Published by :
http://www.ijert.org

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 01, January-2025

Designing Resilient Systems: A Guide to Distributed Caching for Modern Applications

Vigneshwaran Manivelmurugan
Senior Lead Software Engineer
Capital One
Richmond, Virginia, USA

Abstract – **Distributed caching has emerged as a cornerstone for optimizing performance and scalability in cloud-based and microservice architectures. As applications grow in complexity and user expectations increase, ensuring fast, reliable access to data becomes a critical challenge. Distributed caching addresses these challenges by storing frequently accessed data in-memory, closer to the application, thereby reducing latency, backend load, and operational costs. However, distributed caching is not a replacement for traditional databases; instead, it complements them by enabling low-latency data access for high-demand use cases while maintaining durability and consistency through database integration. This article examines the principles of distributed caching, outlines effective strategies for its implementation, and provides a comparative analysis of popular tools like Redis, Memcached, and AWS ElastiCache to guide developers in selecting the most suitable solution for their needs.**

Keywords – **Microservice, Distributed System, Cache, High Performance Computing**

I. INTRODUCTION

In the realm of modern cloud-native architectures, the demand for low-latency applications has never been higher. Distributed systems and microservices have become standard, but with this shift comes the challenge of ensuring fast and reliable access to data. Performance bottlenecks, data inconsistency, and scalability constraints are common hurdles as applications scale. Traditional centralized databases often struggle to handle the high read and write loads generated by distributed systems, resulting in latency and degraded user experience. Furthermore, geographically distributed users exacerbate these issues with network-induced delays, while frequent database access drives up operational costs.

Distributed caching provides a powerful solution by storing frequently accessed data closer to the application, reducing the dependency on databases and enhancing responsiveness. By leveraging in-memory storage, distributed caching ensures that applications can handle high traffic with minimal delays, addressing performance and scalability challenges head-on. This article explores the principles and strategies for effective caching in modern architectures and compares tools like Redis, Memcached, and AWS ElastiCache to guide the selection process for specific use cases. This article delves into the core principles of distributed caching, examining its relevance in modern architectures, common pitfalls, and best practices. We'll also evaluate most popular caching solutions, to guide readers in choosing the right tool for their needs.

Distributed caching optimizes data retrieval by storing data closer to the application. However, it is not designed to replace traditional databases but rather to complement them. Distributed caches offer faster access to frequently used data, making them ideal for low-latency applications. The cost of memory for caches is generally higher than the compute cost for databases, which makes it impractical to use caching for all data. Moreover, the complexity of managing sharding and maintaining consistency across cache nodes adds additional challenges.

Instead, distributed caches enhance performance by storing the most frequently accessed data, reducing the load on primary databases. Writing all updates to the database ensures data durability and provides a mechanism to reconstruct the cache when needed. Additionally, distributed caching solutions like Redis and Memcached support snapshot features, enabling data recovery and cache population during restarts or failures. A well-designed strategy integrates distributed caching with databases, leveraging the strengths of both to deliver reliable and scalable systems.

II. DESIGN STRATEGY TO EFFECTIVELY USE DISTRIBUTED CACHING

Identify Caching Candidates: Designing an effective caching strategy begins with identifying suitable data for caching. Not all data is ideal; the focus should be on frequently accessed, read-heavy datasets with relatively low volatility, such as product catalogs, user profiles, and session data. To further optimize retrieval, consider precomputing and storing aggregated data rather than raw datasets. For instance, instead of caching every transaction record, precompute and store aggregate statistics like total daily sales or active user counts per region. This approach minimizes computational overhead during real-time queries, improves data access speeds, and provides consistent, summarized insights for frequently requested data. By strategically identifying and preparing caching candidates, systems can achieve higher efficiency and better source utilization.

Define Cache Policies: Designing effective cache policies is critical for maintaining performance and cost efficiency in distributed systems. Eviction policies such as Least Recently Used (LRU) or Time-To-Live (TTL) play a central role in managing memory by ensuring that stale or irrelevant data is systematically removed. Without these strategies, caches can become overloaded, leading to excessive memory consumption and increased compute

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 01, January-2025

costs, as well as degraded performance due to slower retrieval speeds. This often necessitates costly full memory clean-ups or scaling operations. When crafting a caching strategy, factors like data access patterns, expected volatility, and memory constraints should guide the choice of eviction policies. TTL policies help to balance freshness and resource efficiency by automatically expiring data after a set period, while LRU ensures that infrequently accessed items are prioritized for removal. The absence of a well-defined policy can result in stale data inconsistencies that compromise application behavior and user experience. An intelligently designed caching strategy optimizes resource utilization, keeps costs under control, and ensures high performance by maintaining a clean and relevant cache.

Use Partitioning: Designing an effective partitioning strategy is essential for distributed caching to enhance scalability and reliability. Consistent hashing is a widely used method that distributes keys evenly across available cache nodes, ensuring minimal disruption when nodes are added or removed which is common in cloud environments. This approach reduces the likelihood of hotspots and maintains a balanced workload. In contrast, range-based partitioning assigns keys to predefined ranges on specific nodes, making it suitable for ordered datasets but prone to hotspots if access patterns are uneven. Modulo-based partitioning uses a modulus operation to determine node allocation, offering simplicity but causing significant rebalancing when nodes change. Dynamic partitioning, which relies on real-time load-balancing algorithms, is ideal for highly variable workloads but comes with added complexity in implementation. When designing the partitioning strategy, factors such as data access patterns, scalability needs, and fault tolerance requirements must be carefully evaluated to choose the most appropriate method for the application's demands. A well-architected partitioning strategy ensures optimized resource utilization, maintains high performance, and supports seamless scaling.

Ensure Cache Coherency: Designing a cache coherency strategy requires a clear understanding of the application's data access patterns and consistency requirements. Cache invalidation techniques, such as write-through, write-behind, or explicit invalidation, are essential for maintaining synchronization between the cache and the primary data source. In a write-through strategy, updates are written to both the cache and the database synchronously, ensuring immediate consistency, which is ideal for critical systems requiring real-time accuracy. Write-behind strategies, where updates are batched and asynchronously written to the database, can improve performance but introduce temporary inconsistencies that need careful management to avoid data conflicts. Explicit invalidation offers precise control, allowing specific data entries in the cache to be refreshed or removed, particularly effective for datasets with predictable or low-frequency changes. When designing these strategies, it is vital to account for potential issues such as cache pollution, where stale data occupies valuable memory, and data staleness, which can lead to incorrect application behavior. Effective cache coherency strategies not only enhance system reliability but also optimize resource usage, ensuring that cached data is accurate, relevant, and delivered with minimal overhead.

Monitor and Optimize: Continuously monitor cache performance metrics like hit ratio and latency to identify and resolve inefficiencies. A well-designed monitoring strategy starts by defining the key performance indicators (KPIs) for your cache, such as cache hit ratio, latency, memory usage, and node availability. These metrics provide insight into how effectively the cache is reducing backend load and improving application responsiveness. To design an effective optimization strategy, identify patterns in cache misses and evaluate whether the data being cached aligns with access patterns. Optimization should be an iterative and ongoing process, leveraging both real-time analytics and historical trends to refine the caching strategy continually.

Leverage Cache Hierarchies: When designing a caching strategy, it is crucial to recognize that distributed caching does not necessarily replace local (in-process) caches. Instead, the two can complement each other to provide faster access to frequently used data. Local caches operate within the application's process space, offering the lowest latency by eliminating the need for network calls. They are ideal for storing small, frequently accessed datasets specific to the application's immediate context, such as user session data or configuration settings. Distributed caches, on the other hand, provide a shared, scalable solution for larger datasets that need to be accessed by multiple services or instances. By combining these hierarchies, local caches handle immediate, high-speed retrievals, while distributed caches ensure broader availability and scalability across the system. This layered approach minimizes latency, optimizes resource utilization, and balances the workload effectively, ensuring a robust and efficient caching architecture.

III. DEFINE KEY AND PAYLOAD STRUCTURE

Choosing an appropriate key design is critical in distributed caching to avoid issues like hotspots and hash collisions, which can degrade performance and system reliability. A poorly designed key can result in uneven data distribution across nodes, leading to overloading some nodes while others remain underutilized. To design effective keys, ensure they are unique, evenly distributed, and derived from predictable attributes of the data, such as user IDs, session identifiers, or composite keys incorporating timestamps or categories. Incorporating consistent hashing ensures that keys are mapped effectively to minimize rebalancing when nodes are added or removed.

Payload size also plays a significant role in caching performance. Large payloads can increase memory usage and network latency, reducing overall cache efficiency. To optimize payload size, employ data serialization

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 01, January-2025

techniques like JSON, Protocol Buffers, or MessagePack, selecting a format based on the trade-off between human readability and compactness. Protocol Buffers and MessagePack, for instance, provide compact and efficient serialization, making them ideal for scenarios where minimal memory and bandwidth usage are essential. By designing lightweight payloads and efficient keys, caching strategies can achieve optimal performance and resource utilization.

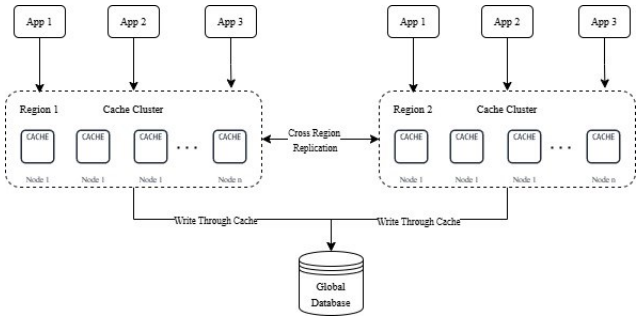

Fig.1. High level design

IV. CHOOSE THE RIGHT CACHE

When selecting a caching solution, the choice between Redis, Memcached, and AWS ElastiCache depends on specific workload characteristics such as read-to-write ratio, data complexity, and scalability requirements.

Redis: Redis is a versatile, high-performance in-memory data store that supports advanced data structures like strings, hashes, lists, and sets. It offers persistence options, enabling durability across system restarts, and supports clustering for high availability and horizontal scaling. Redis's features make it an excellent choice for applications requiring complex operations, pub/sub messaging, or atomic transactions. Its ability to handle a wide range of use cases, such as session storage, real-time analytics, and leaderboards, makes it a robust option for modern applications.

Memcached: Memcached is a lightweight, high-speed caching solution designed for simple key-value storage. It is particularly well-suited for scenarios where simplicity and speed are paramount, such as API response caching or web page fragments. While Memcached excels in lightweight use cases, it lacks the advanced data structures and persistence features offered by Redis, making it less suitable for applications with complex requirements.

AWS ElastiCache: AWS ElastiCache offers a fully managed caching solution that supports both Redis and Memcached. It integrates seamlessly with AWS ecosystems, providing automated scaling, patching, and monitoring capabilities. ElastiCache is ideal for applications running on AWS, as it eliminates operational overhead and allows developers to focus on application logic. Its support for cluster mode enables high availability and horizontal scaling, making it a versatile choice for both simple and complex workloads.

When to Choose What: Redis is best suited for applications requiring advanced data structures, persistence, or high availability through clustering. Its rich feature set makes it an ideal choice for use cases like real-time analytics, pub/sub systems, and session management. Memcached is the preferred option for lightweight, straightforward caching needs where simplicity and speed are critical. For organizations leveraging AWS, ElastiCache provides the flexibility to choose between Redis and Memcached while benefiting from AWS's fully managed infrastructure. By aligning the choice of caching solution with workload requirements and operational constraints, developers can optimize performance and scalability effectively.

V. PERFORMANCE COMPARISON

Distributed caching offers substantial performance improvements compared to traditional databases, particularly in scenarios with high read/write workloads. Databases, while reliable for long-term storage and data integrity, inherently face higher latencies due to disk I/O and the network overhead involved in handling queries. These latencies can increase significantly under heavy workloads. In contrast, distributed caching leverages in-memory storage, drastically reducing data retrieval times and improving system responsiveness.

In our test, solutions like Redis have achieved data retrieval latencies as low as sub-millisecond levels, whereas database queries may take several milliseconds or more, depending on the query complexity and system load. Distributed caches also excel in throughput, often handling over 100,000 requests per second, far surpassing the capabilities of many databases.

The cost dynamics further illustrate their complementary nature. While distributed caches reduce the load on databases by offloading frequent reads, their reliance on memory, which is more expensive than disk-based storage, necessitates efficient cache management. Caches should primarily handle high-demand data, while databases retain responsibility for durable and comprehensive data storage.

Metrics such as cache hit ratio, latency, and throughput provide concrete comparisons. A high cache hit ratio (>90%) indicates that most data requests are fulfilled directly from the cache, significantly reducing database stress. Visual analyses reinforce latency, throughput and cost.

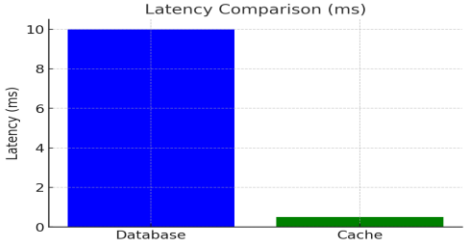

Fig.2. Latency Comparison (Figure shows response times of caches against traditional databases)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 14 Issue 01, January-2025

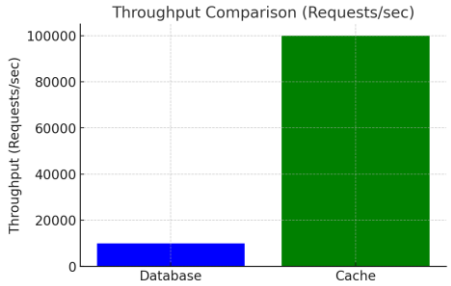

Fig.3. Throughput Comparison (Figure demonstrates the caches capable of serving sustaining concurrent requests)

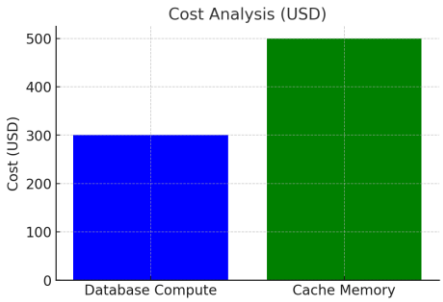

Fig.4. Cost Comparison (Figure demonstrates costs for caches versus compute and storage costs for databases)

VI. CONCLUSION

Distributed caching is a vital component for creating scalable and responsive applications in cloud and microservice environments. By integrating caching solutions into system design, architects can effectively enhance performance and meet low-latency requirements. Solutions like Redis, suitable for complex operations, and Memcached, optimized for lightweight needs, enable developers to tailor strategies for specific workloads.

Distributed caches reduce database dependency by handling high-frequency queries in-memory, ensuring faster response times without compromising resiliency or cost-efficiency. At the same time, databases offer durable storage for data integrity and recovery. This synergy enables recovery and rebuilding of caches during failures, further enhancing system reliability.

By balancing cost and performance, caching strategies prioritize efficient memory utilization while maintaining durability. This thoughtful integration of caching and database systems empowers architects to design high-performance systems that cater to the complex demands of modern applications, ensuring both responsiveness and resilience.

REFERENCES

[1] Redis Documentation: Official Redis documentation covering features like advanced data structures, persistence, and clustering https://redis.io/documentation
[2] Memcached Documentation: Comprehensive guide to Memcached's architecture and use cases for lightweight key-value storage. https://memcached.org/.
[3] AWS ElastiCache Documentation: Insights into AWS's managed caching service with support for Redis and Memcached. http://aws.amazon.com/elasticache/
[4] Google Protocol Buffers Documentation: Official Protocol Buffers documentation, useful for efficient data serialization. https://protobuf.dev/
[5] MessagePack Documentation: Information on MessagePack, a compact and efficient data serialization format. https://msgpack.org/.
[6] Designing Data-Intensive Applications by Martin Kleppmann: Reference for distributed system design principles and best practices, including caching strategies. https://martin.kleppmann.com/2017/03/27/designing-data-intensive-applications.html
[7] Exploring Fine-Grained In-Memory Database Performance for Modern CPUs," in IEEE Transactions on Parallel and Distributed Systems, vol. 34, no. 6, pp. 1757-1772, June 2023, doi: 10.1109/TPDS.2023.3262782.
[8] Performance Evaluation for Distributed Systems https://www.geeksforgeeks.org/performance-evaluation-for-distributed-systems/
[9] High Performance Application With Distributed Caching. https://info.couchbase.com/rs/302-GJY-034/images/High_Performance_With_Distributed_Caching_Couchbase.pdf

AUTHOR PROFILE

Vigneshwaran Manivelmurugan

I am a technology leader with over 15 years of experience in cloud-native architecture, distributed systems, and machine learning engineering. I specialize in delivering mission-critical solutions, building scalable and secure applications, optimizing data pipelines, and modernizing complex platforms. My work has earned recognition, including awards for data engineering excellence and opportunities to judge prestigious industry awards. I am passionate about using technology to create seamless customer experiences and deliver meaningful business value.