A Crash Course on Scaling the Data Layer

BYTEBYTEGO
SEP 05, 2024 · PAID

The scalability of a system is heavily dependent on the data layer.

No matter how much effort is made to scale the API or the application layer, it is limited by the scalability of the data layer. Also, scaling the data layer is often the most difficult task during application design.

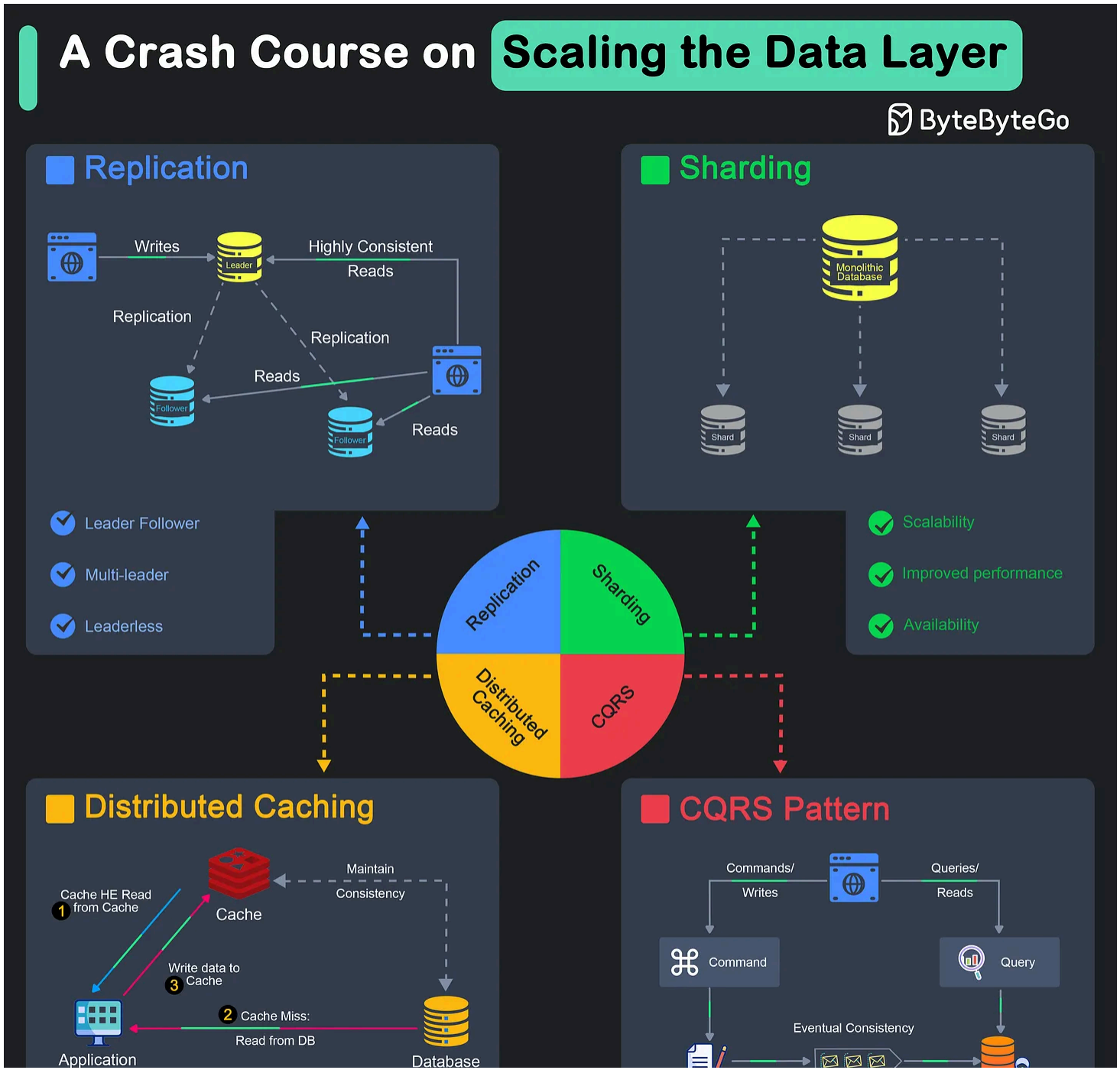
Horizontally scaling the data layer of an application, also known as “scaling out”, involves distributing the data and load across multiple servers or nodes.

This approach is particularly effective for handling large volumes of data and high traffic loads, but it also adds multiple orders of complexity. Since the data is distributed, many issues regarding transactions and consistency that don’t appear in monolithic databases become quite common.

Several techniques are available for horizontally scaling the data layer, each having pros and cons with specific nuances worth considering.

In this post, we’ll explore the major techniques for scaling the data layer horizontally along with examples. Also, we will understand the advantages and disadvantages of each technique to get a

better idea of when to use a particular approach over another choice.


A Crash Course on Scaling the Data Layer
ByteByteGo
Replication
Writes
Leader
Highly Consistent
Reads
Replication
Replication
Reads
Follower
Follower
Reads
Leader Follower
Multi-leader
Leaderless
Sharding
Monolithic Database
Shard
Shard
Shard
Scalability
Improved performance
Availability
Replication
Sharding
Distributed Caching
CQRS
Distributed Caching
Maintain
Consistency
Cache HE Read
1 from Cache
Cache
Write data to
3 Cache
2 Cache Miss:
Read from DB
Application
Database
CQRS Pattern
Commands/
Writes
Queries/
Reads
Command
Query
Eventual Consistency

Replication

Replication is a method of copying data from one location to another, ensuring that the data is available when and where it is needed.

This is one of the most important techniques to horizontally scale the database or the caching layer. It helps achieve several key objectives such as:

- **Improved Durability:** Replication ensures that data remains intact even in the event of hardware failures or other disasters. If one copy of the data becomes unavailable, the other copies can still be accessed, ensuring data durability.
- **Enhanced Availability:** Replication increases the availability of data by providing multiple access points. If one location experiences an outage or becomes inaccessible, the data can still be retrieved from other locations, ensuring continuous access to the data.
- **Reduced Latency:** Replication helps reduce latency by placing data closer to the users or the applications that need it. This is particularly beneficial in distributed systems where data is accessed from multiple geographical locations.
- **Increased Bandwidth and Throughput:** Replication can also help distribute the load across multiple locations, increasing overall bandwidth and throughput. This is achieved by allowing multiple copies of the data to be accessed simultaneously, reducing the load on any single location.

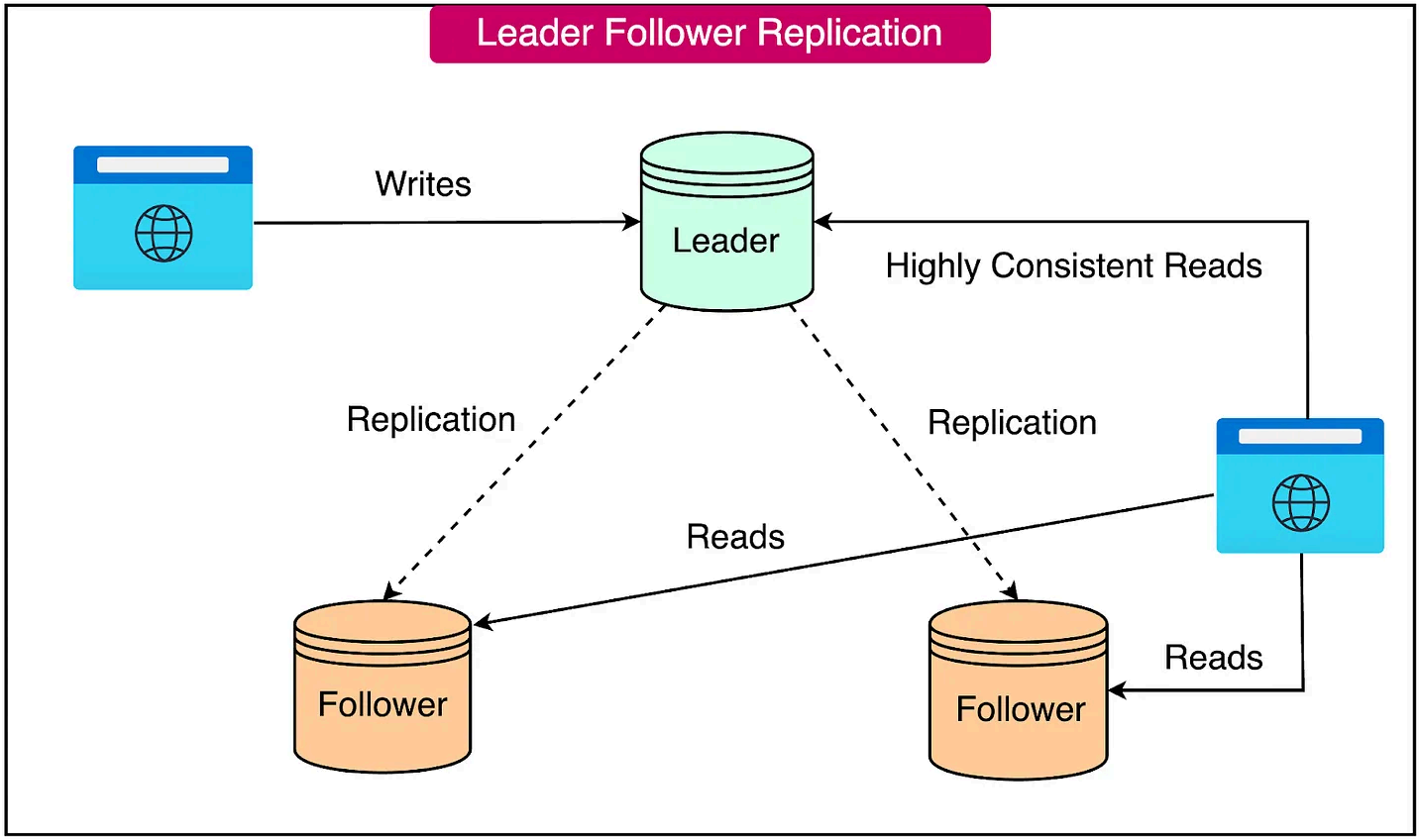
While the concept of replication is straightforward, there are various models for replicating data. Let's look at them in detail.

Leader-Follower Replication Model

The leader-follower model is a widely used approach in data replication, particularly when balancing load between nodes is a priority. This model allows systems to handle a higher volume of requests by parallelizing reads and writes across multiple nodes.

In the leader-follower model, one node is designated as the leader, while the others are followers.

The leader node is responsible for handling all write operations, ensuring that data consistency is maintained. Reads, however, can be directed to any node, including the followers.

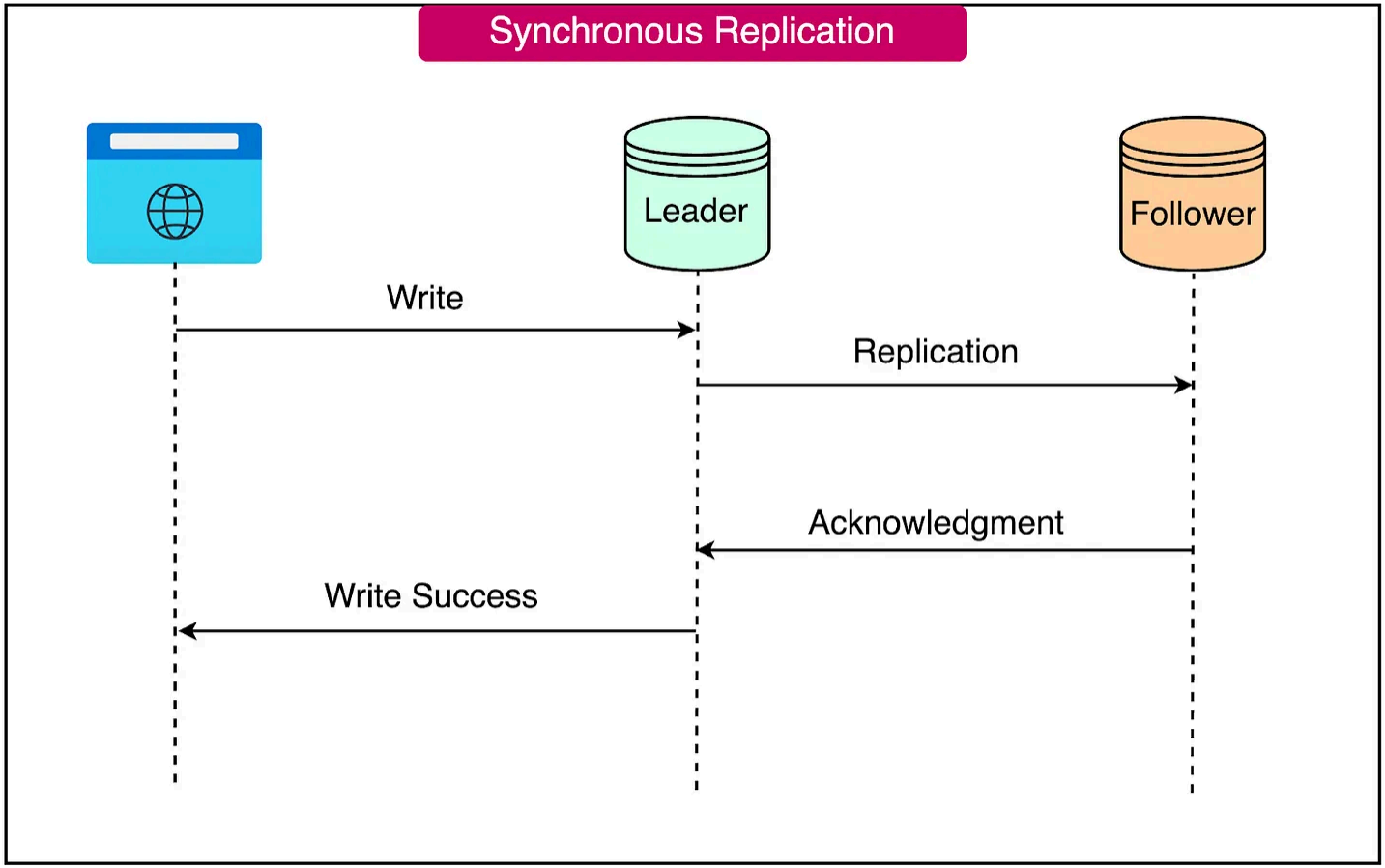


There are two approaches to implementing leader-follower replication: synchronous and asynchronous.

Synchronous Replication

Synchronous replication is a method used within the leader-follower model to ensure strong data consistency.

With synchronous replication, each write operation to the leader node waits for an acknowledgment that the data has been successfully replicated to all follower nodes before completing. This ensures that all nodes, including the followers, have the latest writes before any new write operation occurs.

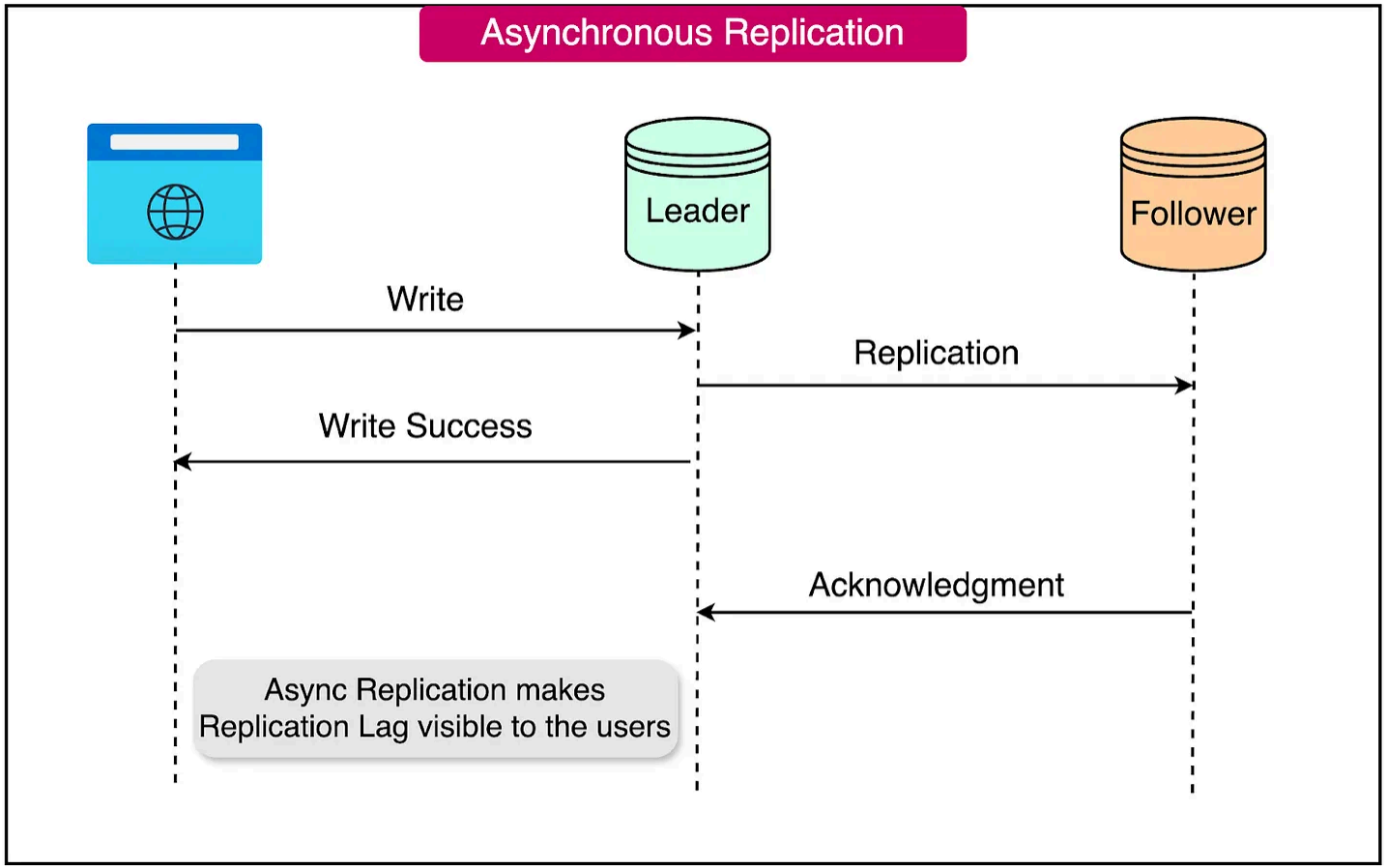


On the downside, the need for replication confirmation introduces additional latency to write operations. Each write operation must wait for acknowledgments from all follower nodes, which

can impact performance.

Asynchronous Replication

In asynchronous replication, the leader node completes write operations without waiting for replication confirmation from the follower nodes. This approach significantly reduces write latency, as the leader node does not need to wait for acknowledgments from the followers.



Asynchronous replication improves performance by allowing the leader node to complete writes quickly, without the overhead of waiting for replication confirmations from the followers. This reduces write latency and improves overall system responsiveness.

On the downside, asynchronous replication introduces a small window of inconsistency between the leader and follower nodes. The time it takes for the data to be replicated from the leader to the followers is known as the replication lag.

Multi-Leader Replication

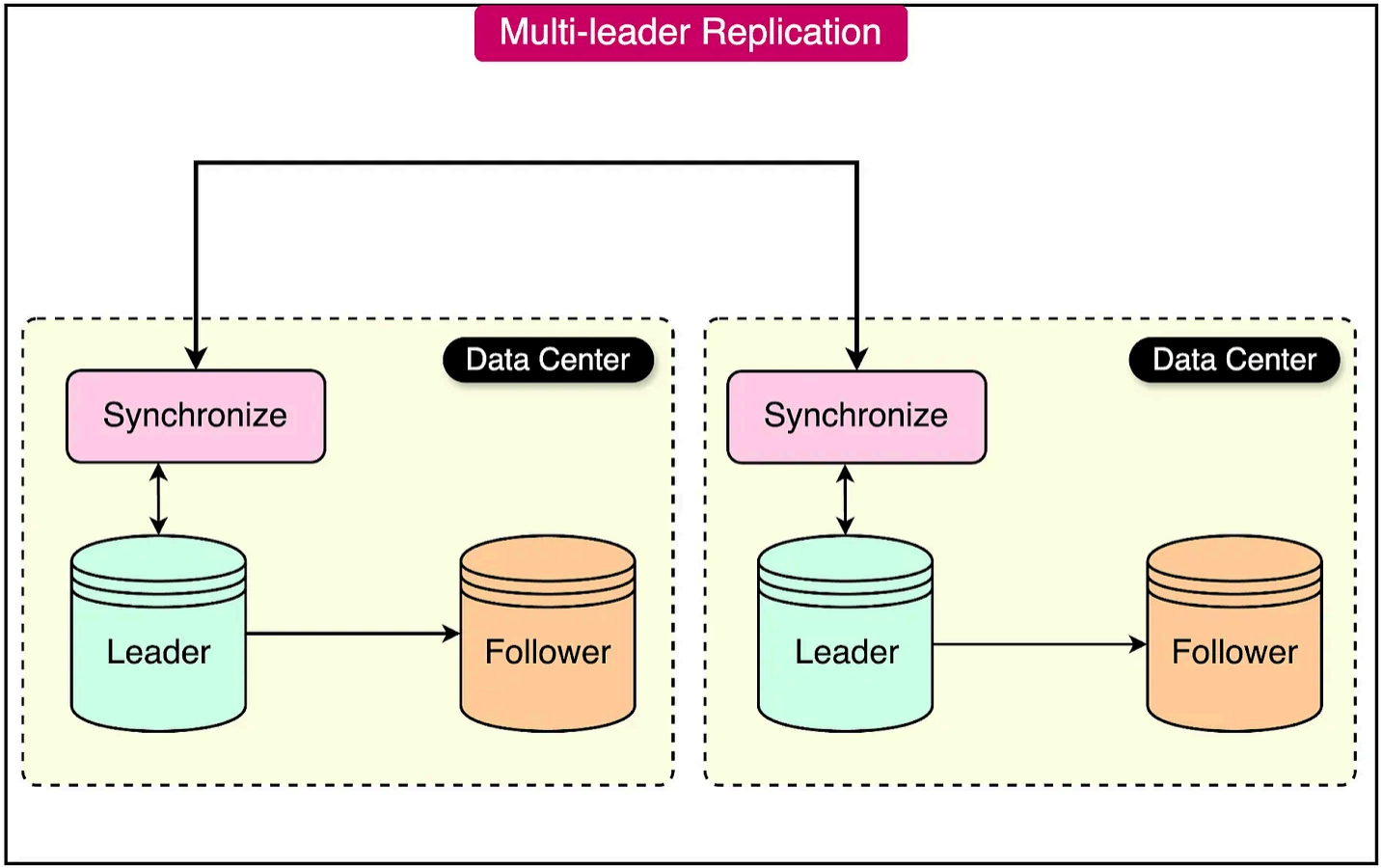
Leader-based replication has a significant downside: there is only one leader, and all the write operations must go through it. If the connection to the leader is disrupted for any reason, such as a network interruption, the write operations to the database become impossible.

A natural extension of the leader-based replication model is to allow more than one node to accept writes.

This model is known as multi-leader replication, also referred to as active-active replication. In this setup, each node acts as both a leader and a follower simultaneously.

In a multi-leader configuration, each node processes write operations independently and forwards the data changes to all the other nodes. This ensures that all nodes remain consistent with each other.

See the diagram below for an example of multi-leader replication where leader nodes are located in different data centers.



While the primary advantage of multi-leader replication is improved availability, it can also introduce additional complexity. For example, conflict resolution mechanisms are necessary to handle situations where different nodes may have different versions of the data due to concurrent writes.

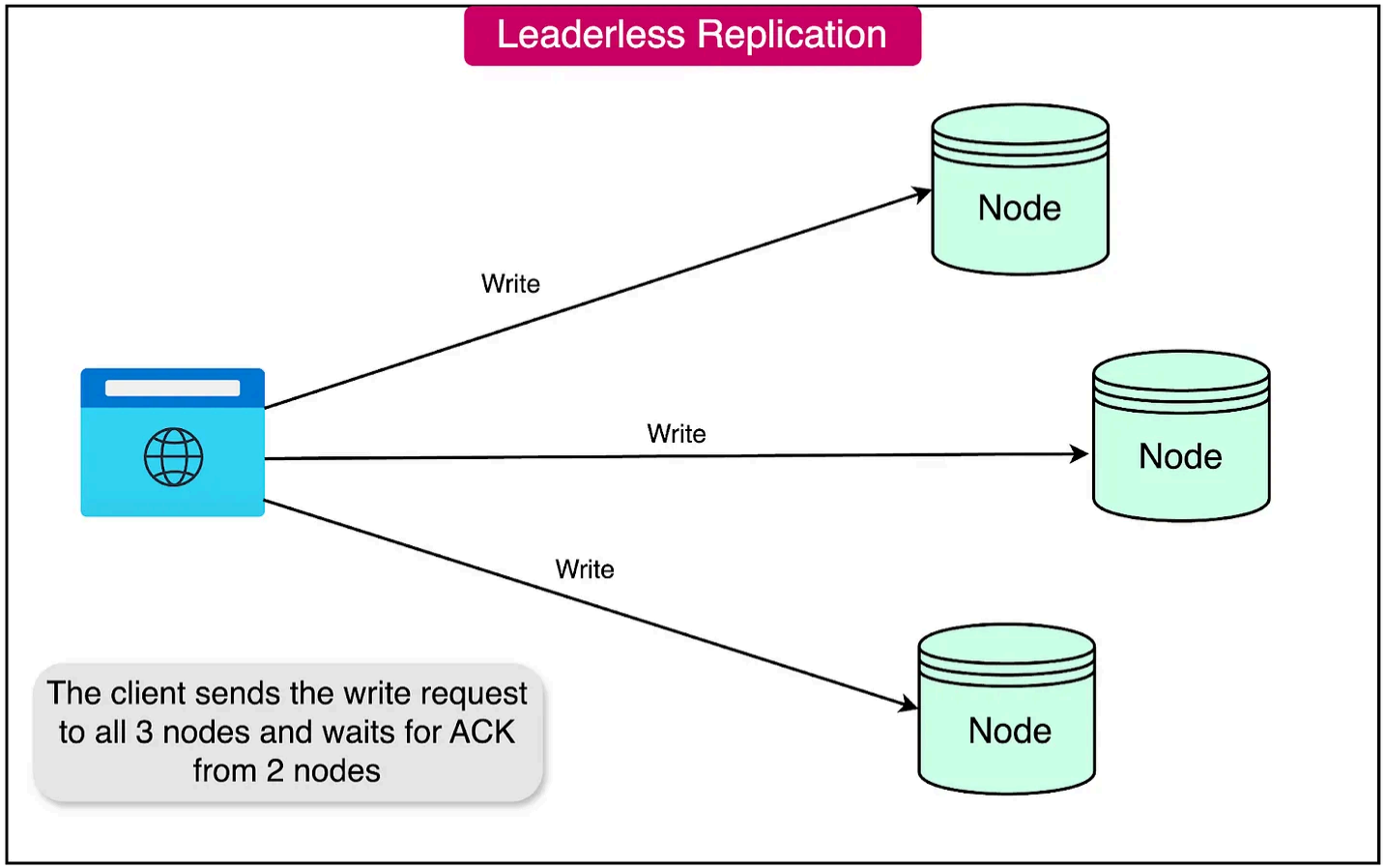
Leaderless Replication

Some data storage systems adopt a leaderless approach, abandoning the traditional concept of a leader node and allowing any replica to accept writes from clients.

In a leaderless setup, clients send their writes directly to multiple replicas in parallel. Alternatively, a coordinator node may handle this task on behalf of the client.

Here's a detailed look at the process:

- **Client Write Operation:** The client or user sends a write request to all available replicas in parallel. For example, if there are three replicas, the client will send the write to all three simultaneously.
- **Replica Acknowledgment:** The available replicas process the write and send acknowledgment responses to the client. If a sufficient number of replicas (e.g., two out of three) acknowledge the write, the client considers the operation successful and ignores any missing acknowledgments from unavailable replicas.
- **Handling Unavailable Replicas:** If one of the replicas is unavailable during the write operation, it will miss the write. When this unavailable replica comes back online, it will not have any writes that occurred while it was down. This can lead to stale (outdated) values being returned if clients read from this replica.



To mitigate the issue of stale values, read operations in leaderless systems are designed to query multiple nodes in parallel:

- **Parallel Read Requests:** When a client reads from the database, it sends read requests to several nodes simultaneously. This ensures that the client can obtain the most up-to-date value by comparing responses from different nodes.

- **Version Numbers:** To determine which value is newer, version numbers are used. The client receives responses from multiple nodes and uses the version numbers to identify the most recent value. This approach helps resolve conflicts and ensure data consistency across different replicas.

Sharding

Sharding is an architectural pattern designed to address the challenges of managing and querying large datasets.

It involves splitting a large database into smaller, more manageable parts called shards. The main benefits of database sharding are as follows:

- **Scalability:** The primary motivation behind sharding is to achieve horizontal scalability. By distributing a large dataset across multiple shards, the query load can be spread across multiple nodes. Each node can independently execute queries for its assigned data, reducing the load on any single node. Additionally, new shards can be added dynamically at runtime without the need to shut down the application for maintenance.
- **Improved Performance:** Retrieving data from a single large database can be time-consuming due to the vast number of rows that need to be searched. In contrast, shards contain a smaller subset of rows compared to the entire database. This reduced search space results in faster data retrieval, as queries have fewer rows to process. Consequently, sharding improves the overall performance of the system by reducing the time required to execute queries.

- **Availability:** In a monolithic database architecture, if the node hosting the database fails, the dependent application experiences downtime. Database sharding mitigates this risk by distributing the data across multiple nodes. In the event of a node failure, the application can continue to operate using the remaining shards, ensuring minimal downtime.

There are several techniques used for sharding databases, each with its own advantages and use cases:

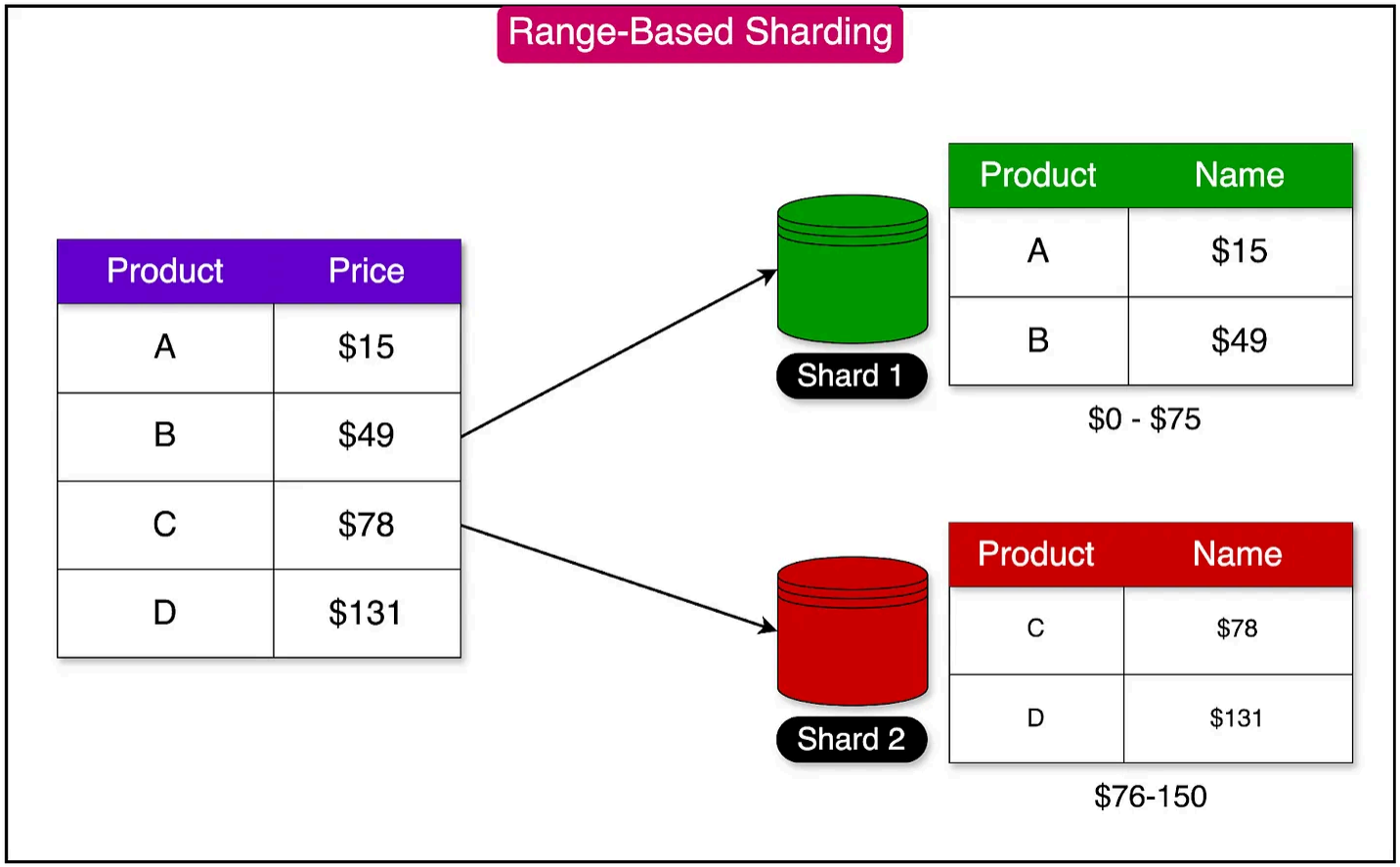
Range-Based Sharding

Range-based sharding involves splitting database rows based on a range of values.

Each shard is assigned a continuous range of keys, spanning from a minimum to a maximum value.

The keys within each shard are maintained in a sorted order to enable efficient range scans. This approach is particularly useful for queries that operate on a specific range of values.

See the diagram below that depicts range-based sharding.



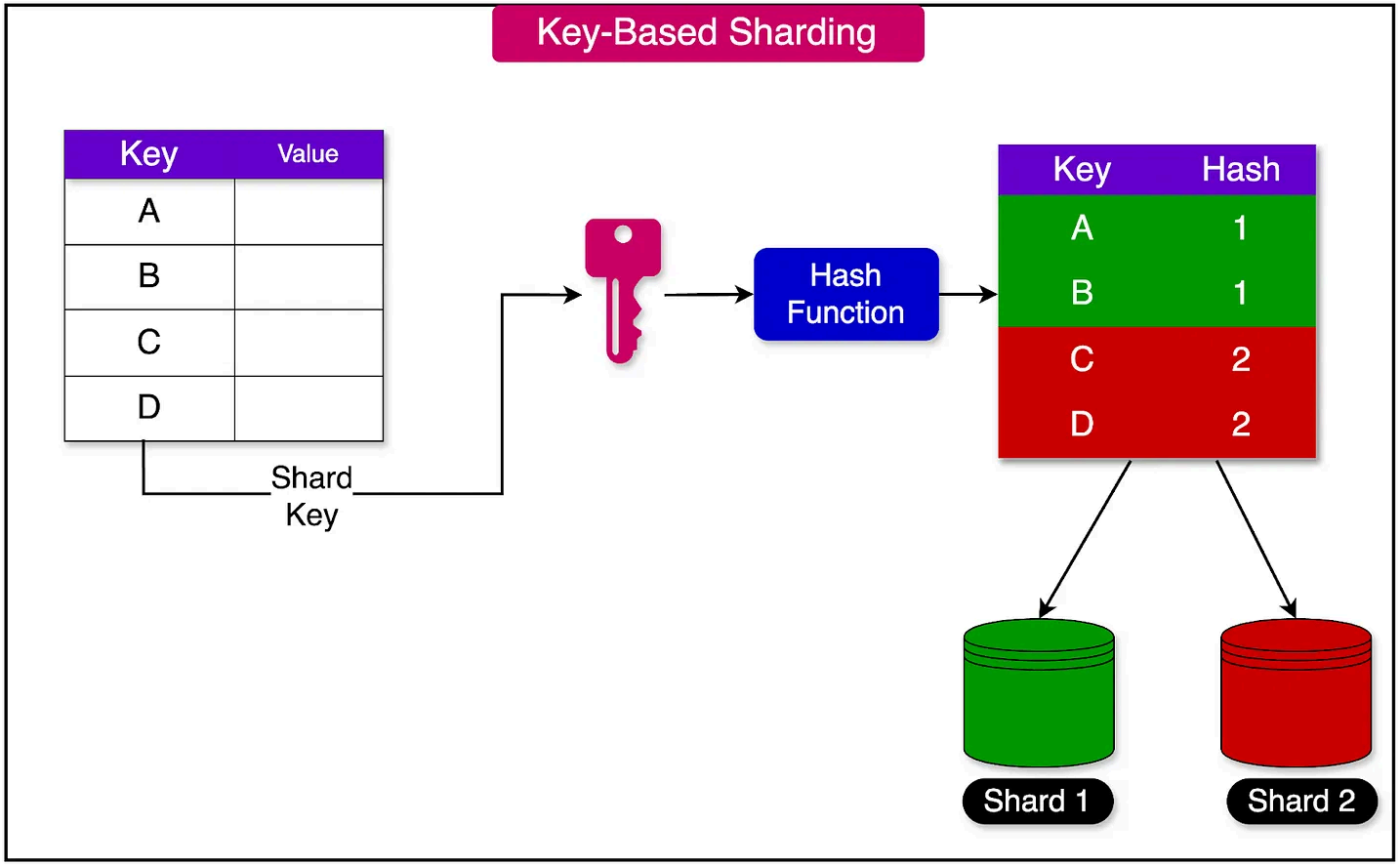
Key or Hash-Based Sharding

Key-based sharding, also known as hash-based sharding, assigns a particular key to a shard using a hash function.

A well-designed hash function is crucial for achieving a balanced distribution of keys across shards. Instead of assigning a range of keys to each shard, hash-based sharding assigns a range of hashes to each shard.

Consistent hashing is a technique often used to implement hash-based sharding, ensuring that the distribution of keys remains balanced even when shards are added or removed.

See the diagram below to get a better idea of hash-based sharding

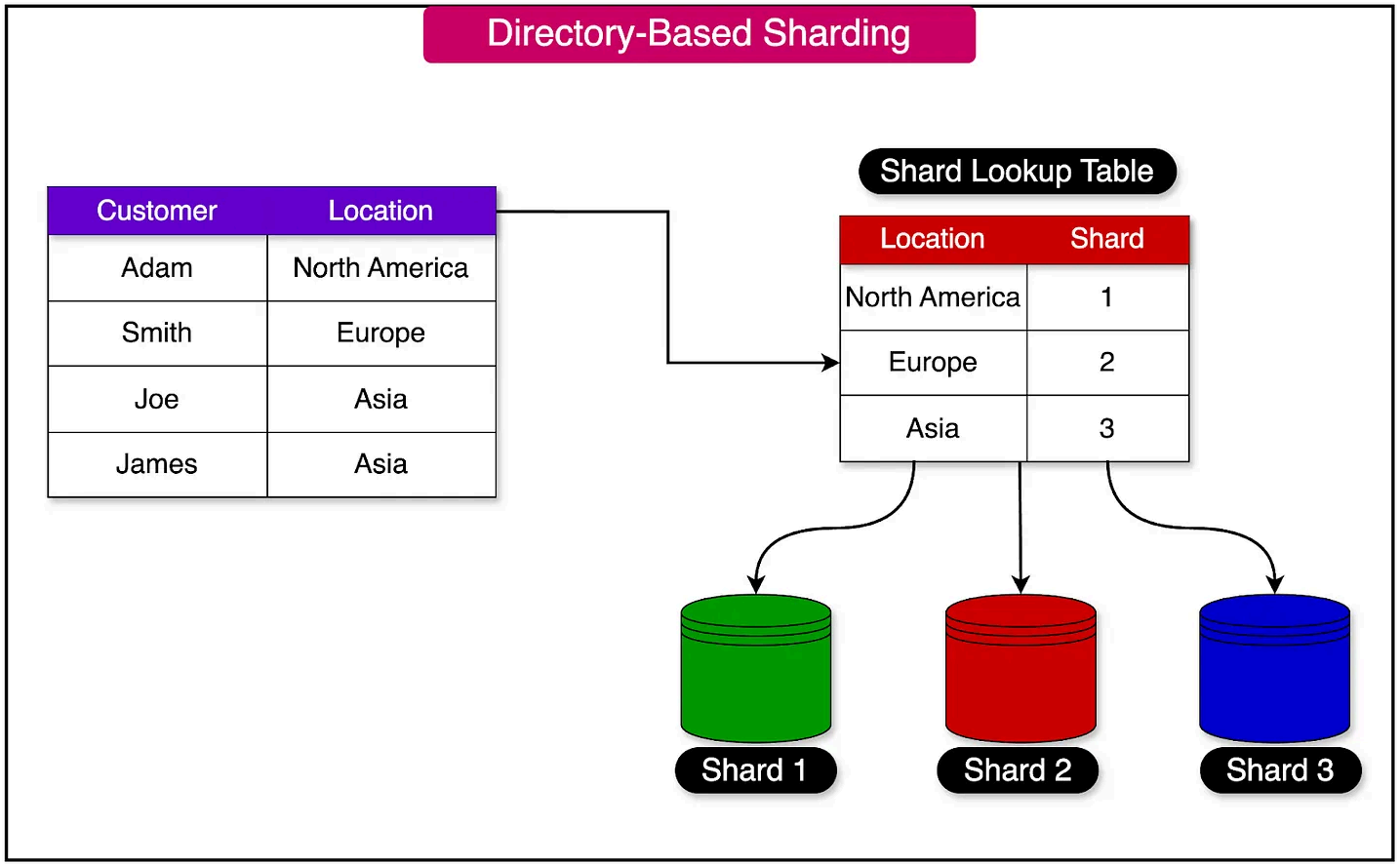


Directory-Based Sharding

Directory-based sharding relies on a lookup table to determine the distribution of records across shards.

This lookup table serves as a directory or an address book, mapping the relationship between the data and the specific shard where it resides.

The table is stored separately from the shards themselves, providing a centralized mechanism for managing the distribution of data.

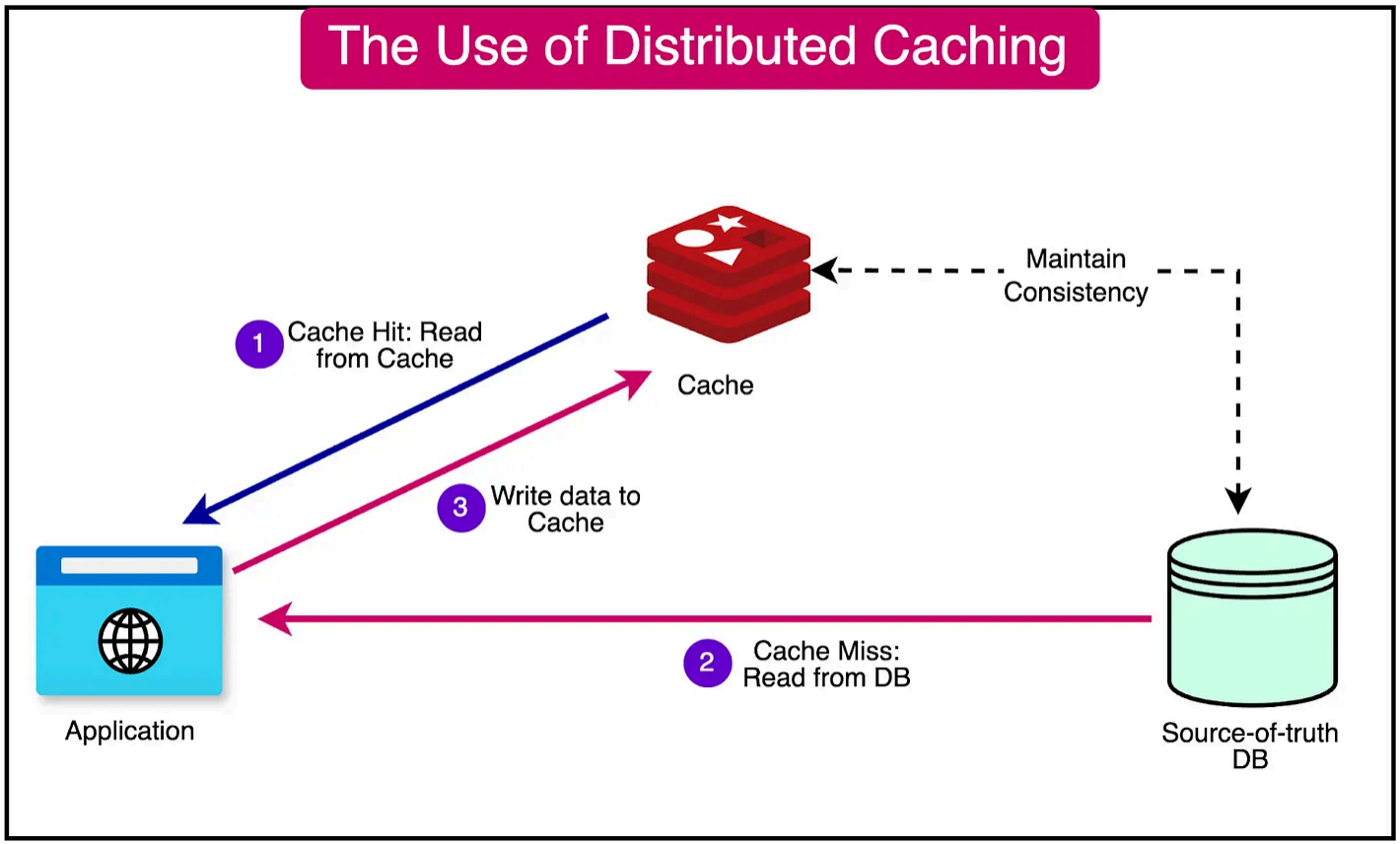


Distributed Caching

A distributed caching system is an in-memory data storage layer that acts as an intermediary between an application and its database.

Storing frequently accessed data in RAM significantly reduces the time required to retrieve data compared to querying a disk-based database.

The diagram below shows how a cache can be used with a database.



Distributed caching offers several key benefits that enhance the performance and scalability of applications:

- **Reduced Database Load:** By serving data from the cache, fewer queries are directed to the database. This reduces the load on the database, allowing it to handle more critical operations efficiently.

- **Improved Response Times:** In-memory access is substantially faster than retrieving data from a disk-based database. This results in quicker response times, enhancing the overall user experience.
- **Increased Scalability:** Caching helps applications handle a higher number of concurrent users. By offloading frequently accessed data to the cache, the system can manage increased traffic without compromising performance.

Several key components are essential for a robust distributed caching system:

Cache Clusters

- **Redis:** Redis is an in-memory data structure store that supports complex data structures such as lists, sets, and hashes. It includes built-in replication and high availability features, making it a reliable choice for distributed caching.
- **Memcached:** Memcached is a distributed memory caching system that operates as a simple key-value store. It is optimized for speed and efficiency, making it suitable for applications requiring fast data retrieval.

Consistent Hashing for Cache Key Distribution

Consistent hashing maps both cache nodes and keys to a circular keyspace. When adding or removing nodes, only a fraction of keys need to be remapped, ensuring an even distribution of data across cache nodes.

This approach has several advantages:

- Minimizes cache misses when scaling the cache cluster,

- Provides better load balancing across cache nodes.
- Simplifies the process of adding or removing cache nodes.

Cache Invalidation Strategies

Dealing with stale data is one of the most crucial aspects of a caching solution. Stale data occurs when the data in the cache diverges from the source-of-truth database. This often requires invalidating the cache entries.

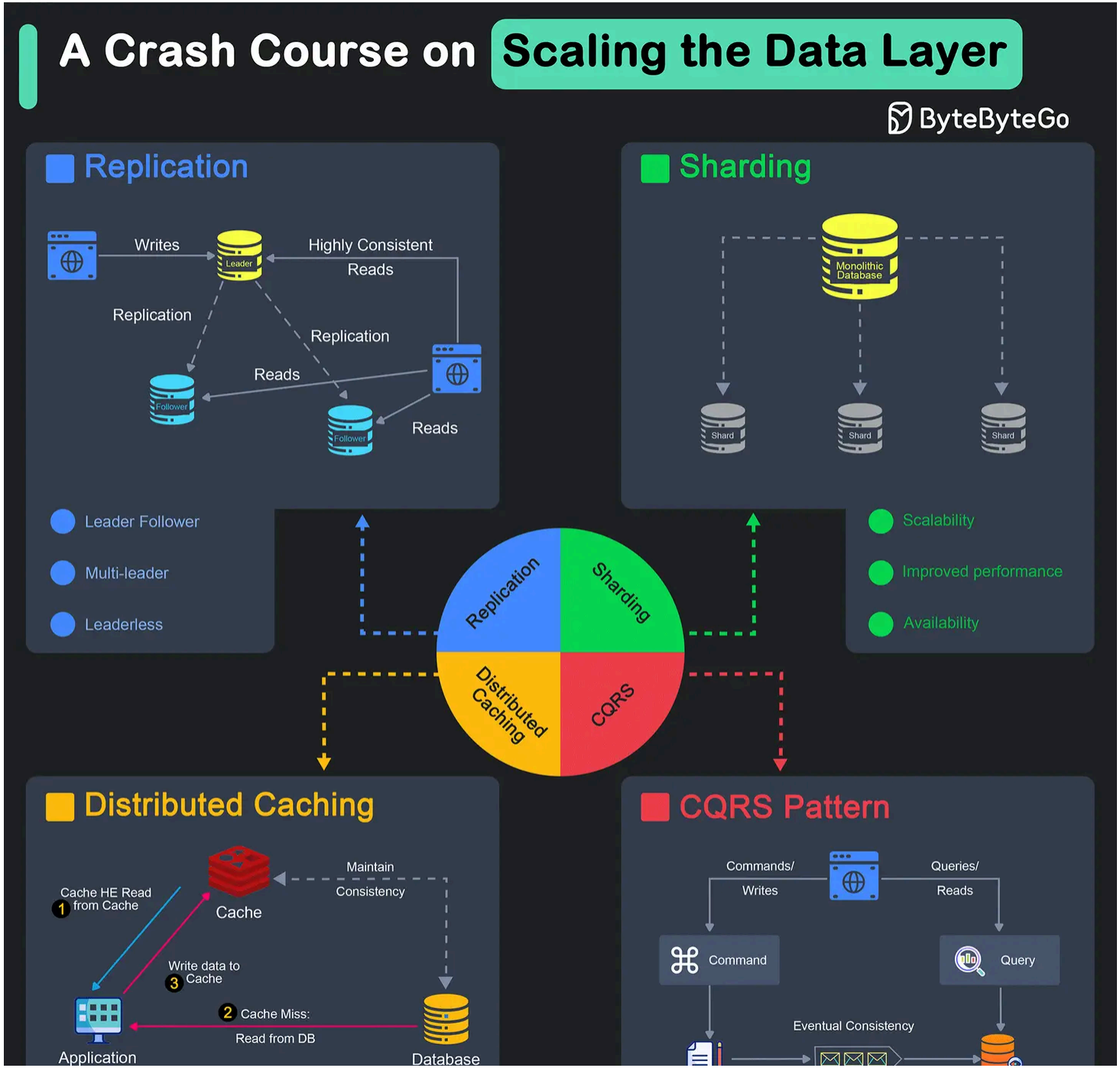
Some common cache invalidation strategies are as follows:

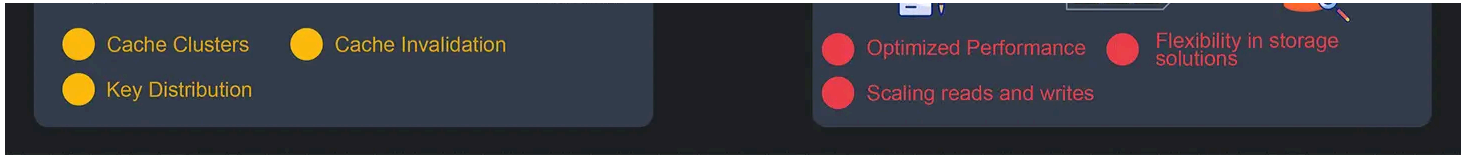
- **Time-Based Expiration:** This strategy involves setting a Time To Live (TTL) for each cached item. The cache automatically removes stale data after the specified time, ensuring that the cached data remains fresh.
- **Event-Based Invalidation:** This approach involves invalidating or updating the cache when data changes in the database. It requires coordination between the application and the cache layer to ensure the cache remains consistent with the database.
- **Version-Based Invalidation:** This strategy assigns version numbers to cached data. When the data changes, the version number is updated, forcing a cache refresh. This ensures that the cache remains consistent with the latest data.
- **Write-Through Caching:** In this approach, the cache and the database are updated simultaneously when data changes. While this ensures cache consistency, it may impact write performance due to the additional overhead of updating both the cache and the database.

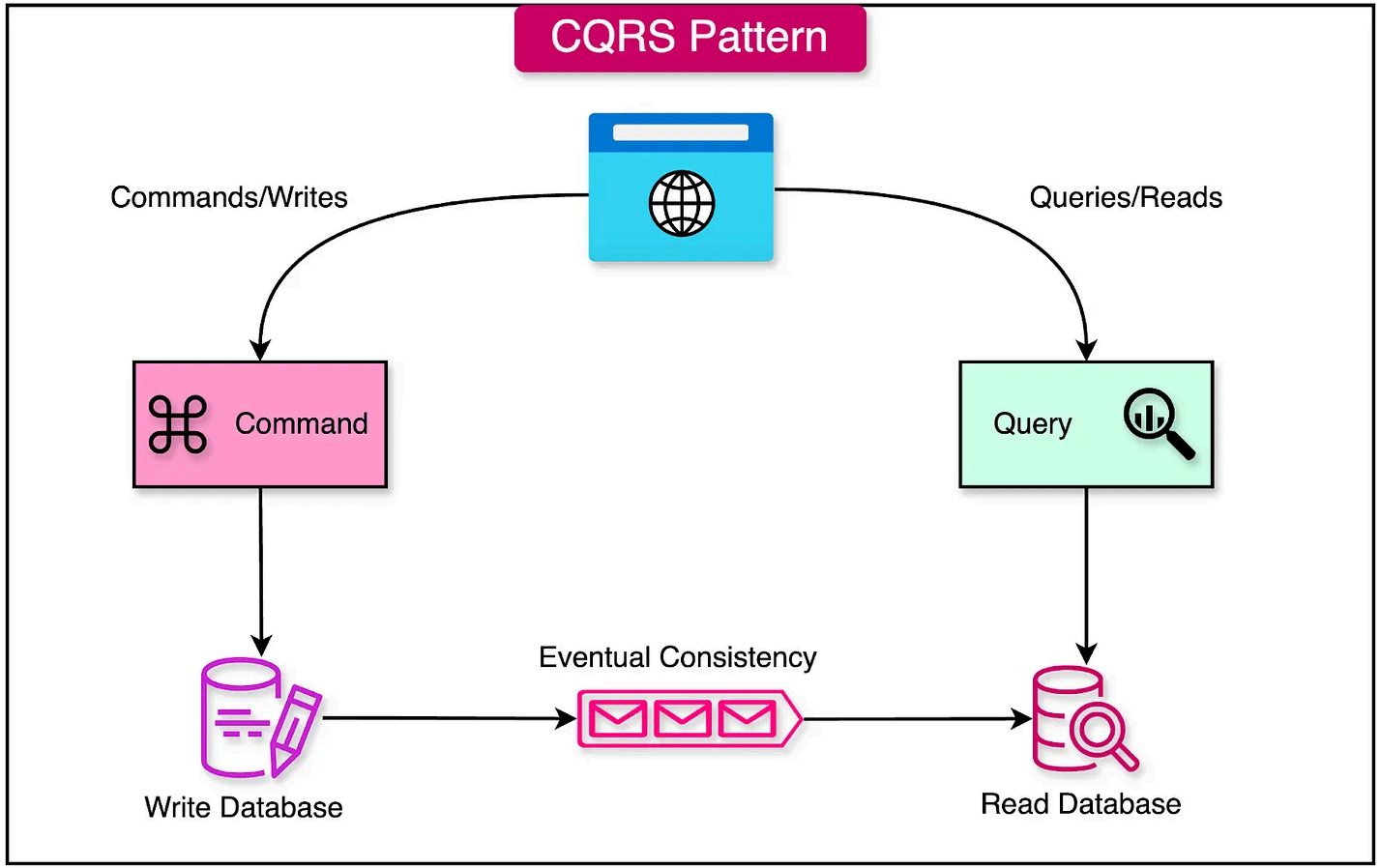
Command-Query Responsibility Segregation

Command Query Responsibility Segregation (CQRS) is a design pattern that separates the responsibilities of handling read and write operations for a data store. This separation allows for optimized performance, scalability, and flexibility in managing data.

The diagram below shows one possible approach for implementing the CQRS pattern where different databases for commands and queries are used.


A Crash Course on Scaling the Data Layer
ByteByteGo
Replication
Writes
Leader
Highly Consistent
Reads
Replication
Replication
Reads
Follower
Follower
Reads
Leader Follower
Multi-leader
Leaderless
Sharding
Monolithic Database
Shard
Shard
Shard
Scalability
Improved performance
Availability
Replication
Sharding
Distributed Caching
CQRS
Distributed Caching
Maintain
Consistency
Cache HE Read
1 from Cache
Cache
Write data to
3 Cache
2 Cache Miss:
Read from DB
Application
Database
CQRS Pattern
Commands/
Writes
Queries/
Reads
Command
Query
Eventual Consistency





CQRS is built around several key concepts that distinguish it from traditional data storage approaches:

- **Commands:** Commands are operations that change the state of the system. These are write operations that modify the data, such as creating, updating, or deleting records.
- **Queries:** Queries are operations that return data without modifying the state of the system. These are read operations that retrieve data from the store.
- **Separate Models:** In CQRS, different data models are used for read and write operations. The write model is typically optimized for handling commands and ensuring data consistency, while the read model is optimized for querying and retrieving data efficiently.

Benefits of CQRS

The CQRS pattern offers several benefits that enhance the performance, scalability, and flexibility of the application's data layer:

- **Optimized Performance:** By separating read and write operations, CQRS allows for optimized performance on both sides. The write model can be optimized for handling commands efficiently, while the read model can be optimized for fast data retrieval.
- **Scalability:** CQRS enables the read and write sides to be scaled independently. This means the system can handle increased read traffic without impacting the write operations, and vice versa. This independent scaling capability is crucial for systems with varying read and write workloads.
- **Flexibility in Storage Technologies:** CQRS enables the flexibility of choosing different storage technologies for read and write operations. For example, the write model might use a relational database for strong consistency, while the read model could use a NoSQL database or a caching layer for faster query performance.

Challenges of CQRS

While CQRS offers multiple benefits, it also introduces some challenges:

- **Increased Complexity:** Implementing CQRS can add complexity to the system architecture. Managing separate models for read and write operations requires careful design and coordination to ensure data consistency and integrity.
- **Potential Data Consistency Issues:** Because CQRS uses separate models for read and write operations, there is a potential for data consistency issues. Ensuring that the read model reflects the latest changes from the write model can be challenging, especially in real-time systems.
- **More Complex Deployment and Maintenance:** CQRS systems require more complex deployment and maintenance processes compared to traditional monolithic architectures. The separate read-and-write models need to be deployed, monitored, and maintained independently, which can increase operational overhead for the development team.

Summary

Horizontally scaling the data layer is one of the most important pieces to scale the application as a whole. There are various techniques to achieve this. However, each has pros and cons.

Let’s go over the techniques we looked at in the article in brief:

- Replication is a method of copying data from one location to another, ensuring that the data is available when and where it is needed.

- There are three types of replication models: leader-follower, multi-leader, and leaderless replication.
- Sharding is an architectural pattern designed to address the challenges of managing and querying large datasets.
- Different types of sharding strategies include range-based sharding, key-based sharding, and directory-based sharding.
- A distributed caching system is an in-memory data storage layer that acts as an intermediary between an application and its database.
- Some important components of a distributed caching setup include cache clusters, consistent hashing algorithms, and cache invalidation strategies.
- Command Query Responsibility Segregation (CQRS) is a design pattern that separates the responsibilities of handling read and write operations for a data store. This allows independent scaling for reads and writes.
- CQRS is built around key concepts like commands, queries, and separate models.

177 Likes · 10 Restacks

Comments

Write a comment...