propagates the changes to the followers. If the leader node crashes, changes in data that are not propagated are lost permanently.

Despite this disadvantage, asynchronous replication is the default strategy for most data stores because it offers flexibility. The idea is: Client is blocked only for the duration that the write happens on the leader!

Synchronous replication

In synchronous replication, once the leader node updates its own copy of the data, it initiates the write operation on its followers: Followers receive the update, apply changes to their copy of data, and then send confirmation to the leader. Once the leader receives confirmation from all followers, it responds to the client and completes the operation.

Synchronous replication ensures that followers are always in sync and consistent with the leader, making this setup fault-tolerant by default. Even if the leader crashes, the entire data is still available to the followers. So the system can easily promote any one of the followers as the new leader and continue to function as usual.

One major disadvantage of synchronous replication is that the client and the leader can remain blocked if a follower becomes non-responsive due to a crash or network partition. In other words, the leader will continue to block all writes until the affected followers become available again. So having a large number of followers in this setup can result in longer block times for the client.

There are many trade-offs to consider with replication. For example, when to use synchronous or asynchronous replication and how to handle failed followers. Explore the master-slave replication blog.

Full replication

In this method, we copy the entire original database at every replica. This makes data highly available and decreases query execution time (data can be fetched from any closest replica). But it can be slow to update the replicas because the entire database needs to be copied at every replica's location.

Full replication is useful when users at different locations need to see the same view of the data. For example, users looking for match scores need to see the same details about the match, regardless of their location.

Partial replication

In partial replication, we store a copy of only a selected part of data from the original database at each replica. So the type and importance of the data determine the number of replicas required. Here update process of each replica is fast because each replica only receives a portion of the entire database.

But the problem is: If the local replica does not contain some required data, it needs to be fetched from the original database. This can increase the query execution time. So, partial replication is useful when one wants to provide an isolated view of data based on their location.

For example, suppose an online fashion retailer sells clothing items in different regions of the world. The retailer may have different inventories in different regions depending on local demand. By replicating only the relevant data to each location, the retailer can ensure that customers in each region see only the relevant products that are available.

Advantages of database replication

- Replication helps us scale out the number of machines that can serve read and write queries, which increases throughput and allows more queries to be processed in parallel.
- It helps us keep data geographically close to users, which reduces latency. The best example is CDN.
- If one of the database servers is destroyed by a natural disaster, data is still preserved. We do not need to worry about data loss because data is replicated across multiple locations.
- By replicating data across different database servers, website remains in operation even if a database server is down due to maintenance or some other reasons.

Disadvantages of database replication

- Replicating data across multiple servers can increase the complexity of the database system.
- Implementing and maintaining a replication system can be expensive, as it requires additional hardware, software, and IT resources.
- Replication lag can occur when changes made to the master database are not immediately propagated to the replicated databases. This can result in inconsistent data across different instances of the database.
- Data conflicts can occur when changes are made to the same data in different instances of the database. This can result in data inconsistencies or even data loss if conflicts are not properly resolved.
- Replication systems require maintenance, monitoring,

troubleshooting, and periodic updates. This can add to the overall maintenance overhead of the database system.

References

- Designing data-intensive applications by Martin Kleppmann
- Web Scalability for Startup Engineers by McGraw-Hill
- Understanding distributed systems by Roberto Vitillo

We will keep updating this blog with more insights on database replication. If you have any queries or feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy system design!