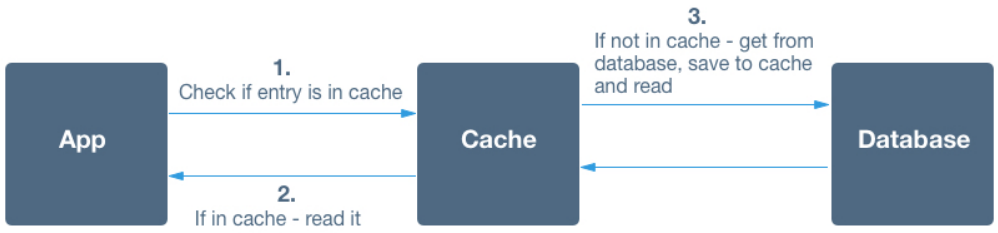
Design a Cache

Design a distributed key value caching system, like Memcached or Redis.



Features:

This is the first part of any system design interview, coming up with the features which the system should support. As an interviewee, you should try to list down all the features you can think of which our system should support. Try to spend around 2 minutes for this section in the interview. You can use the notes section alongside to remember what you wrote.

- **Q:** What is the amount of data that we need to cache?
  A: Let's assume we are looking to cache on the scale of Google or Twitter. The total size of the cache would be a few TBs.

- **Q:** What should be the eviction strategy?
  A: It is possible that we might get entries when we would not have space to accommodate new entries. In such cases, we would need to remove one or more entries to make space for the new entry.

Q: What should be the access pattern for the given cache?
A: There are majorly three kinds of caching systems :

- **Write through cache :** This is a caching system where writes go through the cache and write is confirmed as success only if writes to DB and the cache BOTH succeed. This is really useful for applications which write and re-read the information quickly. However, write latency will be higher in this case as there are writes to 2 separate systems.
- **Write around cache :** This is a caching system where write directly goes to the DB. The cache system reads the information from DB incase of a miss. While this ensures lower write load to the cache and faster writes, this can lead to higher read latency incase of applications which write and re-read the information quickly.
- **Write back cache :** This is a caching system where the write is directly done to the caching layer and the write is confirmed as soon as the write to the cache completes. The cache then asynchronously syncs this write to the DB. This would lead to a really quick write latency and high write throughput.

But, as is the case with any non-persistent / in-memory write, we stand the risk of losing the data incase the caching layer dies. We can improve our odds by introducing having more than one replica acknowledging the write (so that we don't lose data if just one of the replica dies).

Estimation:

This is usually the second part of a design interview, coming up with the estimated numbers of how scalable our system should be. Important parameters to remember for this section is the number of queries per second and the data which the system will be required to handle.
Try to spend around 5 minutes for this section in the interview.

Total cache size : Let's say 30TB as discussed earlier.

Q: What is the kind of QPS we expect for the system?
A: This estimation is important to understand the number of machines we will need to answer the queries. For example, if our estimations state that a single machine is going to handle 1M QPS, we run into a high risk of high latency / the machine dying because of queries not being answered fast enough and hence ending up in the backlog queue.
Again, let's assume the scale of Twitter / Google. We can expect around 10M QPS if not more.

Q: What is the number of machines required to cache?
A: A cache has to be inherently of low latency. Which means all cache data has to reside in main memory.
A production level caching machine would be 72G or 144G of RAM. Assuming beefier cache machines, we have 72G of main memory for 1 machine. Min. number of machine required = 30 TB / 72G which is close to 420 machines.
Do know that this is the absolute minimum. Its possible we might need more machines because the QPS per machine is higher than we want it to be.

The way QPS is calculated is as follows : Earlier we mentioned that each machine would have a RAM of 72 GB of RAM. For serving 30TB of cache, the number of machines required would be 30 TB / 72G which is close to 420. Assume that we have 420 machines to server 30 TB of distributed cache. Now regarding the QPS the requirement was 10 M

Now per machine the QPS would be 10M / 420 = Approximately 23000 QPS. So this meant per machine should be able to handle 23,00 QPS. The approach is similar to how we decided on the number of machines based on the per machine RAM and the total cache size. Similarly for the QPS, it is based on the total QPS / number of machines.

Next assuming that a machine has to serve 23,000 QPS then we look at each machine has 4 core and then we calculate the per request time as - CPU time available per query = 4 * 1000 * 1000 / 23000 microseconds = 174us (Note everything is converted to milliseconds.) So the machines have to return the query in 174 us. This is the way the QPS is derived. Then based on the read / write traffic and the latency numbers as per the https://gist.github.com/jboner/2841832, the QPS is further refined by increasing the number of machines.

Design Goals:

- **Latency** - Is this problem very latency sensitive (Or in other words, Are requests with high latency and a failing request, equally bad?). For example, search typeahead suggestions are useless if they take more than a second.
- **Consistency** - Does this problem require tight consistency? Or is it okay if things are eventually consistent?
- **Availability** - Does this problem require 100% availability?
  There could be more goals depending on the problem. It's possible that all parameters might be important, and some of them might conflict. In that case, you'd need to prioritize one over the other.

Q: Is Latency a very important metric for us?
A: Yes. The whole point of caching is low latency.
Q: Consistency vs Availability?
A: Unavailability in a caching system means that the caching machine goes down, which in turn means that we have a cache miss which leads to a high latency.

As said before, we are caching for a Twitter / Google like system. When fetching a timeline for a user, I would be okay if I miss on a few tweets which were very recently posted as long as I eventually see them in reasonable time.
Unavailability could lead to latency spikes and increased load on DB. Choosing from consistency and availability, we should prioritize for availability.

Deep Dive:

Lets dig deeper into every component one by one. Discussion for this section will take majority of the interview time(20-30 minutes).

Q: How would a LRU cache work on a single machine which is single threaded?
Q: What if we never had to remove entries from the LRU cache because we had enough space, what would you use to support and get and set?
A: A simple map / hashmap would suffice.

Q: How should we modify our approach if we also have to evict keys at some stage?
A: We need a data structure which at any given instance can give me the least recently used objects in order. Let's see if we can maintain a linked list to do it. We try to keep the list ordered by the order in which they are used.
So whenever, a get operation happens, we would need to move that object from a certain position in the list to the front of the list. Which means a delete followed by insert at the beginning. Insert at the beginning of the list is trivial. How do we achieve erase of the object from a random position in least time possible? How about we maintain another map which stores the value to the corresponding linked list node.
Ok, now when we know the node, we would need to know its previous and next node in the list to enable the deletion of the node from the list. We can get the next in the list from next pointer ? What about the previous node ? To encounter that, we make the list doubly linked list.
Head over to https://www.interviewbit.com/problems/least-recently-used-cache/ to write code and see if you completely got it.

A: Since we only have one thread to work with, we cannot do things in parallel. So we will take a simple approach and implement a LRU cache using a linked list and a map. The Map stores the value to the corresponding linked list node and is useful to move the recently accessed node to the front of the list.
Head over to https://www.interviewbit.com/problems/least-recently-used-cache/ to write code and see if you completely got it.

Q: How would a LRU cache work on a single machine which is multi threaded ?
Q: How would you break down cache write and read into multiple instructions?
A:
Read path : Read a value corresponding to a key. This requires :

- Operation 1 : A read from the HashMap and then,
- Operation 2 : An update in the doubly LinkedList

Write path : Insert a new key-value entry to the LRU cache. This requires :

- If the cache is full, then
  - Operation 3: Figure out the least recently used item from the linkedList
  - Operation 4: Remove it from the hashMap
  - Operation 5: Remove the entry from the linkedList.
- Operation 6: Insert the new item in the hashMap
- Operation 7: Insert the new item in the linkedList.

Q: How would you prioritize above operations to keep latency to a minimum for our system?
A: As is the case with most concurrent systems, writes compete with reads and other writes. That requires some form of locking when a write is in progress. We can choose to have writes as granular as possible to help with performance.
Read path is going to be highly frequent. As latency is our design goal, Operation 1 needs to be really fast and should require minimum locks. Operation 2 can happen asynchronously. Similarly, all of the write path can happen asynchronously and the client's latency need not be affected by anything other than Operation 1. Let's dig deeper into Operation 1. What are the things that Hashmap is dealing with?
Hashmap deals with Operation 1, 4 and 6 with Operation 4 and 6 being write operations. One simple, but not so efficient way of handling read/write would be to acquire a higher level Read lock for Operation 1 and Write lock for Operation 4 and 6.
However, Operation 1 as stressed earlier is the most frequent (by a huge margin) operation and its performance is critical to how our caching system works.

Q: How would you implement HashMap?
A: The HashMap itself could be implemented in multiple ways. One common way could be hashing with linked list (colliding values linked together in a linkedList) :
Let's say our hashmap size is N and we wish to add (k,v) to it
Let H = size N array of pointers with every element initialized to NULL
For a given key k, generate g = hash(k) % N newEntry = LinkedList Node with value = v
newEntry.next = H[g]
H[g] = newEntry

More details at https://en.wikipedia.org/wiki/Hash_table
Given this implementation, we can see that instead of having a lock on a hashmap level, we can have it for every single row. This way, a read for row i and a write for row j would not affect each other if i != j. Note that we would try to keep N as high as possible here to increase granularity.

A: The key to understanding and optimizing concurrency problems lies in breaking the problem down into as granular parts as possible.
As is the case with most concurrent systems, writes compete with reads and other writes, which requires some form of locking when a write is in progress. We can choose to have writes as granular as possible to help with performance. Instead of having a lock on a hashmap level if we can have it for every single row, a read for row i and a write for row j would not affect each other if i != j. Note that we would try to keep N as high as possible here to increase granularity.

Q: Now that we have sorted how things look on a single server, how do we shard?
Q: What QPS would a machine have to handle if we shard in blocks of 72GB?
A: In estimation section, we saw total data we would have to store is 30TB. For every chunk of data, we store a copy in the hashmap and we store an entry (without the value) in a linkedList. Let's assume that the size of value is big enough to ignore overheads like an entry in linkedList.
We can accommodate 72G of data on every single machine (We have neglected process memory overheads for the time being). With that, we would need 420 machines.
With that config, every machine would handle around 23000 QPS.

Q: Will our machines be able to handle qps of 23000?
A: CPU time available for 23k queries : 1 second * 4 = 4 seconds
CPU time available per query = 4 * 1000 * 1000 / 23000 microseconds = 174us. Can we handle entries into a hashmap of size 72G with a CPU time of 174us (Do note that context switches has its own overhead. So, even with a perfectly written asynchronous server, we would have much less than 174us on our hand). Make sure you know about the latency numbers from here : https://gist.github.com/jboner/2841832. The actual answer depends on the distribution of read vs write traffic, the size of the value being read, the throughput capacity of our server.

Q: What if we shard among machines with 16GB of RAM?
A: Number of shards = 30 * 1000 / 16 = 1875
This leads to a QPS of approximately 5500 per shard which should be reasonable (Note that with lower main memory size, CPU cycles required for access lowers as well).Now, we also need to decide the shard number for every key. A simple way to do it would be to shard based on hash(key) % TOTAL_SHARDS The hash function might differ based on expected properties of the key. If the key is an auto-incremental user_id, then hash(key) = key hashing might work well.
One downside to this is that if the total number of shard changes, all the currently cached data becomes invalid and all requests would have to hit the DB to warm up the cache. The other way to do it would be to use consistent hashing with multiple copies of every shard on the ring (Read more about consistent hashing at https://en.wikipedia.org/wiki/Consistent_hashing). This would perform well as new shards are added.

A: Recall that the total data we have is 30TB and for every chunk of data, we store a copy in the hashmap and we store an entry (without the value) in a linkedList. Lets assume that the size of value is big enough to ignore overheads like an entry in linkedList. We can accommodate 72G of data on every single machine (We have neglected process memory overheads for the time being). With that, we would need 420 machines which would lead to a QPS of 23000 which is not easily feasible. So we rather create shards of 16GB, 1875 shards each supporting qps of 5500.

Q: What happens when a machine handling a shard goes down?

A: If we only have one machine per shard, then if the machine goes down, all requests to that shard will start hitting the DB and hence there will be elevated latency.
As mentioned in the design goals, we would like to reduce high latency cases as much as possible.
If we have a lot of machines, one way to avoid these cases would be to have multiple machines per shard where they maintain exactly the same amount of data.
A read query for the shard could go to all the servers in the shard and we can use the data from the one that responds first. This takes care of one machine going down, but introduces a bunch of other complications. If occasional high latency is not a big issue wrt product, its better to stick to one server per shard (Less maintenance overhead and a much simpler system).
Complications of multiple servers : Since we have multiple servers maintaining the same data, it is possible that the data is not in sync between the servers. This means that a few keys might be missing on some of the servers, and a few servers might have older values for the same keys (Assuming we support updates as well).
Imagine a case when one of the server goes down, misses a bunch of additions and updates, and then comes back up.
There are few ways we can approach this :

- Master slave technique : There is only one active server at a time in a shard and it has a follower which keeps getting the update. When the master server goes down, the slave server takes over as the master server. Master and slave can maintain a change log with version number to make sure they are caught up.
- If we are fine with all servers becoming eventually consistent, then we can have one master (taking all the write traffic) and many slaves where slaves can service the read traffic as well.