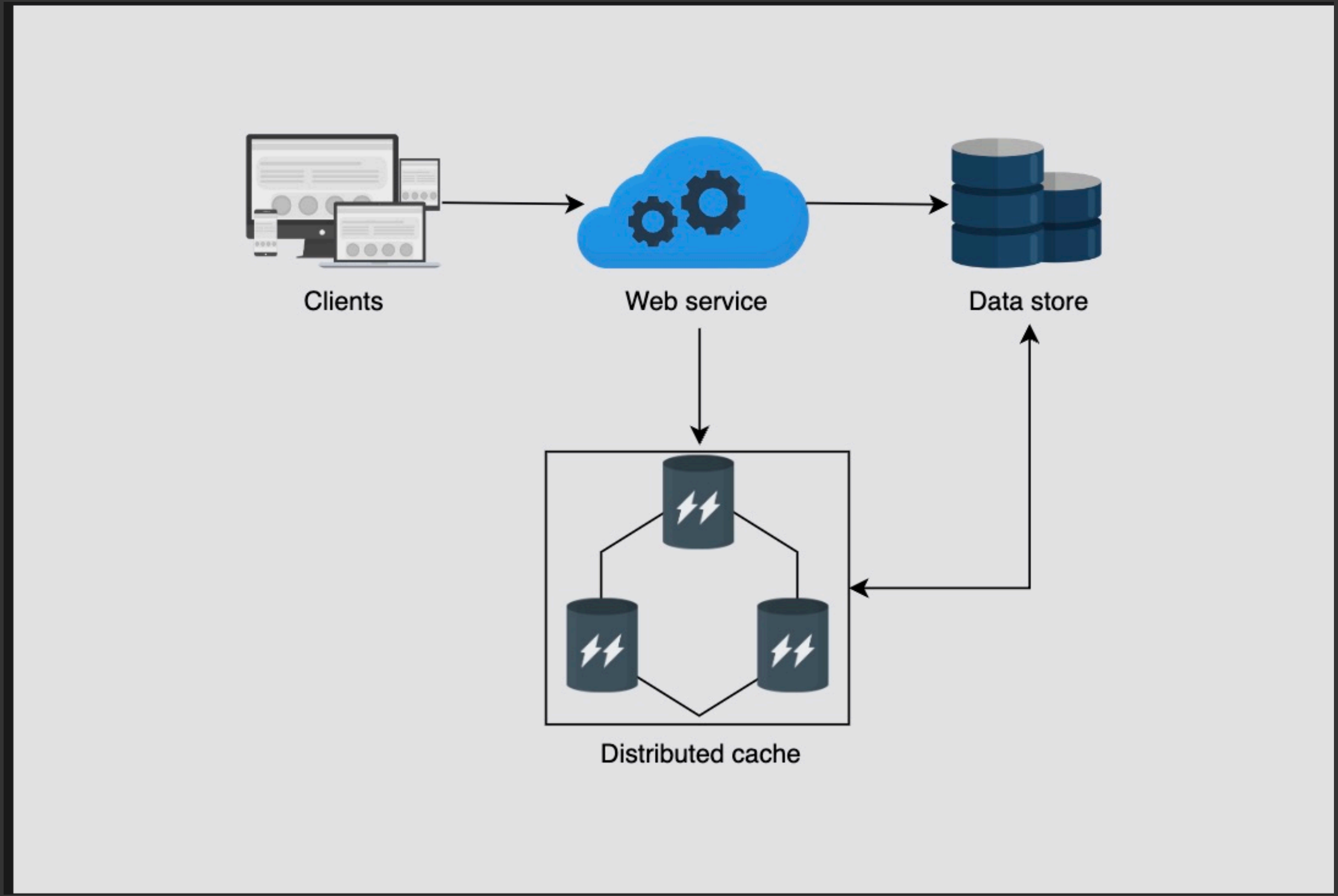
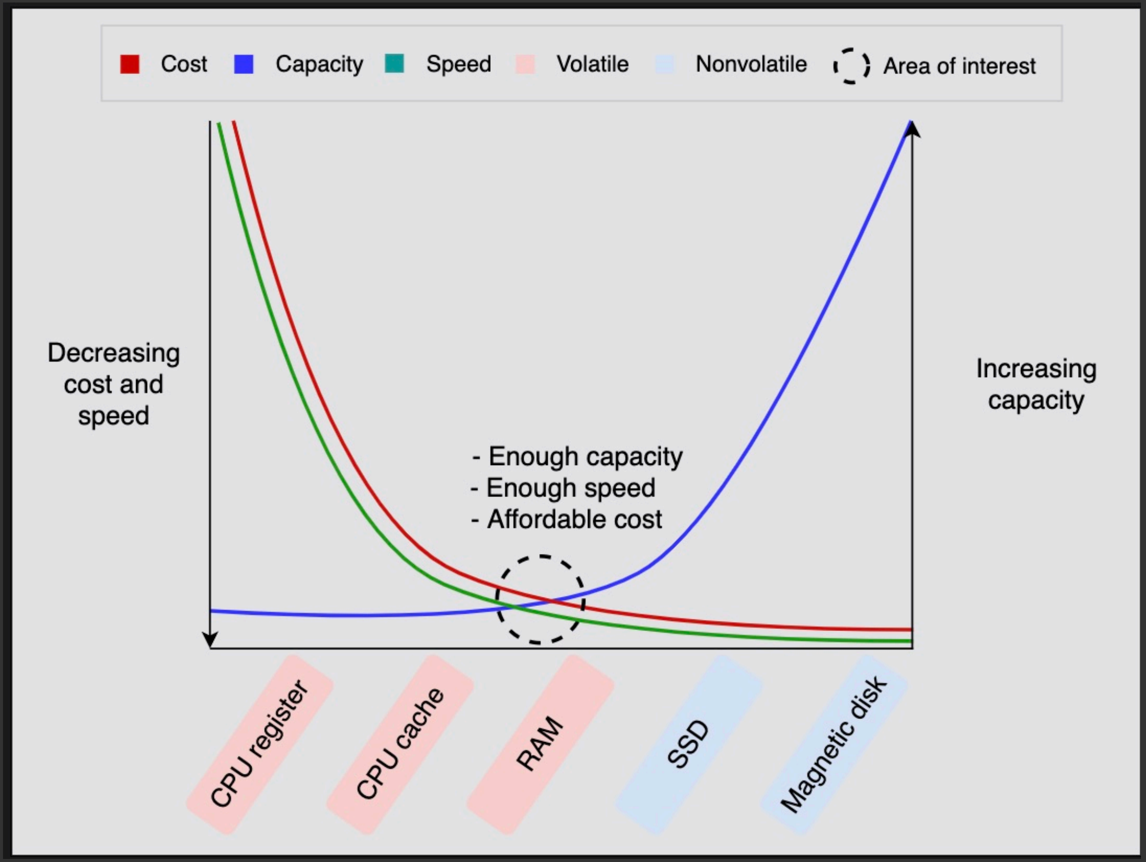
Distributed Cache

The number of users increases, the database queries also increase. And as a result, the service providers are overburdened, resulting in slow performance.

In such cases, a cache is added to the system to deal with performance deterioration. A cache is a temporary data storage that can serve data faster by keeping data entries in memory. Caches store only the most frequently accessed data. When a request reaches the serving host, it retrieves data from the cache (cache hit) and serves the user. However, if the data is unavailable in the cache (cache miss), the data will be queried from the database. Also, the cache is populated with the new value to avoid cache misses for the next time.





what is distributed Cache ?

A distributed cache is a caching system where multiple cache servers coordinate to store frequently accessed data. Distributed caches are needed in environments where a single cache server isn't enough to store all the data. At the same time, it's scalable and guarantees a higher degree of availability.

They minimize user-perceived latency by precalculating results and storing frequently accessed data.
They pre-generate expensive queries from the database.
They store user session data temporarily.
They serve data from temporary storage even if the data store is down temporarily.
Finally, they reduce network costs by serving data from local resources.

why distributed cache ?

When the size of data required in the cache increases, storing the entire data in one system is impractical. This is because of the following three reasons:

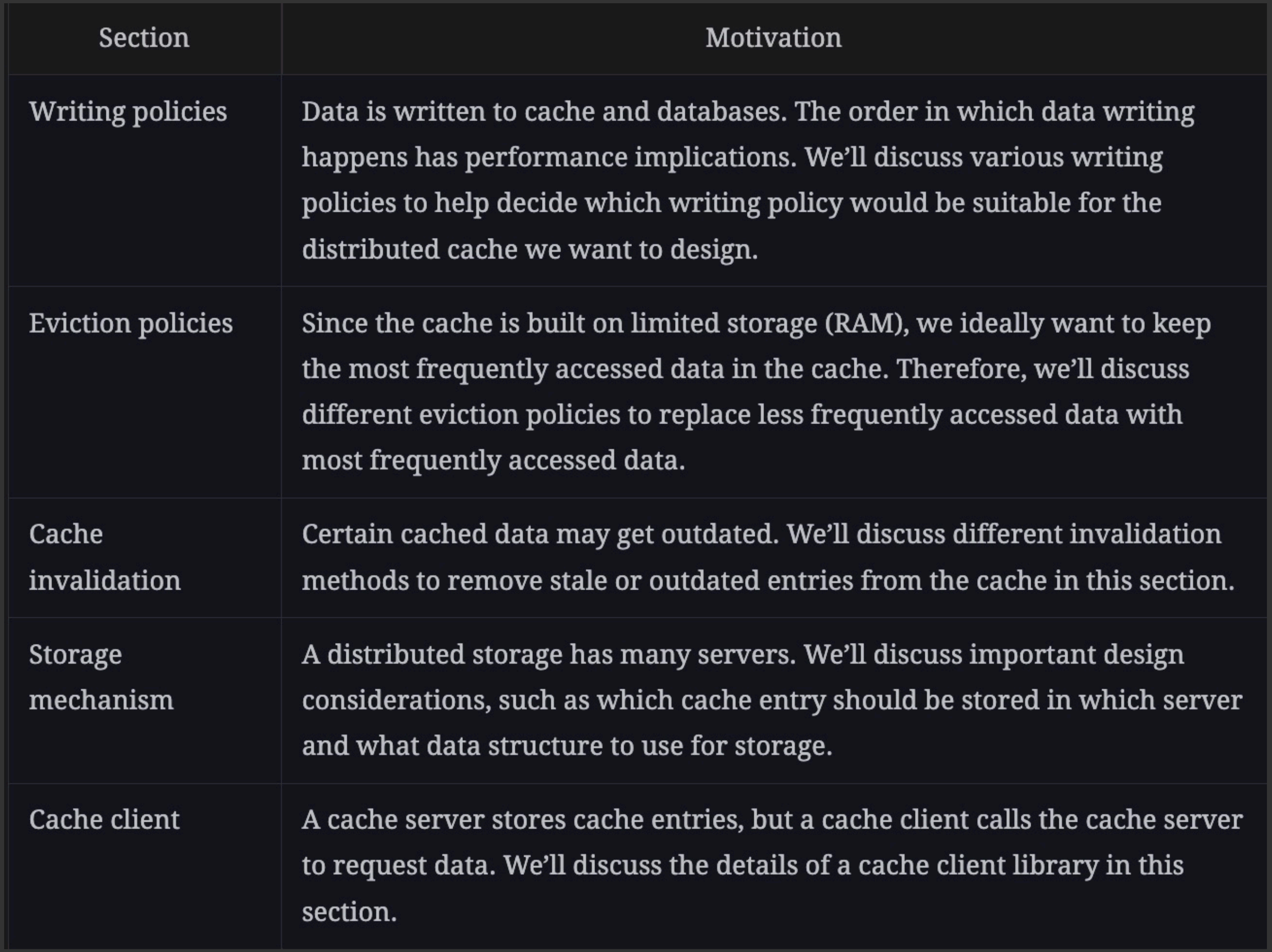
It can be a potential single point of failure (SPOF).

A system is designed in layers, and each layer should have its caching mechanism to ensure the decoupling of sensitive data from different layers.

Caching at different locations helps reduce the serving latency at that layer.

System Layer	Technology in Use	Usage
Web	HTTP cache headers, web accelerators, key-value store, CDNs, and so on	Accelerate retrieval of static web content, and manage sessions
Application	Local cache and key-value data store	Accelerate application-level computations and data retrieval
Database	Database cache, buffers, and key-value data store	Reduce data retrieval latency and I/O load from database

Background of Distributed Cache



Section	Motivation
Writing policies	Data is written to cache and databases. The order in which data writing happens has performance implications. We'll discuss various writing policies to help decide which writing policy would be suitable for the distributed cache we want to design.
Eviction policies	Since the cache is built on limited storage (RAM), we ideally want to keep the most frequently accessed data in the cache. Therefore, we'll discuss different eviction policies to replace less frequently accessed data with most frequently accessed data.
Cache invalidation	Certain cached data may get outdated. We'll discuss different invalidation methods to remove stale or outdated entries from the cache in this section.
Storage mechanism	A distributed storage has many servers. We'll discuss important design considerations, such as which cache entry should be stored in which server and what data structure to use for storage.
Cache client	A cache server stores cache entries, but a cache client calls the cache server to request data. We'll discuss the details of a cache client library in this section.

1. Writing Policies

cache stores a copy (or part) of data, which is persistently stored in a data store. When we store data to the data store, some important questions arise

Write-through cache: The write-through mechanism writes on the cache as well as on the database. Writing on both storages can happen concurrently or one after the other. This increases the write latency but ensures strong consistency between the database and the cache.

Write-back cache: In the write-back cache mechanism, the data is first written to the cache and asynchronously written to the database. Although the cache has updated data, inconsistency is inevitable in scenarios where a client reads stale data from the database. However, systems using this strategy will have small writing latency.

Write-around cache: This strategy involves writing data to the database only. Later, when a read is triggered for the data, it's written to cache after a cache miss. The database will have updated data, but such a strategy isn't favorable for reading recently updated data.

2. Eviction Policy

Least recently used (LRU)
Most recently used (MRU)
Least frequently used (LFU)
Most frequently used (MFU)

3. Cache Invalidation

Apart from the eviction of less frequently accessed data, some data residing in the cache may become stale or outdated over time. Such cache entries are invalid and must be marked for deletion.

The situation demands a question: How do we identify stale entries?

Resolution of the problem requires storing metadata corresponding to each cache entry. Specifically, maintaining a time-to-live (TTL) value to deal with outdated cache items.

We can use two different approaches to deal with outdated items using TTL:

Active expiration: This method actively checks the TTL of cache entries through a daemon process or thread.
Passive expiration: This method checks the TTL of a cache entry at the time of access.
Each expired item is removed from the cache upon discovery.

4. Storage

1st

Hash function
It’s possible to use hashing in two different scenarios:

Identify the cache server in a distributed cache to store and retrieve data.
Locate cache entries inside each cache server.
For the first scenario, we can use different hashing algorithms. However, consistent hashing or its flavors usually perform well in distributed systems because simple hashing won’t be ideal in case of crashes or scaling.

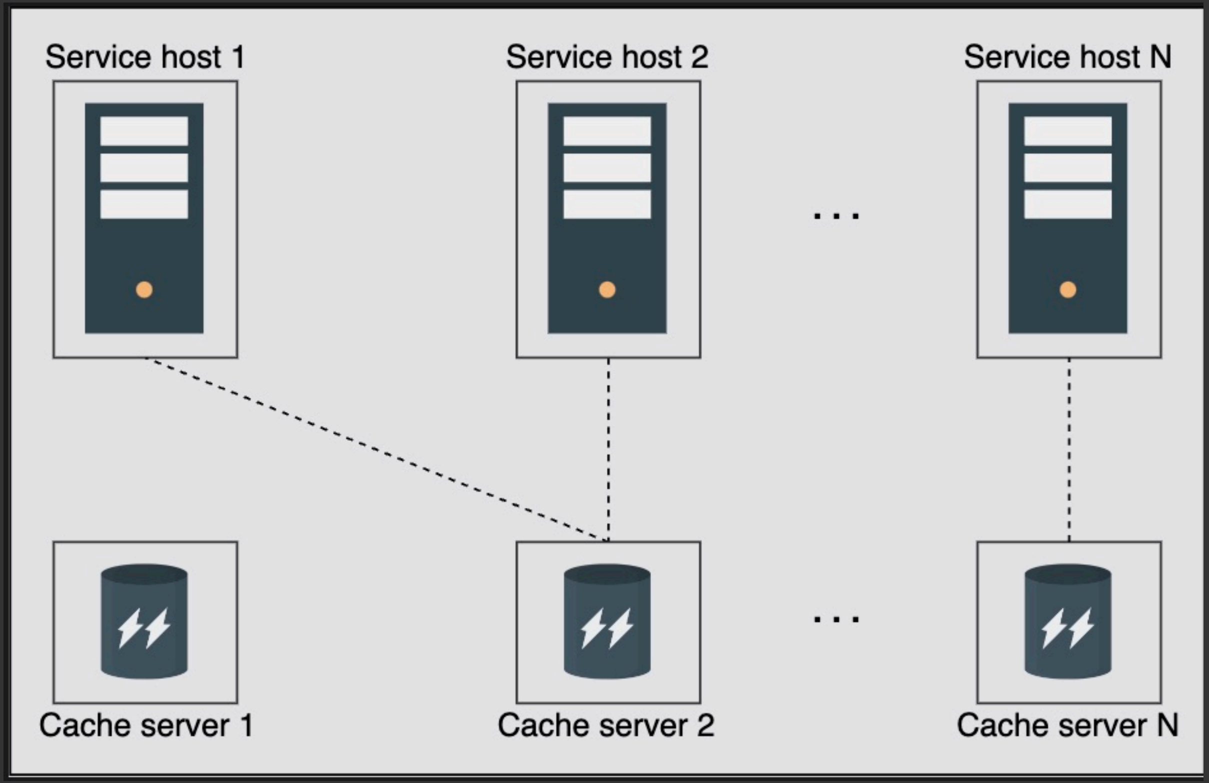
2nd

We’ll use a doubly linked list. The main reason is its widespread usage and simplicity. Furthermore, adding and removing data from the doubly linked list in our case will be a constant time operation. This is because we either evict a specific entry from the tail of the linked list or relocate an entry to the head of the doubly linked list. Therefore, no iterations are required.

5. Sharding of Cache Clusters

To avoid SPOF & high load on single instance, we introduced sharding.

Sharding involves splitting up cache data among multiple cache servers.



Co-located cache
The co-located cache embeds cache and service functionality within the same host.

The main advantage of this strategy is the reduction in CAPEX and OPEX of extra hardware. Furthermore, with the scaling of one service, automatic scaling of the other service is obtained. However, the failure of one machine will result in the loss of both services simultaneously.

Cache client#
We discussed that the hash functions should be used for the selection of cache servers. But what entity performs these hash calculations?

A cache client is a piece of code residing in hosting servers that do (hash) computations to store and retrieve data in the cache servers. Also, cache clients may coordinate with other system components like monitoring and configuration services. All cache clients are programmed in the same way so that the same PUT, and GET operations from different clients return the same results. Some of the characteristics of cache clients are the following:

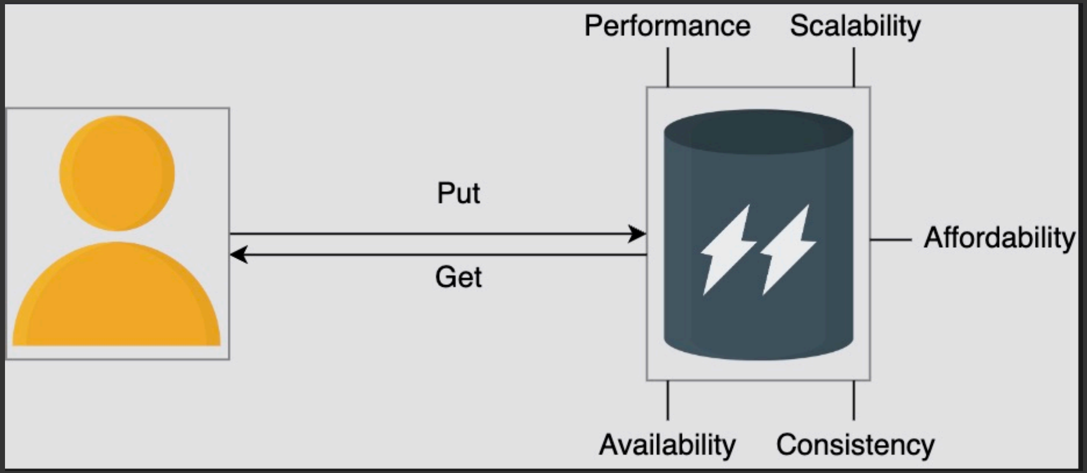
Each cache client will know about all the cache servers.
All clients can use well-known transport protocols like TCP or UDP to talk to the cache servers.

High-level design of Distributed Cache

Requirements

Functional Requirements

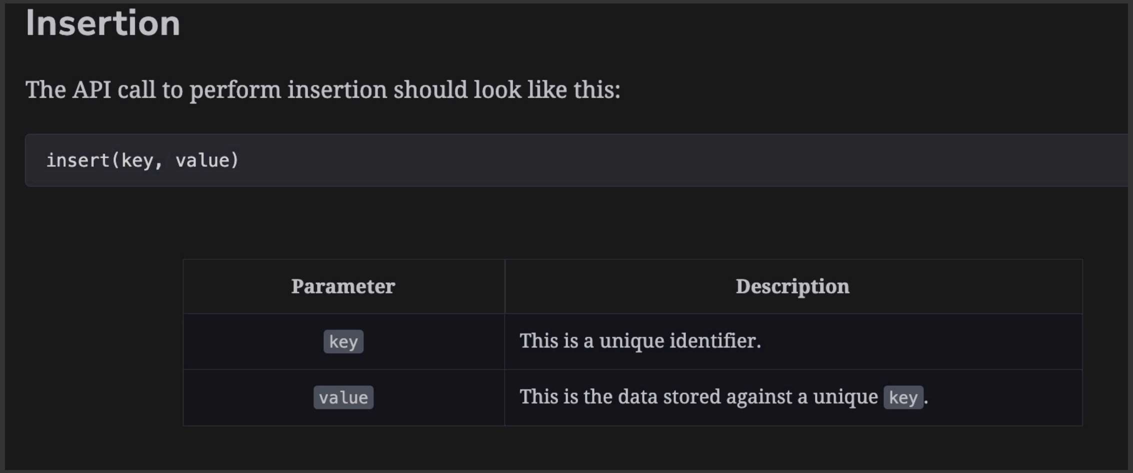
- Insert Data : The user of a distributed cache system must be able to insert entry of the cache.
- Retrieve Data : The user should be able to retrieve data corresponding to a specific key.



Non - Functional Requirement

→ High Performance
→ Scalability
→ High availability
→ Consistency
→ Affordability

API Design

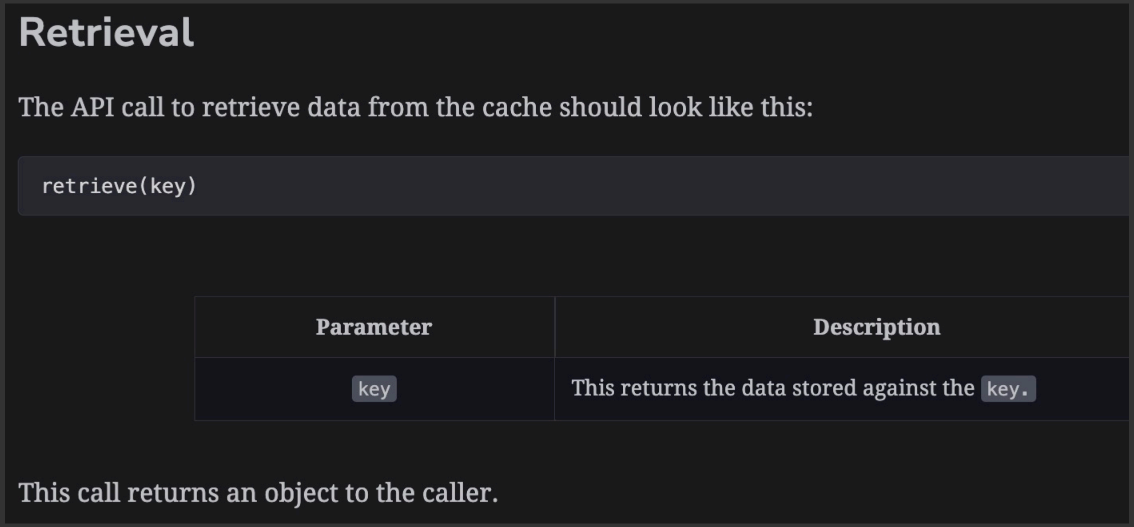


Insertion

The API call to perform insertion should look like this:

```
insert(key, value)
```

Parameter	Description
`key`	This is a unique identifier.
`value`	This is the data stored against a unique `key`.



Retrieval

The API call to retrieve data from the cache should look like this:

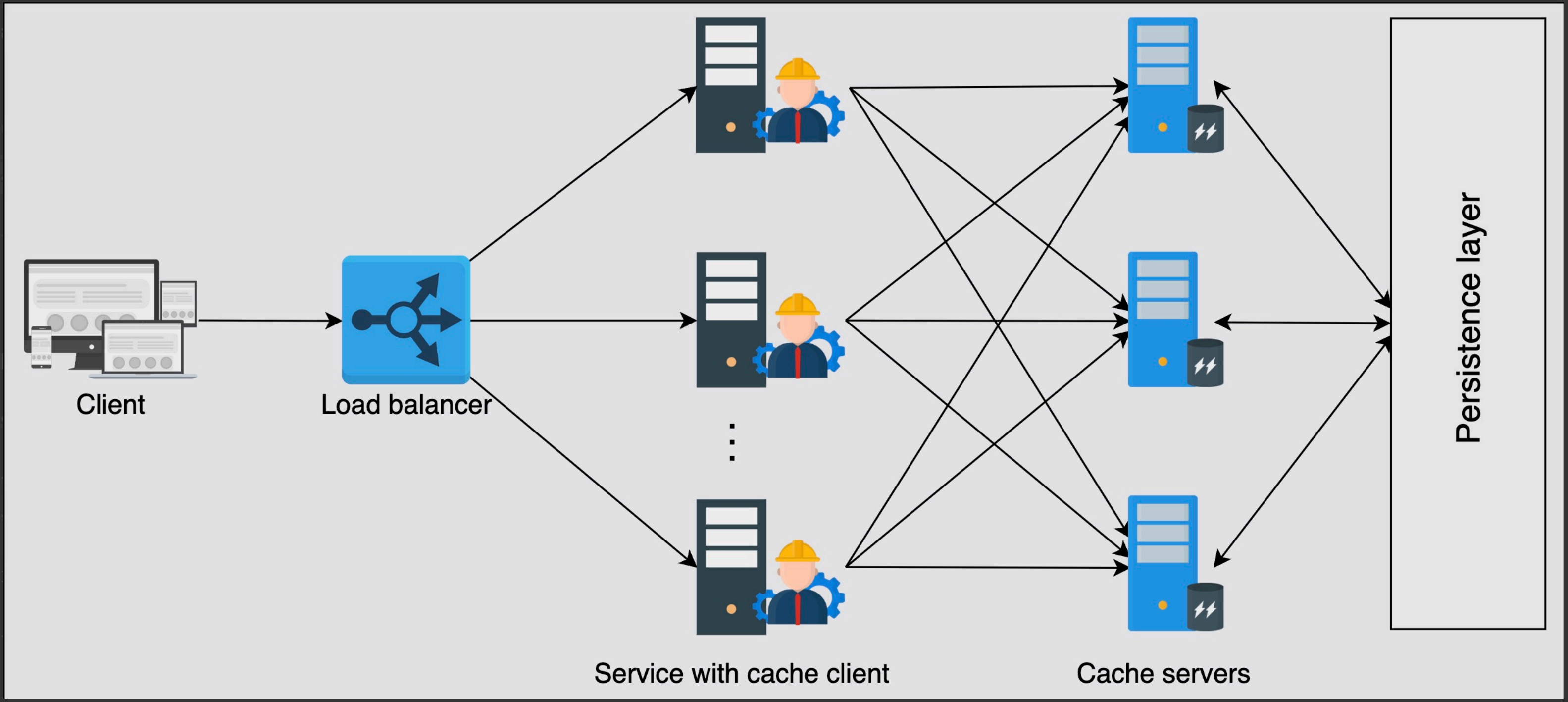
```
retrieve(key)
```

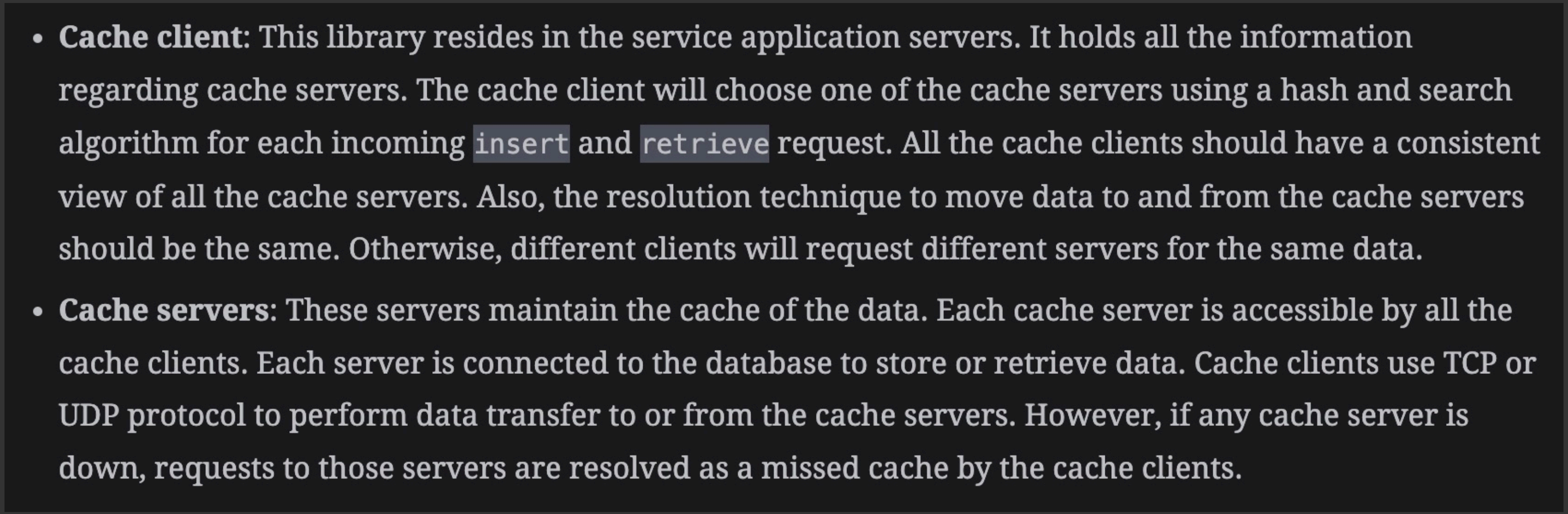
Parameter	Description
`key`	This returns the data stored against the `key.`

This call returns an object to the caller.

Design Considerations

1. Storage hardware
2. Data structure → Hash Table
3. Cache Client
4. writing policy →
5. Eviction Policy





- **Cache client**: This library resides in the service application servers. It holds all the information regarding cache servers. The cache client will choose one of the cache servers using a hash and search algorithm for each incoming `insert` and `retrieve` request. All the cache clients should have a consistent view of all the cache servers. Also, the resolution technique to move data to and from the cache servers should be the same. Otherwise, different clients will request different servers for the same data.
- **Cache servers**: These servers maintain the cache of the data. Each cache server is accessible by all the cache clients. Each server is connected to the database to store or retrieve data. Cache clients use TCP or UDP protocol to perform data transfer to or from the cache servers. However, if any cache server is down, requests to those servers are resolved as a missed cache by the cache clients.

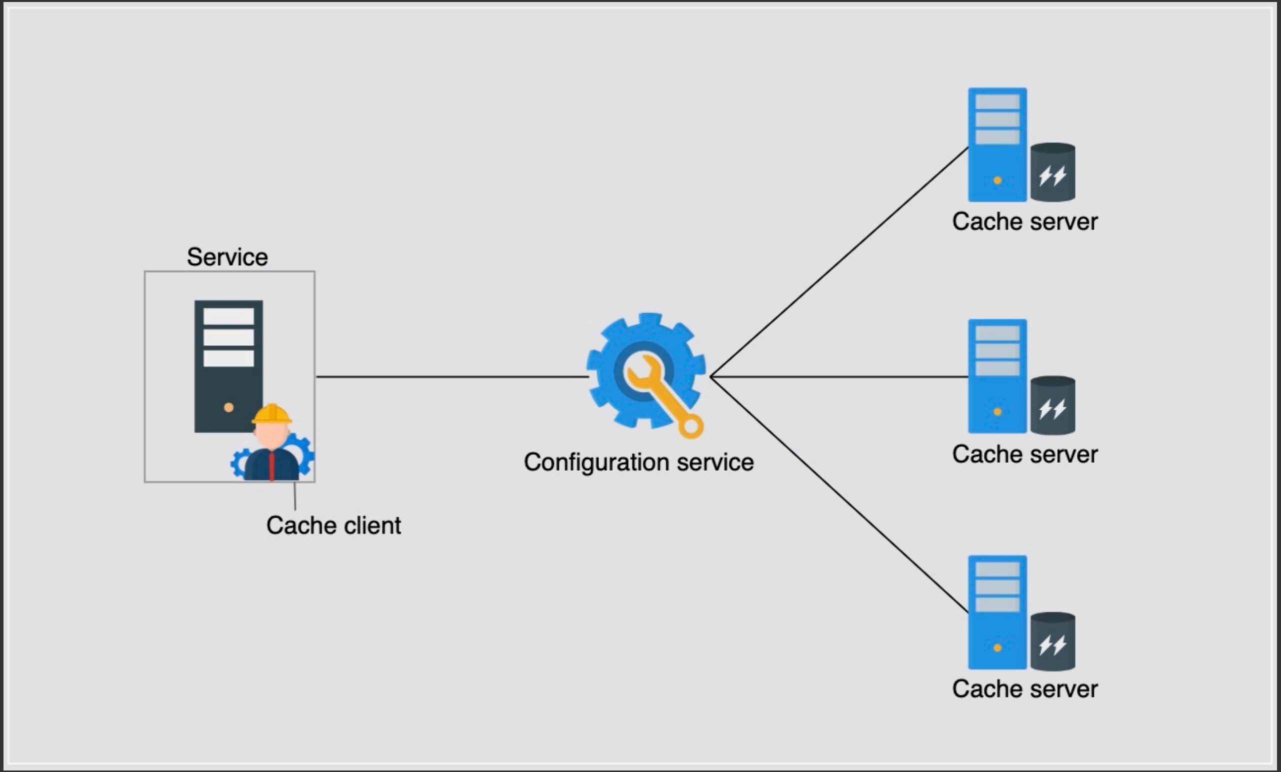
Detailed design of Distributed Cache

Find and remove limitations
Before we get to the detailed design, we need to understand and overcome some challenges:

There's no way for the cache client to realize the addition or failure of a cache server.

The solution will suffer from the problem of single point of failure (SPOF) because we have a single cache server for each set of cache data. Not only that, if some of the data on any of the cache servers is frequently accessed (generally referred to as a hotkey problem), then our performance will also be slow.

Our solution also didn't highlight the internals of cache servers. That is, what kind of data structures will it use to store and what eviction policy will it use?



Solution 1: It’s possible to have a configuration file in each of the service hosts where the cache clients reside. The configuration file will contain the updated health and metadata required for the cache clients to utilize the cache servers efficiently. Each copy of the configuration file can be updated through a push service by any DevOps tool. The main problem with this strategy is that the configuration file will have to be manually updated and deployed through some DevOps tools.

Solution 2: We can store the configuration file in a centralized location that the cache clients can use to get updated information about cache servers. This solves the deployment issue, but we still need to manually update the configuration file and monitor the health of each server.

Solution 3: An automatic way of handling the issue is to use a configuration service that continuously monitors the health of the cache servers. In addition to that, the cache clients will get notified when a new cache server is added to the cluster. When we use this strategy, no human intervention or monitoring will be required in case of failures or the addition of new nodes. Finally, the cache clients obtain the list of cache servers from the configuration service.

Improve availability

The second problem relates to cache unavailability if the cache servers fail. A simple solution is the addition of replica nodes. We can start by adding one primary and two backup nodes in a cache shard. With replicas, there's always a possibility of inconsistency. If our replicas are in close proximity, writing over replicas is performed synchronously to avoid inconsistencies between shard replicas. It's crucial to divide cache data among shards so that neither the problem of unavailability arises nor any hardware is wasted.

This solution has two main advantages:

There's improved availability in case of failures.

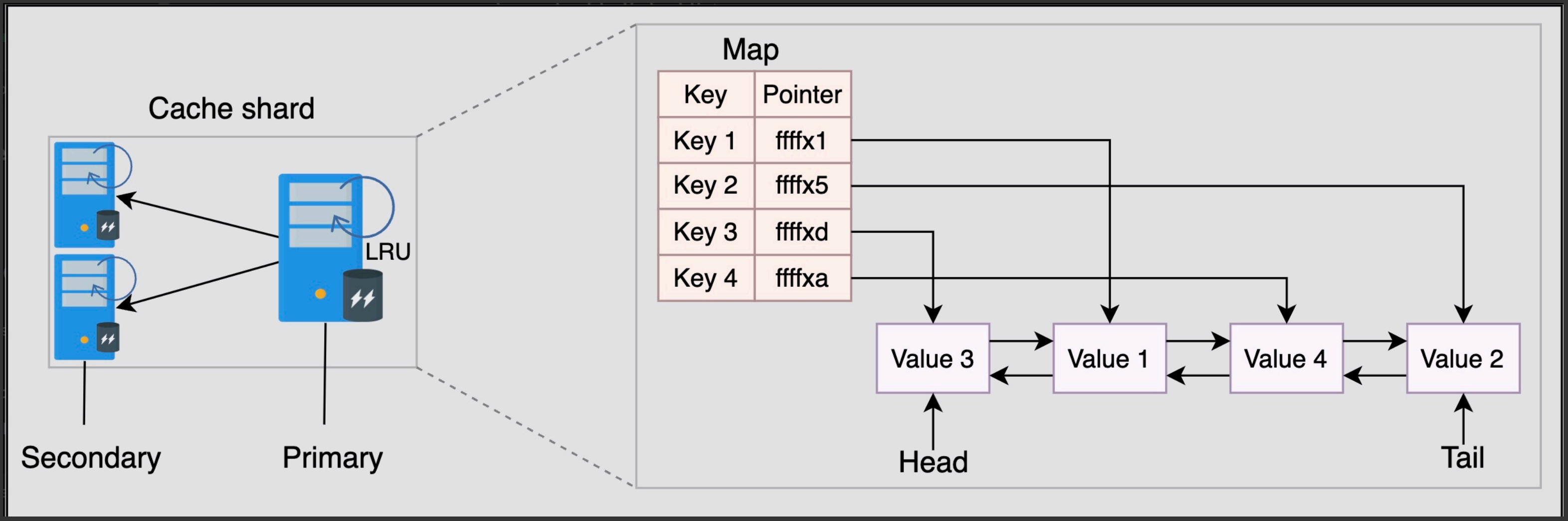
Hot shards can have multiple nodes (primary-secondary) for reads.

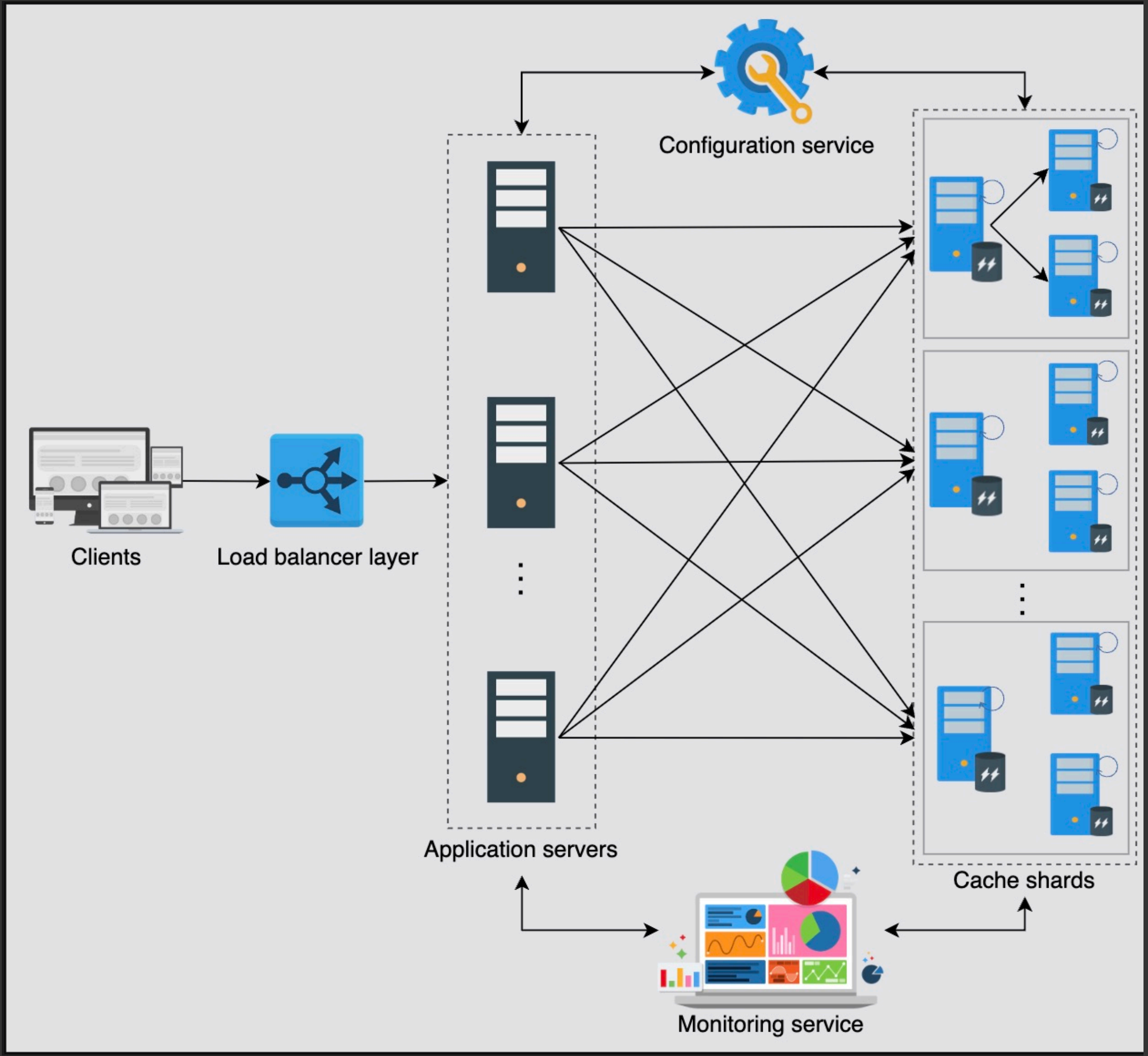
Each cache client should use three mechanisms to store and evict entries from the cache servers:

Hash map: The cache server uses a hash map to store or locate different entries inside the RAM of cache servers. The illustration below shows that the map contains pointers to each cache value.

Doubly linked list: If we have to evict data from the cache, we require a linked list so that we can order entries according to their frequency of access. The illustration below depicts how entries are connected using a doubly linked list.

Eviction policy: The eviction policy depends on the application requirements. Here, we assume the least recently used (LRU) eviction policy.





While consistent hashing is a good choice, it may result in unequal distribution of data, and certain servers may get overloaded. How do we resolve this problem?

A number of consistent hashing algorithms' flavors have been suggested over time. We can use one such flavor (used here) that distributes load uniformly and even makes multiple copies of the same data on different cache servers. Each cache server can have virtual servers inside it, and the number of virtual servers in a machine depends on the machine's capability. This results in a finer control on the amount of load on a cache server. At the same time, it improves availability.

Evaluation of Distributed Cache Design

High performance
Here are some design choices we made that will contribute to overall good performance:

We used consistent hashing. Finding a key under this algorithm requires a time complexity of O(log(N)), where N represents the number of cache shards.

Inside a cache server, keys are located using hash tables that require constant time on average.

The LRU eviction approach uses a constant time to access and update cache entries in a doubly linked list.

The communication between cache clients and servers is done through TCP and UDP protocols, which is also very fast.

Since we added more replicas, these can reduce the performance penalties that we have to face if there’s a high request load on a single machine.

An important feature of the design is adding, retrieving, and serving data from the RAM. Therefore, the latency to perform these operations is quite low.

Scalability
We can create shards based on requirements and changing server loads. While we add new cache servers to the cluster, we also have to do a limited number of rehash computations, thanks to consistent hashing.

Adding replicas reduces the load on hot shards. Another way to handle the hotkeys problem is to do further sharding within the range of those keys. Although the scenario where a single key will become hot is rare, it’s possible for the cache client to devise solutions to avoid the single hotkey contention issue. For example, cache clients can intelligently avoid such a situation that a single key becomes a bottleneck, or we can use dynamic replication for specific keys, and so on. Nonetheless, the solutions are complex and beyond the scope of this lesson.

High availability
We have improved the availability through redundant cache servers. Redundancy adds a layer of reliability and fault tolerance to our design. We also used the leader-follower algorithm to conveniently manage a cluster shard. However, we haven’t achieved high availability because we have two shard replicas, and at the moment, we assume that the replicas are within a data center.

Consistency
It's possible to write data to cache servers in a synchronous or asynchronous mode. In the case of caching, the asynchronous mode is favored for improved performance. Consequently, our caching system suffers from inconsistencies. Alternatively, strong consistency comes from synchronous writing, but this increases the overall latency, and the performance takes a hit.

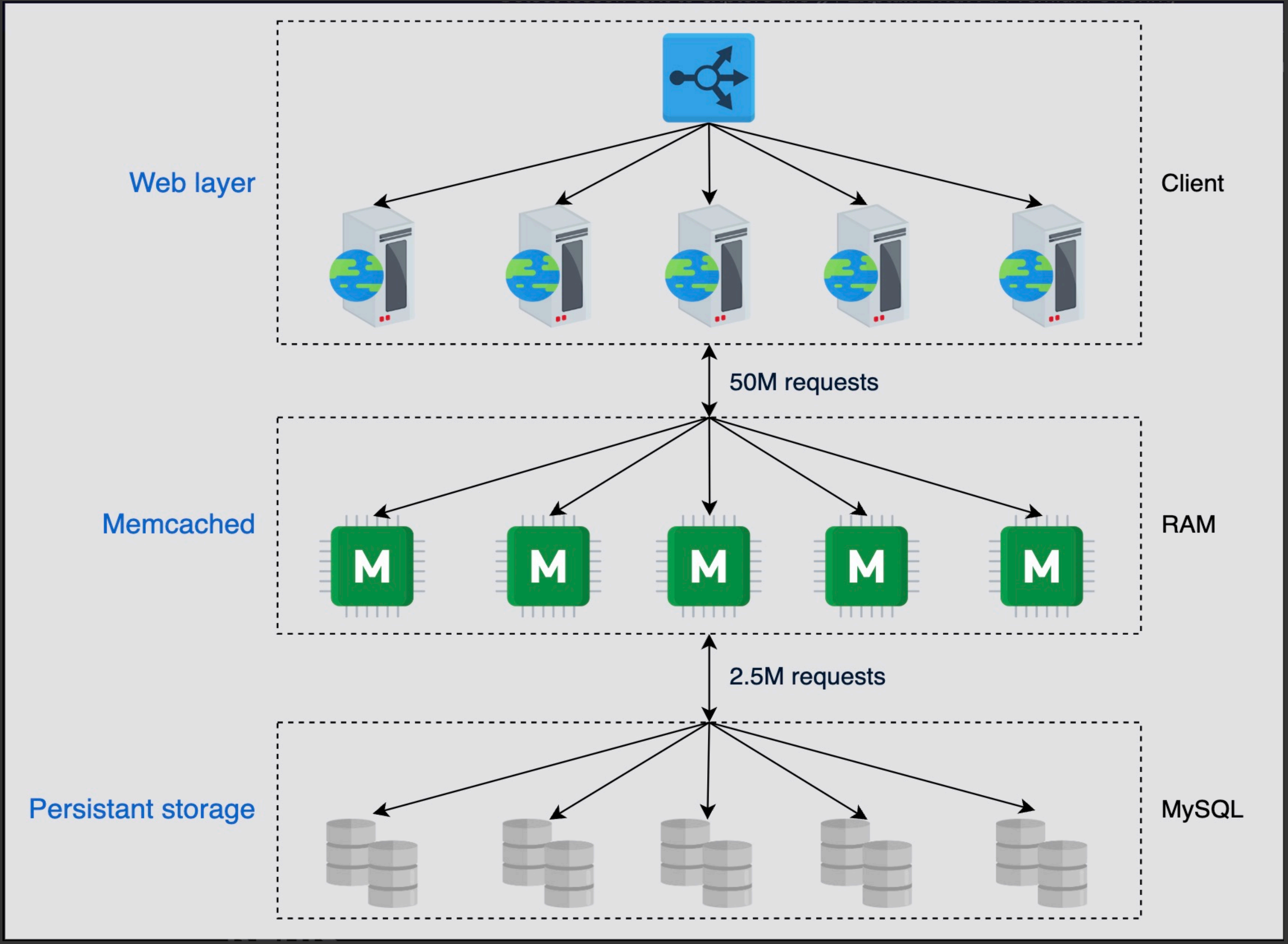
Mem Cache Vs Redis

Mem cache

Memcached was introduced in 2003. It's a key-value store distributed cache designed to store objects very fast. Memcached stores data in the form of a key-value pair. Both the key and the value are strings. This means that any data that has been stored will have to be serialized.

Memcached is able to achieve almost a deterministic query speed (O(1))serving millions of keys per second using a high-end system

```
get <key_1> <key_2> <key_3> ...
set <key> <value> ...
delete <key>[<time>] ...
```

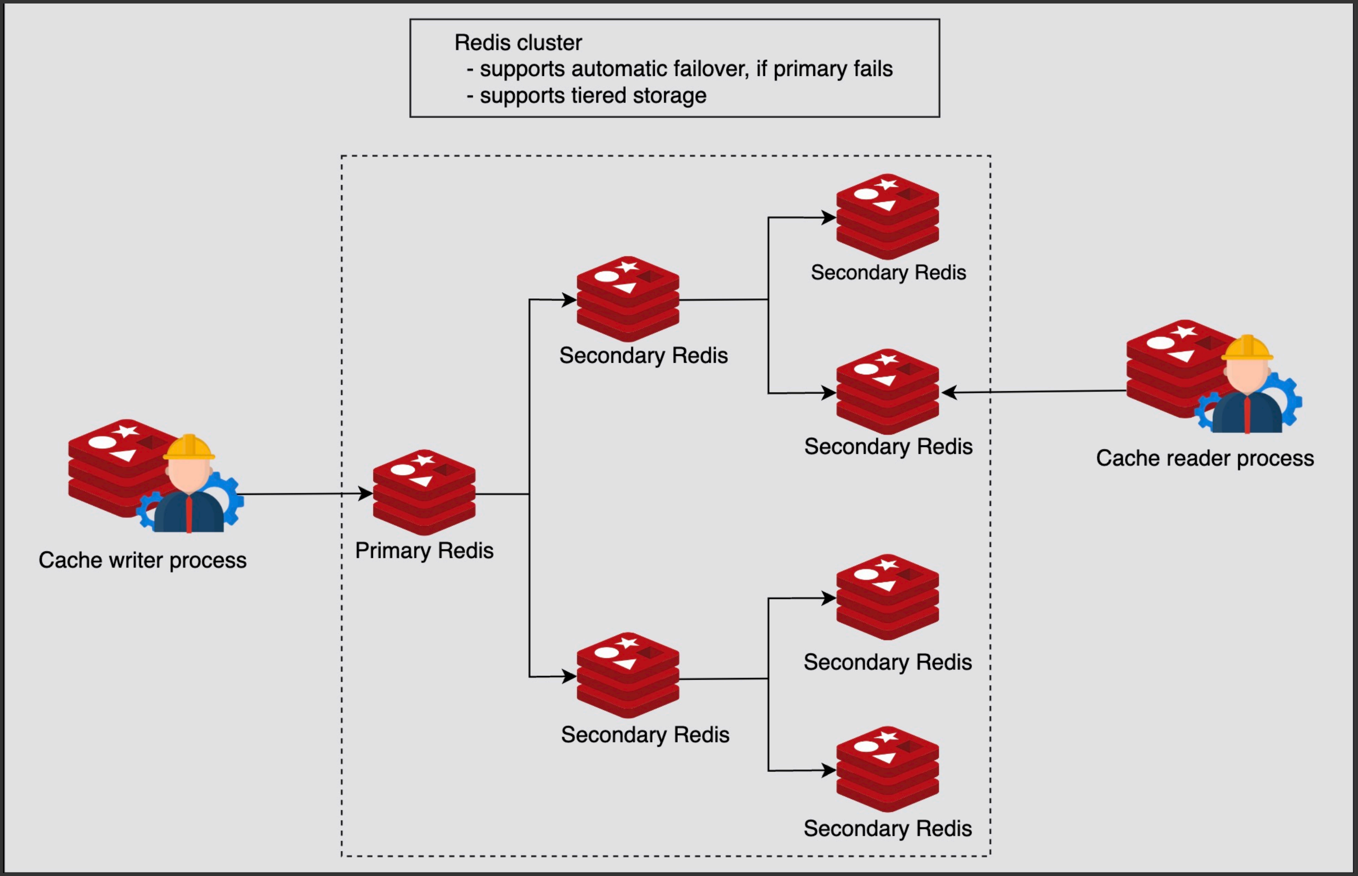

Web layer
Client
50M requests
Memcached
RAM
2.5M requests
Persistant storage
MySQL

Redis

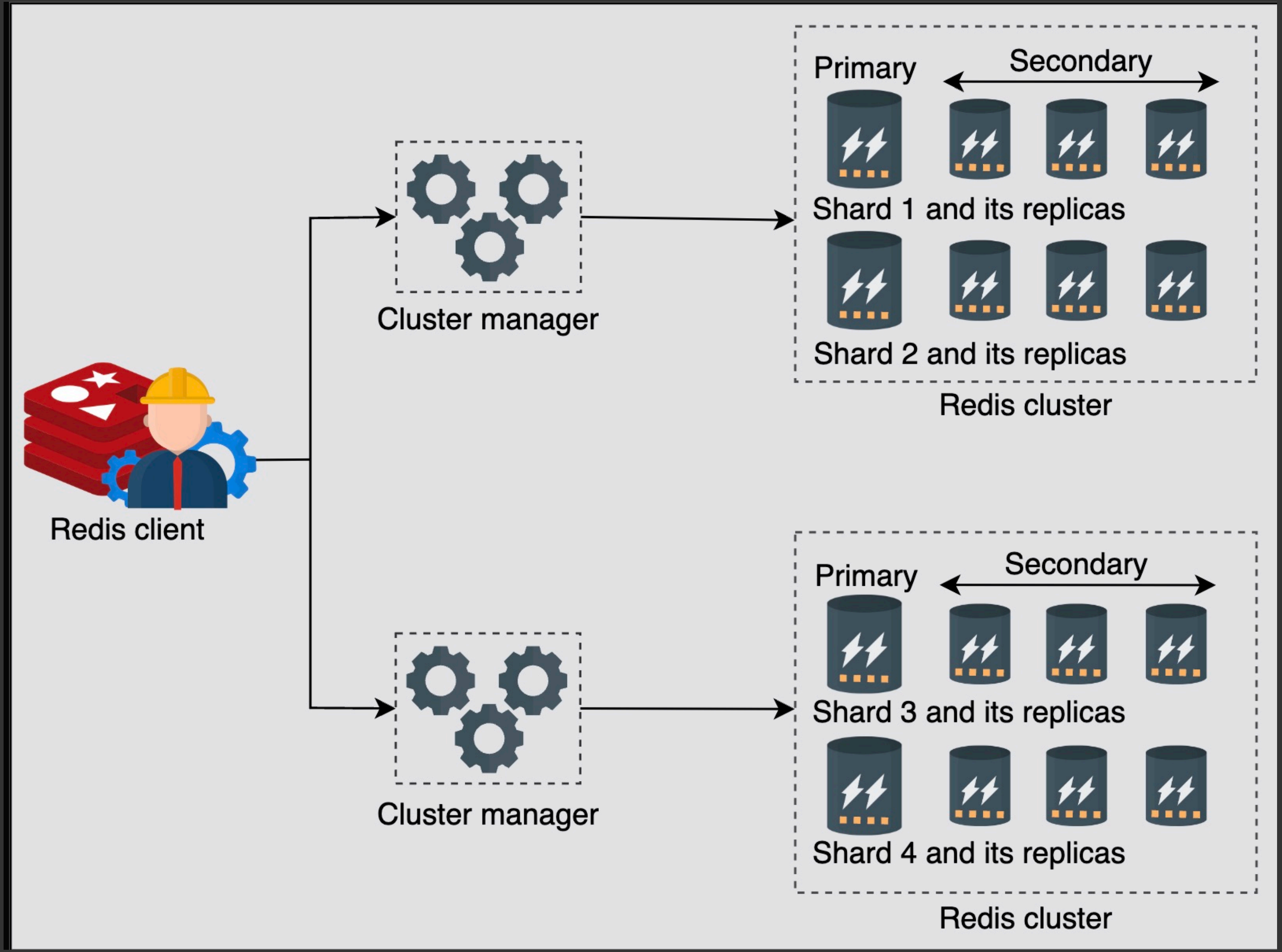
Data structure store: Redis understands the different data structures it stores. We don't have to retrieve data structures from it, manipulate them, and then store them back. We can make in-house changes that save both time and effort.

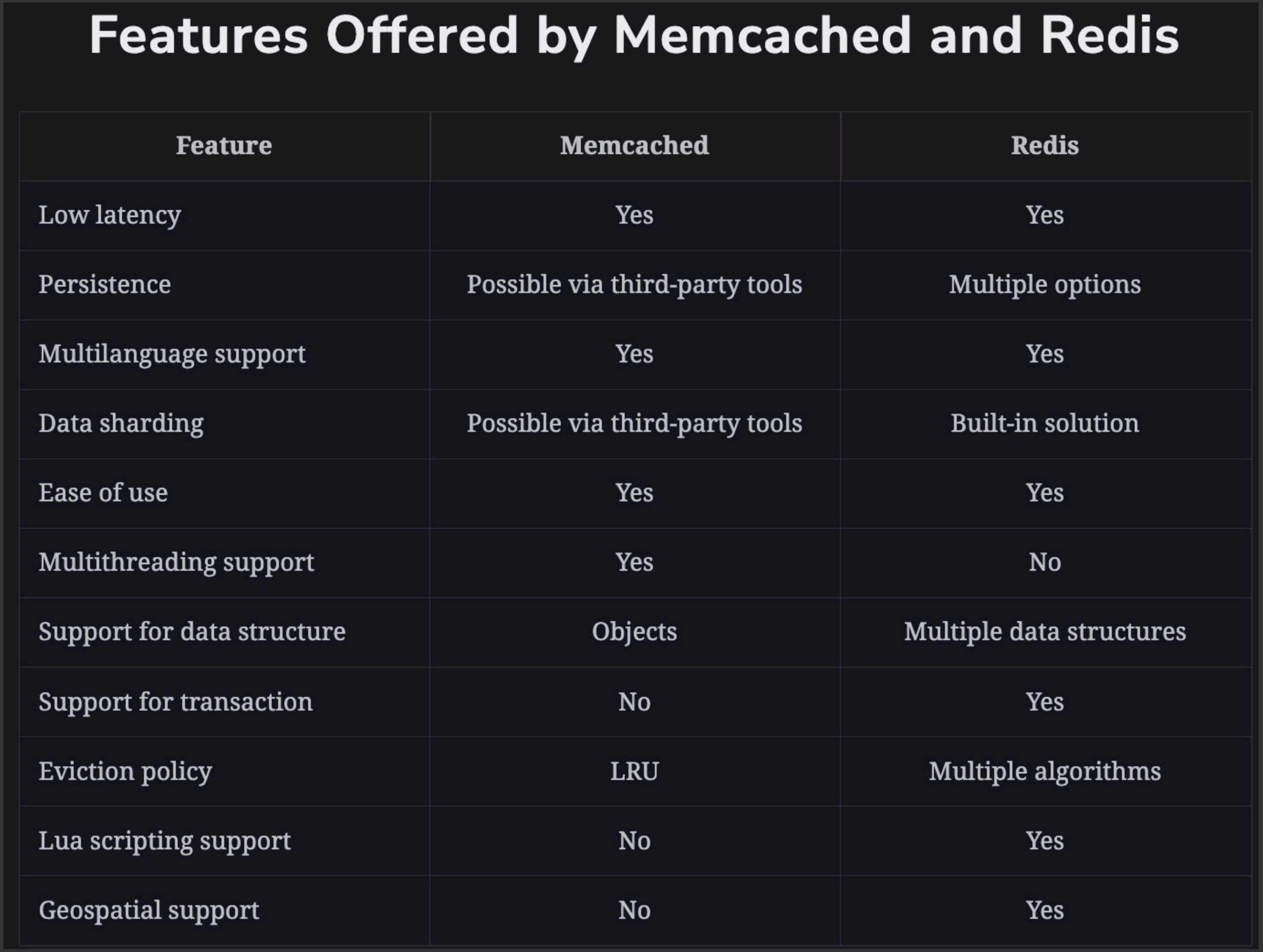
Database: It can persist all the in-memory blobs on the secondary storage.

Message broker: Asynchronous communication is a vital requirement in distributed systems. Redis can translate millions of messages per second from one component to another in a system.



Redis has built-in cluster support that provides high availability. This is called Redis Sentinel. A cluster has one or more Redis databases that are queried using multithreaded proxies. Redis clusters perform automatic sharding where each shard has primary and secondary nodes.


Primary
Secondary
Shard 1 and its replicas
Shard 2 and its replicas
Redis cluster
Cluster manager
Redis client
Primary
Secondary
Shard 3 and its replicas
Shard 4 and its replicas
Cluster manager
Redis cluster



Features Offered by Memcached and Redis

Feature	Memcached	Redis
Low latency	Yes	Yes
Persistence	Possible via third-party tools	Multiple options
Multilanguage support	Yes	Yes
Data sharding	Possible via third-party tools	Built-in solution
Ease of use	Yes	Yes
Multithreading support	Yes	No
Support for data structure	Objects	Multiple data structures
Support for transaction	No	Yes
Eviction policy	LRU	Multiple algorithms
Lua scripting support	No	Yes
Geospatial support	No	Yes