Volume 9 Issue 6 @ November - December 2021 IJIRMPS | ISSN: 2349-7300

Replication and Partition Strategies in Distributed Systems and Usecases

Arjun Reddy Lingala

arjunreddy.lingala@gmail.com

Abstract

Modern computing heavily relies on distributed systems to ensure scalability, fault tolerance, and high availability. Two fundamental techniques that enhance distributed architectures are replication and partitioning. Replication involves maintaining multiple copies of data to improve fault tolerance and accessibility, whereas partitioning divides datasets to enhance scalability and load distribution. This paper provides an in-depth examination of these strategies, highlighting their benefits, trade- offs, and real-world applications. Various replication models such as synchronous and asynchronous replication, multi-leader replication, and leader-less replication are explored. Similarly, partitioning techniques including horizontal and vertical partitioning, range-based, hash-based, partitioning using secondary indexes, rebalancing partitions and its challenges, and impact on system performance. Additionally this study discusses the implications of replication and partitioning as two separate aspects which deep dives into replication considering that the data is fit into a single server, and partitioning considering that the replications is handled well. This study highlights the importance of replication and partitioning without considering the impact on one another and later user can take appropriate way based on the system guarantees that they want for their system optimal performance. This paper serves as a comprehensive resource for system architects, engineers, and researchers striving to optimize distributed systems in an era of growing data complexity.

Keywords: Scaling, Replication, Partitioning, Distributed systems, Consistent hashing, Consensus, Coordination, Leader- follower, Replication lag, Latency, Throughput

I. INTRODUCTION

The shift from monolithic to distributed systems has been driven by the need to handle exponential data growth where IoT devices alone generate 40 zetabytes of data annually, handle global user bases where companies like Facebook serve over 2 billion users globally across 200+ countries, and handle fault tolerance where downtimes are very costly for organizations requiring redundancy of data. The need for distributed systems arises from the limitations of single-node architectures in handling high-throughput workloads, fault tolerance, and geographical distribution. Modern applications, such as social media platforms, online transaction processing (OLTP) systems, and real-time analytics frameworks, demand both rapid data access and resilience to failures. Effective replication and partitioning strategies enable organizations to address these challenges by distributing workloads efficiently and ensuring data availability in the face of failures. Replication strategies include synchronous replication, asynchronous replication, multi-leader replication, and leader-less replication which have various trade-offs in terms of consistency, avail- ability and network overhead. Similarly partition strategies include range based partitioning, hash based partitioning, and directory based partitioning which impact system performance based on data distribution and query efficiency.

This paper aims to provide a comprehensive analysis of replication and partitioning strategies in distributed systems. We discuss replication and partitioning as approaches independent of one another in terms of their advantages, challenges, and real-world applications. Through this study, we discuss insights into replication and partitioning strategies and how organizations can prioritize various types of strategies based on their needs.

II. Replication

Replication is keeping a copy of the same data on multiple machines that are connected via a network to keep data geographically close to your users reducing latency, to allow the system to continue working even if some of its parts have failed increasing availability, to scale out the number of machines that can serve read queries increasing read throughput. While discussing replication, let us assume that the dataset is so small that each machine can hold a copy of the entire dataset relaxing the assumption of partitioning of dataset that is too big for a single machine. 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 done. All of the difficulty in replication lies in handling changes to replicated data. We will discuss popular algorithms for replicating changes between multiple nodes of distributed cluster: single leader – synchronous and asynchronous, multileader and leaderless replication and pros and cons of these approaches.

A. Single-Leader Replication

In single leader replication, one of the replicas is designated the leader and when clients want to write to the database, they must send their requests to the leader which first writes the data to its local storage. The other replicas know as followers read data sent by leader as part of a replication log. Each follower replica takes the log from the leader and updates its local copy of the data by applying writes in the same order as they were processed on the leader replica. When a client wants to read from the database, it can query either the leader or any of the follower replicas, but the writes are only accepted on leader replica. Another important information of replicated system is whether the replication is synchronous or asynchronous. In case of synchronous replication, the leader waits until follower replica has confirmed that it received the write before reporting success to the client where as in case of asynchronous replication, he leader sends the message, but doesn't wait for a response from the follower.

B. Multi-Leader Replication

Leader-based replication has one major downside: there is only one leader, and all writes must go through it.iv If you can't connect to the leader for any reason, for example due to a network interruption between you and the leader, you can't write to the database. A natural extension of the leader- based replication model is to allow more than one node to accept writes. Replication still happens in the same way each node that processes a write must forward that data change to all the other nodes. Each leader simultaneously acts as a follower to the other leaders in multi-leader replication. A replication topology describes the communication paths along which writes are propagated from one node to another. In case of multi-leader replication multiple replication topologies are available with most common being all-to-all topology in which every leader sends its writes to every other leader. Few other topologies include circular topology where each node receives writes from one node and forwards those writes to one other node and star topology where one designated root node forwards writes to all of the other nodes

C. Leader-Less Replication

In case of leader less replication clients send each write to several nodes, and read from several nodes in parallel in order to detect and correct nodes with stale data. In this case, client sends a write request to one node (the leader), and the database system takes care of copying that write to the other replicas. A leader determines the order in which writes should be processed, and followers apply the leader's writes in the same order. In leader less replication strategy quorum based reading and writing is used to guarantee the data. If there are n replicas, every write must be confirmed by w nodes to be considered successful, and we must query at least r nodes for each read. As long as w + r ¿ n, we expect to get an updated value when reading, because at least one of the r nodes we're reading from must be up to date. Reads and writes that obey these r and w values are called quorum reads and writes. If we know that every successful write is guaranteed to be present on at least two out of three replicas, that means at most one replica can be stale. Thus, if we read from at least two replicas, we can be sure that at least one of the two is up to date. If the third replica is down or slow to respond, reads can nevertheless continue returning an updated value.

III. PARTITIONING

Partitioning is usually combined with replication so that copies of each partition are stored on multiple nodes. This means that, even though each record belongs to exactly one partition, it may still be stored on several different nodes for fault tolerance. The main reason for wanting to partition data is scalability. Different partitions can be placed on different nodes in a shared-nothing cluster. Thus, a large dataset can be distributed across many disks, and the query load can be distributed across many processors. For queries that operate on a single partition, each node can independently execute the queries for its own partition, so query throughput can be scaled by adding more nodes. Large, complex queries can potentially be parallelized across many nodes, although this gets significantly harder. As discussed replication, we discuss partition independent of replication and considering that the nodes are available.

A. Range Partitioning

Goal with partitioning is to spread the data and the query load evenly across nodes. If the partitioning is unfair, so that some partitions have more data or queries than others, we call it skewed. The presence of skew makes partitioning much less effective. A partition with disproportionately high load is called a hot spot. The simplest approach for avoiding hot spots would be to assign records to nodes randomly. Range partitioning assigns range of keys to each partition. If we also which partition is assigned to which node, then we can make your request directly to the appropriate node. The ranges of keys are not necessarily evenly spaced, because the data may not be evenly distributed. The downside of key range partitioning is that certain access patterns can lead to hot spots.

B. Hash Partitioning

Due to risk of skew and hot spots, many distributed data- stores use a hash function to determine the partition for a given key. A good hash function takes skewed data and makes it uniformly distributed. Given a new string, hash function returns a seemingly random number between 0 and 2^{32}. Even if the input strings are very similar, their hashes are evenly distributed across that range of numbers. For partitioning pur- poses, the hash function need not be cryptographically strong. Many programming languages have simple hash functions built in like Java's Object.hashCode() but they may not be suitable for partitioning. Once we have a suitable hash function for keys, we can assign each partition a range of hashes, and every key whose hash falls within a partition's range will be stored in that partition. This technique is good at distributing keys fairly among the partitions. The partition boundaries can be evenly spaced. Consistent Hashing [1] is a way of evenly distributing load across an internet-wide system of caches. It uses randomly chosen partition boundaries to avoid the need for central control or distributed consensus.

C. Partitioning and Secondary Indexes

Secondary Indexes are mainly used in relational databases. In this indexing approach, each partition is completely separate: each partition maintains its own secondary indexes, covering only the documents in that partition. Whenever we need to write to the database to add, remove, or update a document, we only need to deal with the partition that contains the document ID that you are writing. Querying in secondary index partitioned database is sometimes known as scatter and gather, and it can make read queries on secondary indexes quite expensive. Even if you query the partitions in parallel, scatter gather is prone to increase query latency.

IV. Challenges

A. Replication

Any node in the system can go down, perhaps unexpectedly due to a fault, but just as likely due to planned maintenance. Being able to reboot individual nodes without downtime is a big advantage for operations and maintenance. Thus, our goal is to keep the system as a whole running despite individual node failures, and to keep the impact of a node outage as small as possible. Node outages can be divided into leader outage and follower outage. On its local disk, each follower keeps a log of the data changes it has received from the leader. If a follower crashes and is restarted, or if the network between the leader and the follower is temporarily interrupted, the follower can recover quite easily from its log which is called as last transaction that was processed before the fault occurred. Thus, the follower can connect to the leader and request all the data changes that occurred during the time when the follower was disconnected. When it has applied these changes, it has caught up to the leader and can continue receiving a stream of data changes as before. Handling a failure of the leader is trickier as one of the followers needs to be promoted to be the new leader, clients need to be reconfigured to send their writes to the new leader, and the other followers need to start consuming data changes from the new leader. This process is called as failover which consists of three major steps: *1) Determine the leader failed* – There is no foolproof way of detecting what has gone wrong, so most systems simply use a timeout. Nodes frequently bounce messages back and forth between each other, and if a node doesn’t respond for some period of time, it is assumed to be dead. *2) Choosing a new leader* – A new leader could be appointed by a previously elected controller node. The best node for leadership is usually the replica with the most updated data changes from the old leader minimizing data loss. Getting all the nodes to agree on a new leader is usually done using consensus algorithm like Zookeeper [2]. *3) Reconfigure the system to use new leader* – If the old leader comes back, it might still believe that it is the leader, not realizing that the other replicas have forced it to step down. The system needs to ensure that the old leader becomes a follower and recognizes the new leader.

B. Partitioning

Over time, things change in a database where the query throughput increases, so you want to add more CPUs to handle the load, the dataset size increases, so you want to add more disks and RAM to store it, a machine fails and other machines need to take over the failed machine’s responsibilities. All of these changes call for data and requests to be moved from one node to another. The process of moving load from one node in the cluster to another is called rebalancing. After rebalancing it is expected the load should be shared fairly between the nodes in the cluster, the database should continue accepting reads and writes while rebalancing is happening, and only minimum amount of data to make rebalancing fast with optimizing data distribution.

1) Fixed Number of partitions: Create many more partitions than there are nodes, and assign several partitions to each node. If a node is added to the cluster, the new node can steal a few partitions from every existing node until partitions are fairly distributed once again. If a node is removed from the cluster, the same happens in reverse. Only entire partitions are moved between nodes. The number of partitions does not change, nor does the assignment of keys to partitions. The only thing that changes is the assignment of partitions to nodes. This change of assignment is not immediate— it takes some time to transfer a large amount of data over the network, so the old assignment of partitions is used for any reads and writes that happen while the transfer is in progress.

2) *Dynamic partitioning:* When a partition grows to exceed a configured size, it is split into two partitions so that approximately half of the data ends up on each side of the split. If lots of data is deleted and a partition shrinks below some threshold, it can be merged with an adjacent partition. An advantage of dynamic partitioning is that the number of partitions adapts to the total data volume. If there is only a small amount of data, a small number of partitions is sufficient, so overheads are small; if there is a huge amount of data, the size of each individual partition is limited to a configurable maximum.

3) *Consistent Hashing:* Consistent hashing [1] is a technique used in distributed systems [10] to distribute data across multiple nodes in a way that minimizes disruption when nodes are added or removed. It is particularly useful in partitioning strategies for distributed databases, caching systems, and load balancing. Unlike traditional hashing, adding or removing nodes does not require remapping all keys—only a small fraction of data is affected. It allows dynamic scaling of distributed systems [10] with minimal rebalancing. Virtual nodes (multiple positions per node on the ring) can be used to evenly distribute the load

V. CONCLUSION

This paper examined the fundamental role of replication and partitioning strategies in distributed systems [10], highlighting their importance in achieving high availability, fault tolerance, scalability, and performance optimization. We analyzed different replication methods, such as primary-backup, multi-primary, and quorum-based techniques, discussing their trade- offs concerning consistency, availability, and latency. Additionally, we explored partitioning strategies, including range- based, hash-based, and dynamic partitioning, emphasizing their impact on data distribution, load balancing, and system resilience. While replication enhances reliability and read performance, it introduces challenges such as data synchronization, increased storage overhead, and potential conflicts in multi-primary setups. Partitioning improves scalability but requires efficient data routing and can lead to imbalanced load distribution if not carefully managed. The interplay between these strategies necessitates a thoughtful design approach to meet the demands of modern distributed applications that require low-latency access and high throughput. Future re- search directions could explore AI-driven predictive models for data placement and advancements in consensus algorithms and distributed transaction processing can further enhance the efficiency of distributed systems [10]

REFERENCES

1. G. Swart, ”Spreading the load using consistent hashing: a preliminary report,” Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, Cork, Ireland, 2004, pp. 169-176, doi: 10.1109/ISPDC.2004.47.
2. L. B. Goel and R. Majumdar, ”Handling mutual exclusion in a distributed application through Zookeeper,” 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad, India, 2015, pp. 457-460, doi: 10.1109/ICACEA.2015.7164748.
3. "MongoDB Documentation," MongoDB, Available: https://www.mongodb.com/docs/. [October 2021].
4. Amazon Web Services, ”Amazon Dynamo DB”, [Online]. Available: https://www.elastic.co/guide/index.html.
5. P. A. Bernstein and N. Goodman, “Concurrency control in distributed database systems,” ACM Comput. Surv., vol. 13, no. 2, pp. 185–221, Jun. 1981, doi: 10.1145/356842.356846.
6. D. Abadi, “Consistency tradeoffs in modern distributed database system design,” IEEE Comput., vol.

45, no. 2, pp. 37–42, Feb. 2012, doi: 10.1109/MC.2012.33.

7. E. Brewer, “CAP twelve years later: How the ‘rules’ have changed,” IEEE Comput., vol. 45, no. 2, pp. 23–29, Feb. 2012, doi: 10.1109/MC.2012.37.
8. L. Lamport, “Paxos made simple,” ACM SIGACT News, vol. 32, no. 4, pp. 18–25, Dec. 2001, doi: 10.1145/568425.568433.
9. A. Elmore et al., “A survey of adaptive database replication,” IEEE Trans. Knowl. Data Eng., vol. 27, no. 10, pp. 2701–2720, Oct. 2015, doi: 10.1109/TKDE.2015.2430320.
10. S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” IEEE Trans. Comput., vol. 52, no. 10, pp. 1357–1374, Oct. 2003, doi: 10.1109/TC.2003.1234554.