[Distributed Caching]

* Cache could be in some Machine or that of server
* Could be independent Machine

This scales well horizontally

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

But as the system gets **bigger**, the cache size also gets bigger and **a single-node cache** often falls short when scaling to handle millions of users and massive datasets.

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

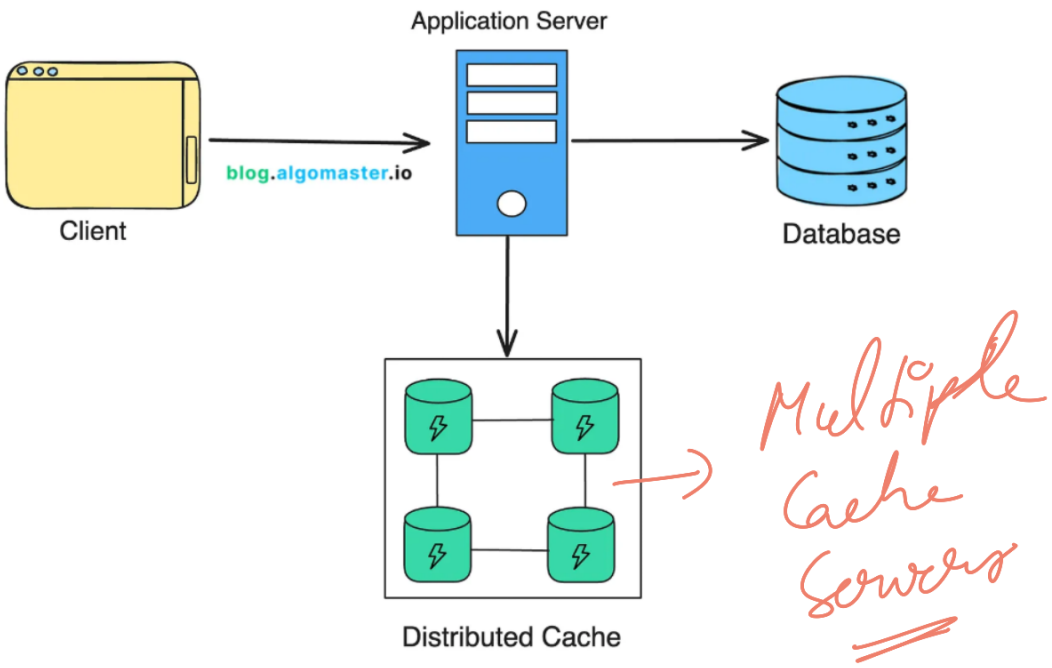
This is where **distributed caching** comes into play.

* If Cache is in same Machine
  → data retrieval is fast
  → hard to scale
* Cache is seperate Machine → easy to scale

What is Distributed Caching?

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

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



Visualized using Multiplayer

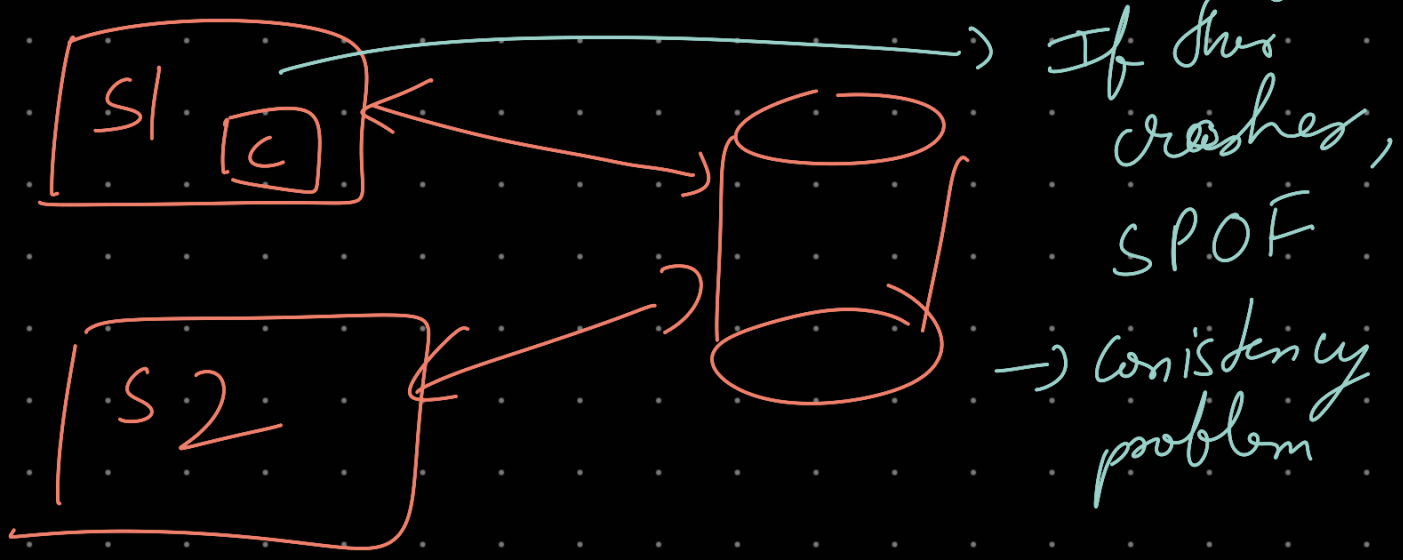
What are components of Distributed Caching

🧩 Components of Distributed Caching

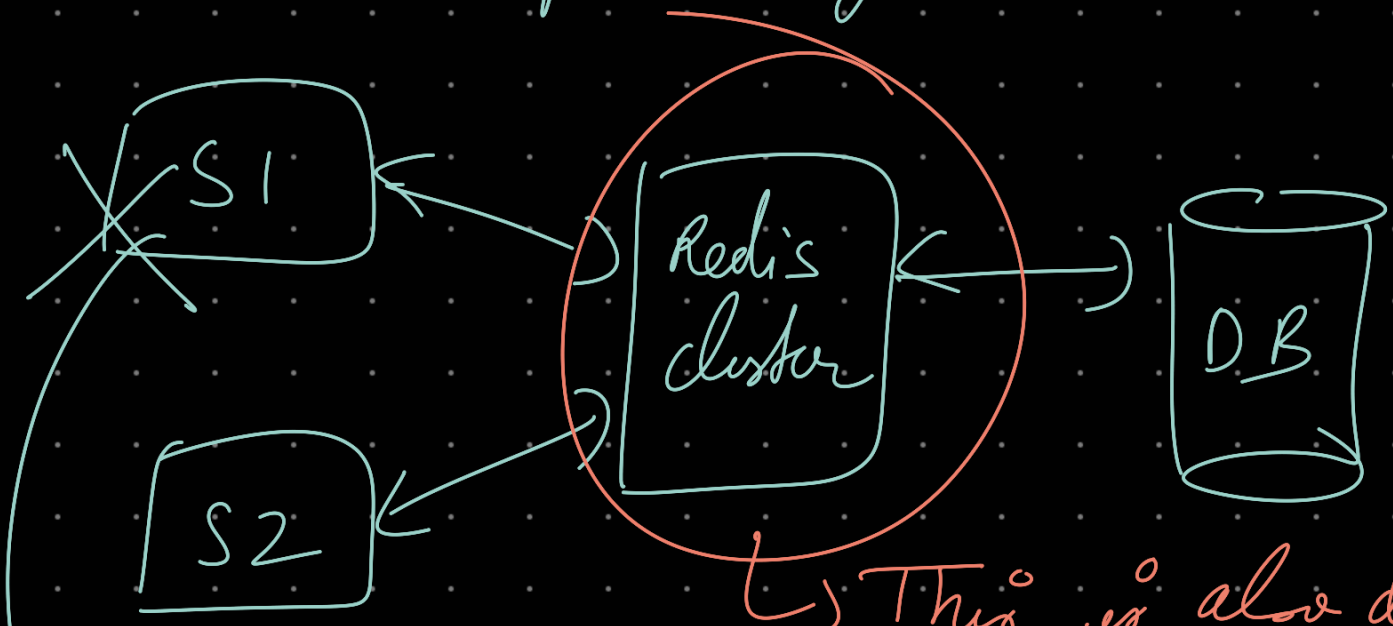
A distributed cache system typically consists of the following components:

1. **Cache Nodes**: These are the **individual servers** where the cache data is stored. Each node is a part of the overall cache cluster.
2. **Client Library/Cache Client**: Applications use a **client library** to talk to the distributed cache. This library handles the logic of connecting to cache nodes, distributing data, and retrieving cached data.
3. **Consistent Hashing**: This method spreads data **evenly** across cache nodes. It ensures that adding or removing nodes has minimal impact on the system.
4. **Replication**: To make the system more reliable, some distributed caches **replicate data** across multiple nodes. If one node goes down, the data is still available on another.
5. **Sharding**: Data is **split** into shards, and each shard is stored on a different cache node. It helps distribute the data **evenly** and allows the cache to scale horizontally.
6. **Eviction Policies**: Caches implement eviction policies like **LRU** (Least Recently Used), **LFU** (Least Frequently Used), or **TTL** (Time to Live) to get rid of old or less-used data and make space for new data.
7. **Coordination and Synchronization**: Coordination mechanisms like **distributed locks** or **consensus protocols** ensure that cache nodes remain synchronized, especially when multiple nodes try to change the same data at the same time.

Let's talk about Cache within same Machine as that of server



* Cache is a seperate server and scale independently



↳ This is also distributed

) If this fails, no impact on cache

* If our application data is small static, some server can be used for backend service and Cache

* For large scale system distributed dedicated servers are used for caching

🚧 Challenges in Distributed Caching

- **Data Consistency**: Ensuring that all cache nodes have consistent data can be challenging, especially in a **write-heavy** application.
- **Cache Invalidation**: Deciding when to invalidate or update the cache can be complex, particularly when dealing with multiple cache nodes.
- **Network Partitioning**: In a distributed system, network partitions can occur, leading to situations where cache nodes are unable to communicate with each other.
- **Scalability and Load Balancing**: As the system scales, ensuring that the cache is evenly balanced across nodes without any one node becoming a bottleneck requires sophisticated load balancing strategies.

✅ Best Practices for Implementing Distributed Caching

To get the most out of your distributed caching system, consider the following best practices:

1. **Cache Judiciously**: Not all data benefits from caching. Focus on frequently accessed, relatively static data.
2. **Set Appropriate TTLs**: Use Time-To-Live (TTL) values to automatically expire cached data and reduce staleness.
3. **Implement Cache-Aside Pattern**: Load data into the cache only when it's requested, to avoid unnecessary caching.
4. **Monitor and Tune**: Regularly monitor cache hit rates, memory usage, and network traffic to optimize performance.
5. **Plan for Failure**: Design your system to gracefully handle cache node failures without significant impact on the application.
6. **Implement Cache Warming**: Develop strategies to pre-populate critical data in the cache to avoid cold starts.

E.g -) Redis, Memcached, Amazon ElastiCache