What is Distributed Caching?
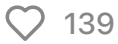
When one cache server is not enough

ASHISH PRATAP SINGH

SEP 03, 2024
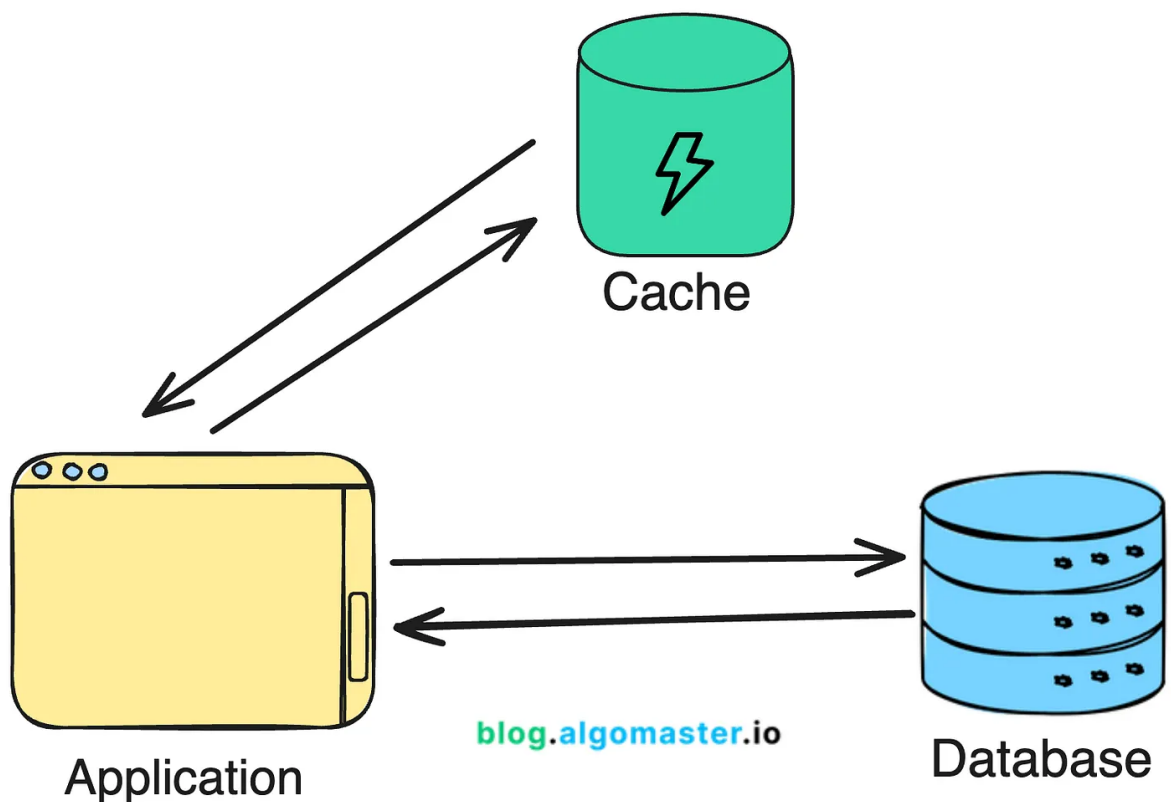
Caching is used to temporarily store copies of **frequently accessed data** in storage layers (such as RAM) to **reduce latency** and load on the server or da

Visualized using **Multiplayer**

When your dataset size is small, it's usually enough to keep all the cache da server.

But as the system gets **bigger**, the cache size also gets bigger and a **single-n**

often falls short when scaling to handle millions of users and massive datas

In such scenarios, we need to **distribute** the cache data across multiple serv

This is where **distributed caching** comes into play.

In this article, we will explore distributed caching in detail, including what it's important, how it works, it's components, challenges, best practices and distributed caching solutions.
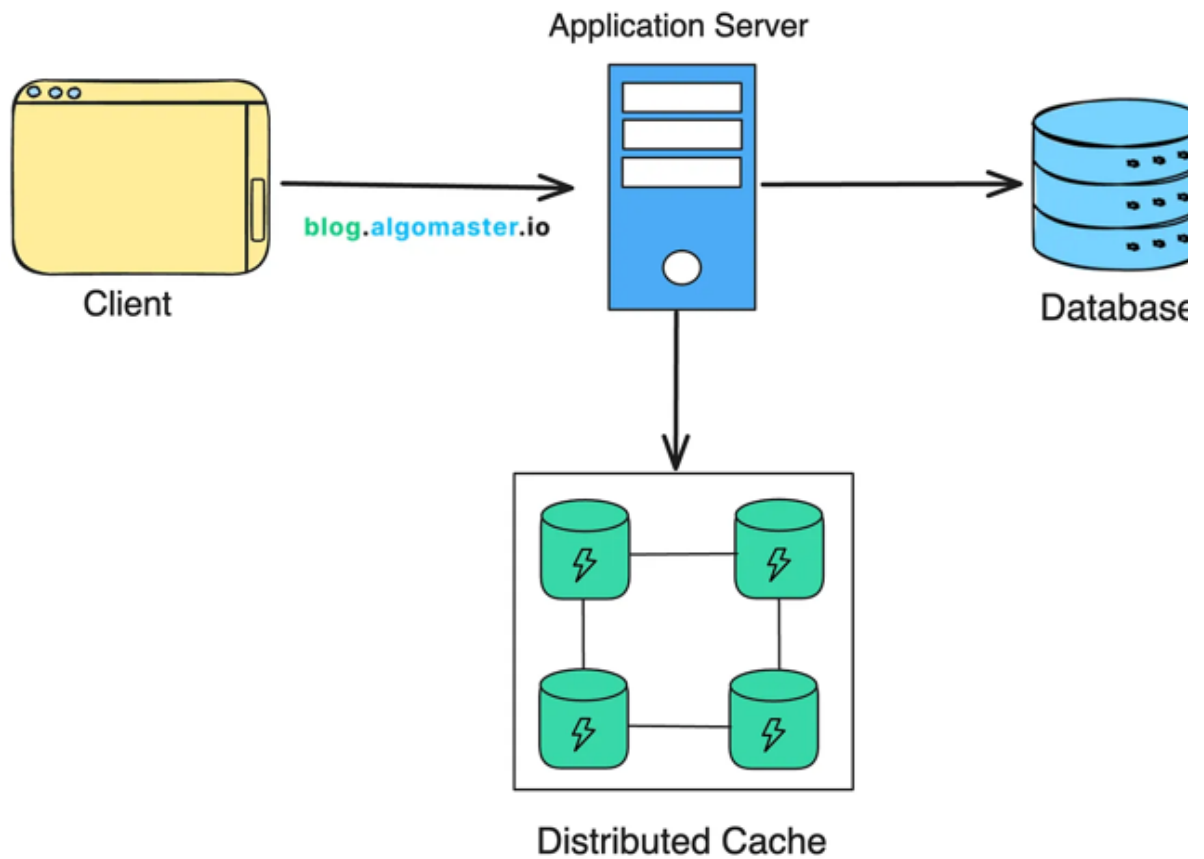
1. What is Distributed Caching?

Distributed caching is a technique where cache data is stored across **multi** (servers) instead of being confined to a single machine.

This allows the cache to **scale horizontally** and accommodate the needs of I **applications**.

Visualized using **Multiplayer**

---

2. Why Use Distributed Caching?

1. Scalability

Distributed caching allows applications to **scale horizontally** by adding mo
nodes.

This helps manage more traffic without a significant drop in performance.

2. Fault Tolerance

Since data is spread across **multiple nodes**, the failure of a single node does
the loss of the entire cache.

Remaining nodes can continue to serve requests allowing the system to rec
gracefully.

3. Load Balancing

By distributing the cache across several nodes, the load is spread **evenly**.

This helps prevent any single node becoming a bottleneck.

3. Components of Distributed Caching

A distributed cache system typically consists of the following components:

1. **Cache Nodes**: These are the **individual servers** where the cache data is
   Each node is a part of the overall cache cluster.

2. **Client Library/Cache Client**: Applications use a **client library** to talk to
   distributed cache. This library handles the logic of connecting to cache
   distributing data, and retrieving cached data.

3. **Consistent Hashing**: This method spreads data **evenly** across cache no
   ensures that adding or removing nodes has minimal impact on the syst

4. **Replication**: To make the system more reliable, some distributed cache
   data across multiple nodes. If one node goes down, the data is still avai
   another.

5. **Sharding**: Data is **split** into shards, and each shard is stored on a differ
   node. It helps distribute the data **evenly** and allows the cache to scale h

6. **Eviction Policies**: Caches implement eviction policies like **LRU** (Least
   Used), **LFU** (Least Frequently Used), or **TTL** (Time to Live) to get rid of
   used data and make space for new data.

7. **Coordination and Synchronization**: Coordination mechanisms like **dis**
   locks or **consensus protocols** ensure that cache nodes remain synchron
   especially when multiple nodes try to change the same data at the same

4. Dedicated Cache Servers vs. Co-located C

When designing a caching strategy for a distributed system, one of the criti
decisions you need to make is **where to host the cache.**

The two primary options are using **dedicated cache servers** or **co-locating t
with application servers.**

1. Dedicated Cache Servers

Dedicated cache servers are standalone machines or virtual instances used
caching.

They are separate from the application servers and are **optimized for cachi**
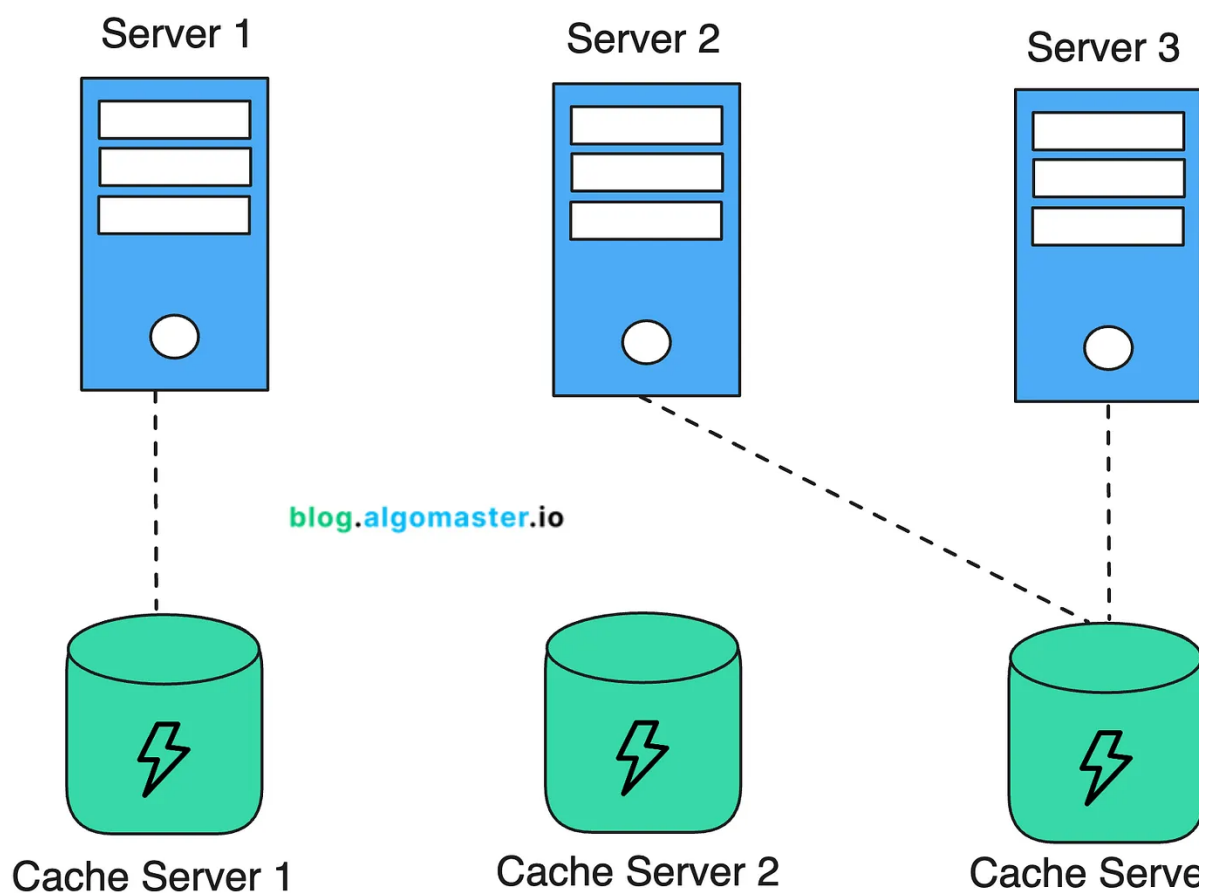
Visualized using **Multiplayer**

Advantages:

- **Scalability:** They can be scaled independently of the application servers add more cache nodes without impacting the application layer.

- **Resource Isolation:** Keeping cache on separate servers means it won't your main servers.

Disadvantages:

- **Cost:** Running dedicated cache servers can be pricey, especially if you such nodes.

- **Network Latency:** Since they are separate from the application servers, take **longer** to fetch data from the cache.

2. Co-located Cache

Co-located cache means running the cache and the application on the same

In this setup, the application and the cache share the **same hardware resou** as CPU, memory, and network interfaces.
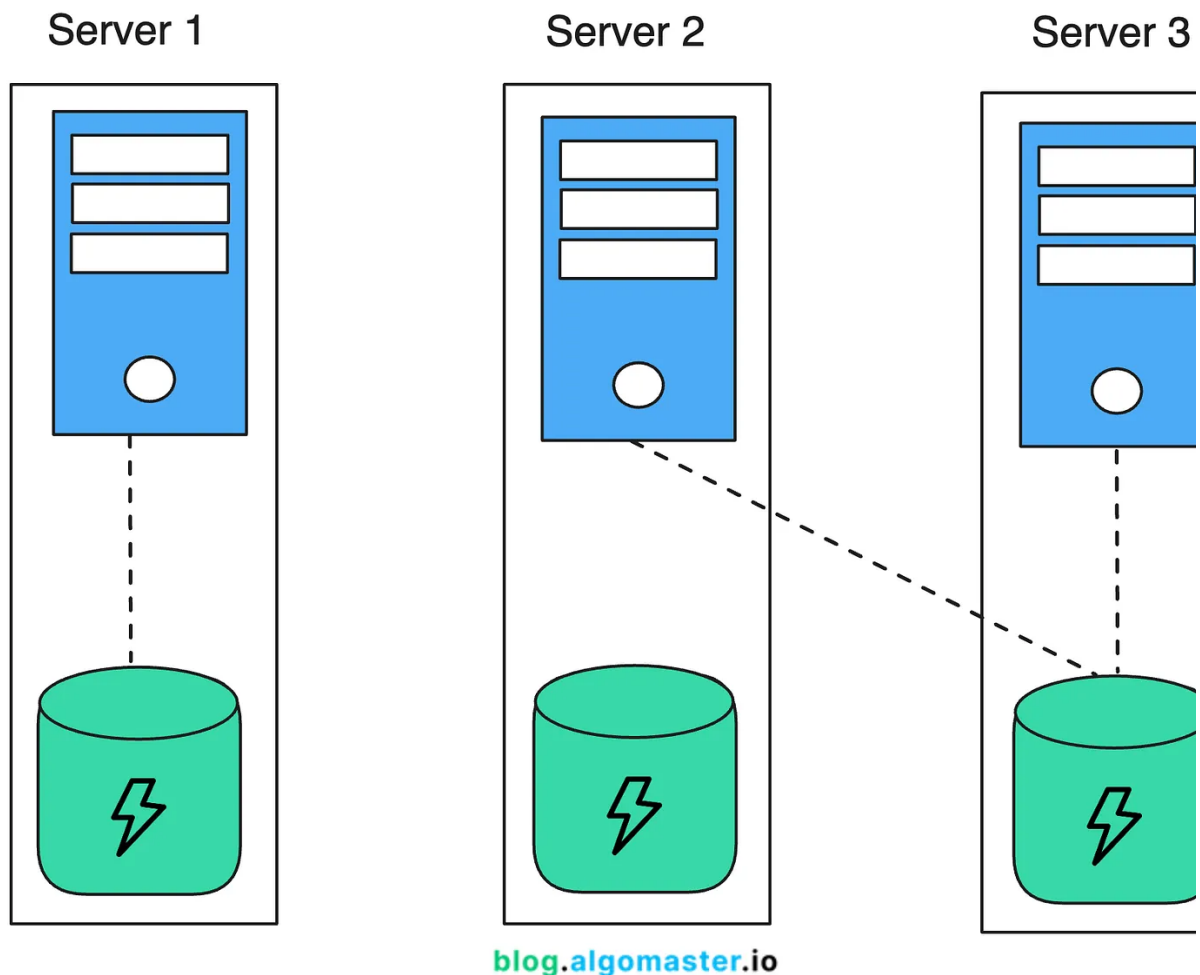
Visualized using **Multiplayer**

Advantages:

- **Low Latency**: Since both the cache and the application are on the same
  accessing the cache is very **quick**. This is ideal for applications where e
  millisecond counts, like real-time gaming or high-frequency trading pl

- **Cost Efficiency**: Co-locating the cache with the application can be mor
  effective, especially for small to medium-sized applications.

Disadvantages:

- **Resource Contention**: The cache and the application use the same serv
  resources like CPU memory, and I/O. This can slow things down under
  if both demand significant resources.

- **Limited Scalability:** Co-located caching is less scalable compared to de cache servers. Scaling the cache might require upgrading the entire ser

- **Complex Cache Invalidation**: When using co-located caches in a distri environment, keeping the cache up-to-date and getting rid of old data c tricky.

If you need your system to handle a lot of users, keep resources separate, an budget for it, using dedicated cache servers is likely the better option.

But if your app is smaller, you want to save money, or need it to be super fas the cache and app on the same server can work well.

---

5. How Does Distributed Caching Work?

1. **Data Distribution:** When data is cached, the client library typically **has** associated with the data to determine which cache node will store it.

2. **Data Replication:** For reliability, the cache system replicates the cache across multiple nodes. So, if one node, say A, stores the data, it might a copied to another node, like B, as a backup.

3. **Data Retrieval:** To get data from the cache, application provides the **ke** client library. The client library uses this key to **find and query** the nod has the data. If the data is present (a cache hit), it's returned to the appl not (a cache miss), the data is fetched from the primary data store (e.g., and it can be cached for future use.

4. **Cache Invalidation:** To keep the cache data in sync with the primary da it needs to be invalidated or updated periodically. Cache systems imple strategies like **time-based expiration** or **event-based expiration** for cac invalidation.

5. **Cache Eviction:** Since caches have limited space, they need an eviction

make room for new data. Common eviction policies include:

    a. **Least Recently Used (LRU)**: Evicts the data that hasn't been accesse[...] longest time.

    b. **Least Frequently Used (LFU)**: Evicts data that has been accessed th[...] frequently.

    c. **Time-to-Live (TTL)**: Evicts data that has been in the cache longer t[...] specified duration.

6. Challenges in Distributed Caching

- **Data Consistency**: Ensuring that all cache nodes have consistent data c[...] challenging, especially in a **write-heavy** application.

- **Cache Invalidation**: Deciding when to invalidate or update the cache c[...] complex, particularly when dealing with multiple cache nodes.

- **Network Partitioning**: In a distributed system, network partitions can [...] leading to situations where cache nodes are unable to communicate wi[...] other.

- **Scalability and Load Balancing**: As the system scales, ensuring that th[...] evenly balanced across nodes without any one node becoming a bottlen[...] requires sophisticated load balancing strategies.

7. Best Practices for Implementing Distribute[...] Caching

To get the most out of your distributed caching system, consider the follow[...] practices:

1. **Cache Judiciously**: Not all data benefits from caching. Focus on freque accessed, relatively static data.

2. **Set Appropriate TTLs**: Use Time-To-Live (TTL) values to automatically cached data and reduce staleness.

3. **Implement Cache-Aside Pattern**: Load data into the cache only when requested, to avoid unnecessary caching.

4. **Monitor and Tune**: Regularly monitor cache hit rates, memory usage, a network traffic to optimize performance.

5. **Plan for Failure**: Design your system to gracefully handle cache node f without significant impact on the application.

6. **Implement Cache Warming**: Develop strategies to pre-populate critic the cache to avoid cold starts.

---

8. Popular Distributed Caching Solutions

1. Redis

[Redis](#) is an open-source, in-memory data store that supports storing a wide data structures, including strings, hashes, lists, sets, sorted sets, bitmaps, h and geospatial indexes.

It supports data replication and persistence making it a popular choice for applications that require data durability and fault tolerance.

2. Memcached

[Memcached](#) is another open-source, in-memory caching system designed f and simplicity.

It is widely used for caching small chunks of data such as results of databas API calls, or page rendering.

Memcached is a pure in-memory cache with no persistence layer. This mak
for use cases where data does not need to be stored permanently, such as ca
results from database queries.

3. Amazon ElastiCache

[Amazon ElastiCache](#) is a fully managed in-memory caching service provide
It supports both Redis and Memcached, allowing developers to choose the
engine that best suits their needs.

ElastiCache supports multi-AZ (Availability Zone) deployments with autom
failover, ensuring that the cache remains available even in the event of an A

It can automatically scale the number of nodes in the cache cluster based or
allowing the cache to handle varying levels of traffic without manual interv

9. Conclusion

Distributed caching is an essential technique for building scalable, high-pe
applications.

By distributing data across multiple nodes, it allows for greater **scalability**,
tolerance, and **load balancing**.

However, it also introduces complexities, such as **data consistency**, **cache
invalidation**, and **network partitioning**, which must be carefully managed.

Make sure to carefully evaluate your needs, choose the right architecture, ar
best practices to ensure a successful implementation.

Thank you for reading!

If you found it valuable, hit a like ❤️ and consider subscribing for more suc
every week.

If you have any questions or suggestions, leave a comment.

This post is public so feel free to share it.

P.S. If you're enjoying this newsletter and want to get even more value, con:
becoming a **paid subscriber**.

As a paid subscriber, you'll unlock all **premium articles** and gain full access
premium courses on **algomaster.io**.

There are group discounts, gift options, and referral bonuses available.

Checkout my **Youtube channel** for more in-depth content.

Follow me on **LinkedIn**, **X** and **Medium** to stay updated.

Checkout my **GitHub repositories** for free interview preparation resources.

I hope you have a lovely day!

See you soon,
Ashish

← Previous

Discussion about this post

Comments Restacks

Write a comment...

Luv Singh 3 Sep 2024

💙 Liked by Ashish Pratap Singh

Loved it

♡ LIKE (3) 💬 REPLY

1 reply by Ashish Pratap Singh

Pratik Patel 3 Sep 2024

💙 Liked by Ashish Pratap Singh

What a clear explaination!! Nice and easy to understand.

♡ LIKE (2) 💬 REPLY

1 reply by Ashish Pratap Singh

5 more comments...