Scalable Distributed Cache Using Consistent Hashing

Authors: Santosh Mohan (santoshm@stanford.edu), Nobelle Tay (nobellet@stanford.edu)
Code: https://github.com/thatindiandude/CS244B

Abstract

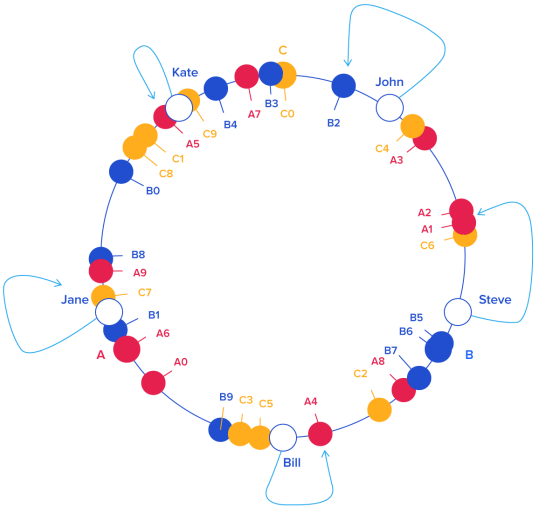
Distributed cache is useful for bypassing the memory limitations of using a single computer, which could be exponentially expensive, allowing for the construction of arbitrarily large hash tables using affordable servers. This paper explores different implementations for distributed cache and presents a case-study of how consistent hashing works. Besides that, we implemented a fully functional scalable distributed key-value store, demonstrating proof of concept. Moreover, we simulated various scenarios to explore different metrics of the key value store implemented, such as server load, query performance, and availability. We also implement the "load-aware" consistent hashing from Google's Zanzibar, a Consistent, Global Authorization System , which allows for replication based on the hotness of a particular cached object.

Server Model

We divide the cache into two components: the primary cache server and its workers. The primary cache is responsible for routing SET and GET requests to the appropriate cache worker servers, bringing up and down workers, and load balancing.

Set

For any incoming Set request, the primary server will try writing to the appropriate worker. In the event that the worker's memory is full, a new node is brought up and the new machine's hash will be computed. The insertion will be attempted again on the new node.



Similar to the figure on the left, we will produce multiple hashes to represent a single machine. This is to bring more eveness to the distribution of objects to our worker servers.

Get

For any incoming get requests, we will look through all workers containing this object. If none are found, the worker will return an error.

Based on how we designed the primary cache interface, we are able to configure caches such that

they can alter the **maximum capacity** of the cache, the **number of** initial **machines** or nodes supporting the cache, the initial **cache size**, the replication factor of each write, and the **hash function** being used. Bolded values correspond to configurable parameters used in the PrimaryCacheServer object.

Our implementation supports asynchronous requests to the server so that calls can be blocked by the cache depending on the underlying worker machine's current load. This exposes edge cases based on the cache's configuration, such as an increased cost from trying to make many read requests to a single, large cached object that hasn't been properly replicated. This would expose a lack of proper availability built into the cache's design.

Hash Functions

The built-in hash Python hash function, `hash()`, is known to not be a cryptographic hash. This results in an uneven distribution of objects in the consistent hash ring. We will also use the SHA256 cryptographic hash to understand if this will produce better results at scale.

Request Types

We test two different modes of making parallel requests: request hedging and awaiting. With 'hedging', we will return immediately on the first successful cache hit while we will wait for all parallel requests to return before proceeding with 'awaiting'.

I/O Operation Cost Modeling

Individual machines will need to be written to and read from. They will also need to compute hashes. This is a table of how we compute the I/O operation cost modeling.

I/O
Reading an object - Equivalent to object size*****
Writing an object - Equivalent to object size*****
Deleting an object - No cost is evaluated for deletions
***** *Modern SSDs like the Samsung 870 EVO series have a write speed and read speed that are relatively similar (560 MB/s read, 530 MB/s write)*
Cache Miss - Constant (5)
Read Database - Constant(500)

Consistent Hashing
Adding Node - Represents bringing up a new worker cache server. Constant (300)
Computing hash - Computing a hash for a machine or object. Constant (5)

Load Modeling

Since networking is hard to account for by the operations themselves, we encapsulate these using a per-worker load value. When a worker cache server is being written to, it has an additional load of two, and when it is read from, it has an additional load of one. The worker will block any incoming requests where the request's additional load would have the worker exceed its load threshold. In this way, `load_threshold` allows the cache to shed load routed to a busy server.

Scaling Strategies

No Replication
In a naive implementation, consistent hashing will make a single worker responsible for any given object. This maximizes the cache hit ratio and eliminates hot spots on the underlying backend, like Spanner.

Replication by Factor (RF)
Every time we write a new object to the cache, we will write the object to *N* number of replicas, where *N* is the replication factor the user has specified. The latency of the first write will be slower due to the multiple writes (and potential need to bring up new machines if neighbors are full), but will make the object much more available for later reads. This is especially beneficial if the object is hot.

Load-Aware Consistent Hashing (LA-CH)
In cases where there are very few objects, objects may be too hot for a single worker but not so numerous that replication would benefit all objects given the latency of replication. Hot workers must divert a large fraction of requests to a new worker, which is typical without replication. However, this introduces latency by the way of having to compute the next consistent hash and use a different worker. With tail latency, or the highest percentile latencies experienced by users, response times can be catastrophically long for a cache. Instead, using multiple workers for a hot object can reduce the need for recomputing the next hash.

To achieve this, we will have the primary cache server maintain a count of the number of outstanding operations per hash value.

Let *L*, or the *load threshold*, be a number such that if *L* many requests are made to the object, we add a node to handle follow-on requests. Naturally, lower *L* values lend to more latency initially due to more replication, but will naturally increase the availability of a hot object and reduce the average latency of read requests to the hot object over time.

For super hot objects, we will use all workers to forward requests to the single worker responsible for the object. As we use *L* to identify the hotness of an object on a worker, we will use 2*L on the Primary Cache Server to identify objects whose primary worker ID will be cached for faster access.

Deviations from Zanzibar

We make some deviations from the original paper to incorporate it into our simulation of a cache. The original paper has a single worker responsible for normal objects, with traffic diverted to the machine at the next consistent hash when the initial server is too busy. Its fix is to have the caller keep track of the load for an object and divert traffic on its own side only when the load for the object, not the single server, requires it. It fans out requests as a function of this load. We do not support redirecting traffic in our simulation. Instead, we add workers responsible for an object as its load increases and make parallel GET calls to multiple workers (request hedging), in the `replication_factor` and `load_aware` cases to reduce the tail latency.

At the limit, `load_aware` reads on super hot objects would divert traffic to all workers, reducing the cache hit ratio on any single worker. The paper presents an alternative where the workers will redirect traffic to a single worker to maximize cache hits. This forms a “two-cache hierarchy”. Instead, we cache the worker/machine indexes allocated for the super hot object when it was just considered a hot object and force a call to the first machine responsible for it if these all result in a miss, mimicking the two layers of caching for the single object.

Scenario Testing

Our battery of tests include faults to workers nodes. Given the replication available in specific scaling methods, we expect RF and LA-CH scaling schemes to keep cached objects available despite these faults. We do not test resilience to the failure of our primary node.

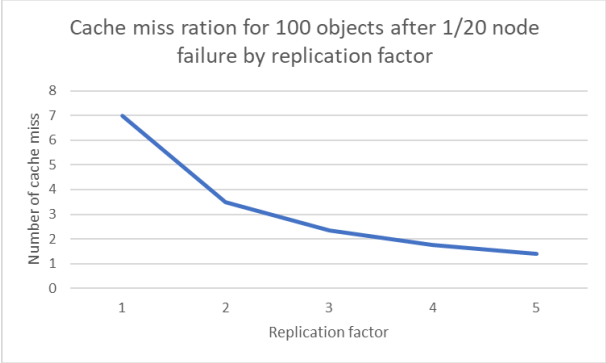
Tests

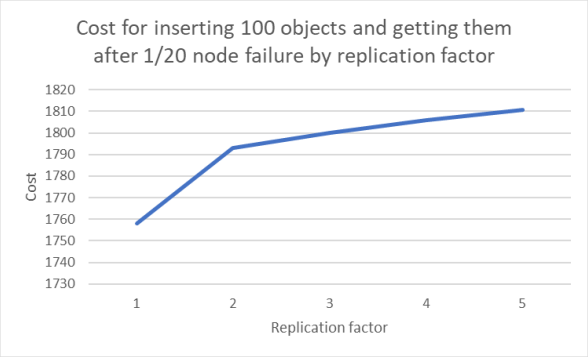
BasicWriteAndRead() - Tests whether the cache can asynchronously write and read multiple objects.
BasicWriteAndReadWithNodeFailures() - The same test as BasicWriteAndRead() with node failures before caches are read from.

Metrics

A good cache will keep reads fast and the cached objects available. A great cache will maximize the availability of a super hot object. The smaller the total scenario cost, the better the availability of an object should be on GET calls.

Results





Cache miss happens less frequently when the replication factor is greater. In case of a cache miss in a worker due to eviction or failure, the primary server can attempt to send the GET request to another worker(s) up to the replication factor, resulting in a decrease in cache miss ratio.

A Get request is more expensive when the replication factor is greater because a GET request is routed to another worker in case of a cache miss in the first worker, and more up till the replication factor, resulting in increase in latency.

A SET request is also more expensive when the replication factor is greater due to the increase in storage used. When the memory of a worker is fully used, a new node is brought up on demand, resulting in higher latency.

Note that the SET request latency is not affected by synchronous routing to workers because the primary cache server returns success when the object insertion to the first node is successful and does not wait for the asynchronous object insertion to the remaining nodes. That being said, if the architecture is modified such that the primary cache server issues GET requests to multiple nodes in parallel, we expect to see the request latency to remain the same even for various replication factors. We did not implement this architecture to reduce the server load assuming a low write and read heavy workload, such as social media posts where the read to write ratio is high, especially for trending posts.

As indicated by the results, it is evident that there is a tradeoff between the cache miss ratio and GET request latency. Depending on the expected workload and requirements, we can determine the ideal parameter.

Hash Functions

We saw no difference between using `hash()` and `hashlib.sha256()` in I/Os or cache hit ratio.

Request Hedging vs Awaiting All Requests

Scaling Strategy	Replication Factor	Cache Size per Machine	# of Writes	Cost	Request Type	Cache Miss Ratio
RF	5	1000	500	509672	Hedging	0.916
RF	9	10000	2000	439466	Hedging	0.137
Load Aware	5	1000	1000	588423	Hedging	0.57
Load Aware	9	1000	1000	608403	Hedging	0.597
RF	9	1000	2000	210000	Awaiting	0.0
Load Aware	5	1000	2000	22742	Awaiting	0.0

We find that hedging is costly and awaiting, while allowing for tail latency, does allow for objects to stay in caches long enough to be retrieved on the first call, even with a small replication factor. In the case of hedging, we would have to wait for another GET request for the object to now be resident in some machine's cache. Comparing the last two rows, at lower replication factors, we can see that we bear 1/10th of the performance cost for similar performance in `load_aware` jobs.

As we increase the cache size per-machine, the cache miss ratio with hedging starts to improve but is still not reliable, and the increased cost shows that the cache is thrashing without enough memory on workers.

Future Work

Right now, we use a form of load shedding whereby multiple writes/reads to busy servers will fail an incoming request. We could experiment with using task queues and/or a load balancer to limit the amount per worker while minimizing the latency introduced to keep the cache fast. We should also test whether a single large object can be extremely available despite multiple concurrent readers.

References

Arpit, "Consistent hashing," Medium, 25-May-2020. [Online]. Available: https://levelup.gitconnected.com/consistent-hashing-27636286a8a9. [Accessed: 03-Jun-2022].

J. P. Carzolio, “A guide to consistent hashing,” Toptal Engineering Blog, 25-Apr-2017. [Online]. Available: https://www.toptal.com/big-data/consistent-hashing?fbclid=IwAR2P2ZXlQZmeVHLtR4--upq9VnOqwBtlxObXtY9I9DegDbSG9p2n7l5_xFk. [Accessed: 03-Jun-2022].
O. Elgabry, “Consistent hashing: Beyond the basics,” Medium, 24-Nov-2020. [Online]. Available: https://medium.com/omarelgabrys-blog/consistent-hashing-beyond-the-basics-525304a12ba. [Accessed: 03-Jun-2022].

Pang, Ruoming, et. al. “Zanzibar:{Google’s} Consistent, Global Authorization System.” *2019 USENIX Annual Technical Conference (USENIX ATC 19). 2019.*