Evaluation of a Distributed Cache's Design

Let's evaluate our design in the context of our requirements.

We'll cover the following

Let’s evaluate our proposed design according to the design requirements.

High performance

educative

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.

> **Note:** A critical parameter for high performance is the selection of the eviction algorithm because the number of cache hits and misses depends on it. The higher the cache hit rate, the better the performance.

To get an idea of how important the eviction algorithm is, let's assume the following:

- Cache hit service time (99.9^{th} percentile): 5 ms
- Cach miss service time (99.9^{th} percentile): 30 ms (this includes time to get the data from the database and set the cache)

Let's assume we have a 10% cache miss rate using the most frequently used (MFU) algorithm, whereas we have a 5% cache miss rate using the LRU algorithm. Then, we use the following formula:

$$EAT = Ratio_{hit} \text{ x } Time_{hit} + Ratio_{miss} \text{ x } Time_{miss}$$

Here, this means the following:

EAT: Effective access time.

$Ratio_{hit}$: The percentage of times a cache hit will occur.

$Ratio_{miss}$: The percentage of times a cache miss will occur.

$Time_{hit}$: Time required to serve a cache hit.

$Time_{miss}$: Time required to serve a cache miss.

For MFU, we see the following:

```
EAT = 0.90 x 5 milliseconds + 0.10 x 30 milliseconds = 0.0045 + 0.003 = 0.007
5 = 7.5 milliseconds
```

For LRU, we see the following:

```
EAT = 0.95 x 5 milliseconds + 0.05 x 30 milliseconds =  0.00475 + 0.0015 = 0.0062
5 = 6.25 milliseconds.
```

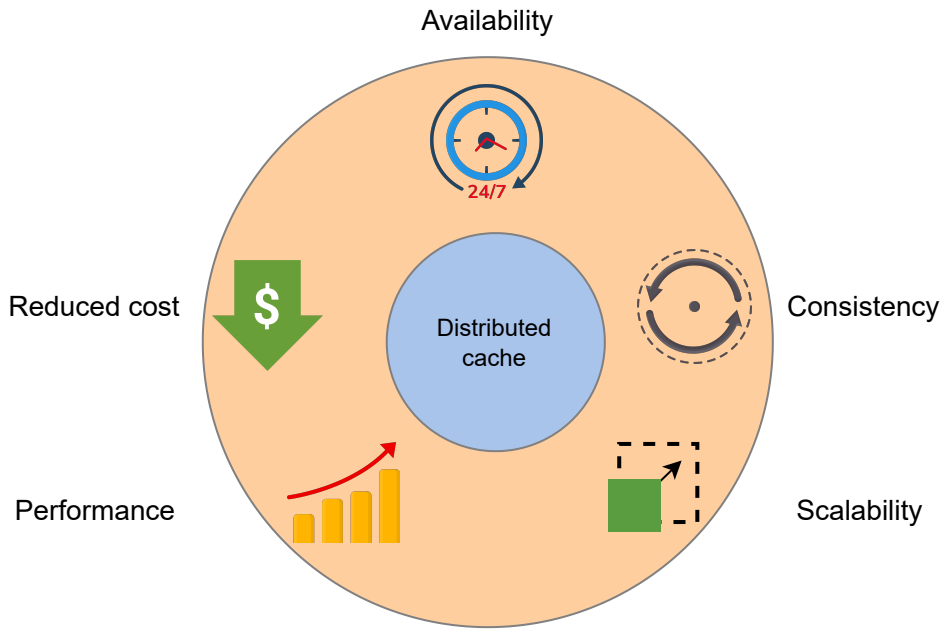
The numbers above highlight the importance of the eviction algorithm to increase the cache hit rate. Each application should conduct an empirical study to determine the eviction algorithm that gives better results for a specific workload.

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.



Guarantees provided by a distributed cache

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.

It's possible to achieve higher availability by splitting the leader and follower servers among different data centers. But such high availability comes at a price of consistency. We assumed synchronous writes within the same data center. But synchronous writing for strong consistency in different data centers has a serious performance implication that isn't welcomed in caching systems. We usually use asynchronous replication across data centers.

For replication within the data center, we can get strong consistency with good performance. We can compromise strong consistency across data center replication to achieve better availability (see CAP and PACELC theorems).

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.

Inconsistency can also arise from faulty configuration files and services. Imagine a scenario where a cache server is down during a write operation, and a read operation is performed on it just after its recovery. We can avoid such scenarios for any joining or rejoining server by not allowing it to serve requests until it's reasonably sure that it's up to date.

Affordability

Our proposed design has a low cost because it's feasible and practical to create such a system using commodity hardware.

Points to Ponder

Question 1

What happens if the leader node fails in the leader-follower protocol?

Show Answer

1 of 3

Summary

We studied the basics of the cache and designed a distributed cache that has a good level of availability, high performance, high scalability, and low cost. Our mechanism maintains high availability using replicas, though if all replicas are in one data center, such a scheme won't tackle full data center failures. Now that we've learned about the basics of design, let's explore popular open-source frameworks like Memcached and Redis.

Back

Detailed Design of a Distributed Cache

Next

Memcached versus Redis

Mark as Completed