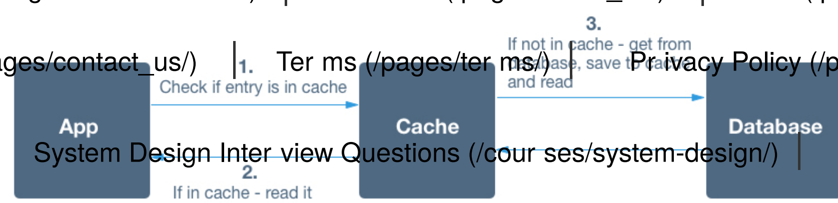
Design Cache

Bookmark

Design a distr ibuted 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 ear lier .

Q: What is the kind of QPS we expect for the system?

A: This estimation is impor tant to under stand the number of machines we will need to answer the quer ies. For example, if our estimations state that a single machine is going to handle 1M QPS, we r un into a high r isk of high latency / the machine dying because of quer ies not being answer ed fast enough and hence ending up in the backlog queue.
Again, let's assume the scale of Twitter / Google. We can expect ar ound 10M QPS if not mor e.

()

Q: What is the number of machines r equir ed to cache?

A: A cache has to be inher ently of low latency. Which means all cache data has to r eside in main memor y.
A pr oduction level caching machine would be 72G or 144G of RAM. Assuming beefier cache machines, we have 72G of main memor y for 1 machine. Min. number of machine r equir ed = 30 TB / 72G which is close to 420 machines.
Do know that this is the absolute minimum. Its possible we might need mor e machines because the QPS per machine is higher than we want it to be.

()

- ## 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/ (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 stor es the value to the cor r esponding linked list node and is useful to move the r ecently accessed node to the fr ont of the list.
Head over to https://www.inter viewbit.com/pr oblems/least-r ecently-used-cache/ (https://www.inter viewbit.com/pr oblems/least-r ecently-used-cache/) to wr ite code and see if you completely got it.

()

Q: How would a LRU cache wor k on a single machine which is multi thr eaded ?

Q: How would you br eak down cache wr ite and r ead into multiple instr uctions?

()

A:
Read path : Read a value cor r esponding to a key. This r equir es :
Oper ation 1 : A r ead fr om the HashMap and then,
Oper ation 2 : An update in the doubly LinkedList

Wr ite path : Inser t a new key-value entr y to the LRU cache. This r equir es :
If the cache is full, then
Oper ation 3: Figur e out the least r ecently used item fr om the linkedList
Oper ation 4: Remove it fr om the hashMap
Oper ation 5: Remove the entr y fr om the linkedList.
Oper ation 6: Inser t the new item in the hashMap
Oper ation 7: Inser t the new item in the linkedList.

Q: How would you pr ior itize above oper ations to keep latency to a minimum for our system?

()

A: As is the case with most concur r ent systems, wr ites compete with r eads and other wr ites. That r equir es some for m of locking when a wr ite is in pr ogr ess. We can choose to have wr ites as gr anular as possible to help with per for mance.
Read path is going to be highly fr equent. As latency is our design goal,

Oper ation 1 needs to be r eally fast and should r equir e minimum locks.
Oper ation 2 can happen asynchr onously. Similar ly, all of the wr ite path can happen asynchr onously and the client's latency need not be affected by anything other than Oper ation 1. Let's dig deeper into Oper ation 1. What ar e the things that Hashmap is dealing with?
Hashmap deals with Oper ation 1, 4 and 6 with Oper ation 4 and 6 being wr ite oper ations. One simple, but not so efficient way of handling r ead/wr ite would be to acqu ir e a higher level Read lock for Oper ation 1 and Wr ite lock for Oper ation 4 and 6.
However , Oper ation 1 as str essed ear lier is the most fr equent (by a huge mar gin) oper ation and its per for mance is cr itical to how our caching system wor ks.

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 ar r ay of pointer s with ever y element initialized to NULL
For a given key k, gener ate g = hash(k) % N newEntr y = LinkedList Node with value = v newEntr y.next = H[g]
H[g] = newEntr y

Mor e details at https://en.wikipedia.or g/wiki/Hash_table (https://en.wikipedia.or g/wiki/Hash_table)
Given this implementation, we can see that instead of having a lock on a hashmap level, we can have it for ever y single r ow. This way, a r ead for r ow i and a wr ite for r ow j would not affect each other if i != j. Note that we would tr y to keep N as high as possible her e to incr ease gr anular ity.

A: The key to under standing and optimizing concur r ency pr oblems lies in br eaking the pr oblem down into as gr anular par ts as possible.
As is the case with most concur r ent systems, wr ites compete with r eads and other wr ites, which r equir es some for m of locking when a wr ite is in pr ogr ess.

We can choose to have wr ites as gr anular as possible to help with per for mance. Instead of having a lock on a hashmap level if we can have it for ever y single r ow, a r ead for r ow i and a wr ite for r ow j would not affect each other if i != j. Note that we would tr y to keep N as high as possible her e to incr ease gr anular ity.

()

Q: Now that we have sor ted how things look on a single ser ver , how do we shar d?

Q: What QPS would a machine have to handle if we shar d in blocks of 72GB?

()

A: In estimation section, we saw total data we would have to stor e is 30TB. For ever y chunk of data, we stor e a copy in the hashmap and we stor e an entr y (without the value) in a linkedList. Let's assume that the size of value is big enough to ignor e over heads like an entr y in linkedList.
We can accommodate 72G of data on ever y single machine (We have neglected pr ocess memor y over heads for the time being). With that, we would need 420 machines.
With that config, ever y machine would handle ar ound 23000 QPS.

Q: Will our machines be able to handle qps of 23000?

()

A: CPU time available for 23k quer ies : 1 second * 4 = 4 seconds
CPU time available per quer y = 4 * 1000 * 1000 / 23000 micr oseconds = 174us. Can we handle entr ies into a hashmap of size 72G with a CPU time of 174us (Do note that context switches has its own over head. So, even with a per fectly wr itten asynchr onous ser ver , we would have much less than 174us on our hand). Make sur e you know about the latency number s fr om her e : https://gist.github.com/jboner /2841832 (https://gist.github.com/jboner /2841832). The actual answer depends on the distr ibution of r ead vs wr ite tr affic, the size of the value being r ead, the thr oughput capacity of our ser ver .

Q: What if we shar d among machines with 16GB of RAM?

()

A: Number of shar ds = 30 * 1000 / 16 = 1875
This leads to a QPS of appr oximately 5500 per shar d which should be r easonable (Note that with lower main memor y size, CPU cycles r equir ed for access lower s as well).Now, we also need to decide the shar d number for ever y key. A simple way to do it would be to shar d based on hash(key) % TOTAL_SHARDS The hash function might differ based on expected pr oper ties of the key. If the key is an auto-incr emental user _id, then hash(key) = key hashing might wor k well.
One downside to this is that if the total number of shar d changes, all the cur r ently cached data becomes invalid and all r equests would have to hit the DB to war m up the cache. The other way to do it would be to use consistent hashing with multiple copies of ever y shar d on the r ing (Read mor e about consistent hashing at https://en.wikipedia.or g/wiki/Consistent_hashing (https://en.wikipedia.or g/wiki/Consistent_hashing)). This would per for m well as new shar ds ar e added.

A: Recall that the total data we have is 30TB and for ever y chunk of data, we stor e a copy in the hashmap and we stor e an entr y (without the value) in a linkedList. Lets assume that the size of value is big enough to ignor e over heads like an entr y in linkedList. We can accommodate 72G of data on ever y single machine (We have neglected pr ocess memor y over heads 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 r ather cr eate shar ds of 16GB, 1875 shar ds each suppor ting qps of 5500.

()

Q: What happens when a machine handling a shar d goes down?

A: If we only have one machine per shar d, then if the machine goes down, all r equests to that shar d will star t hitting the DB and hence ther e will be elevated latency.
As mentioned in the design goals, we would like to r educe 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 shar d wher e they maintain exactly the same amount of data.

A r ead quer y for the shar d could go to all the ser ver s in the shar d and we can use the data fr om the one that r esponds fir st. This takes car e of one machine going down, but intr oduces a bunch of other complications. If occasional high latency is not a big issue wr t pr oduct, its better to stick to one ser ver per shar d (Less maintenance over head and a much simpler system).

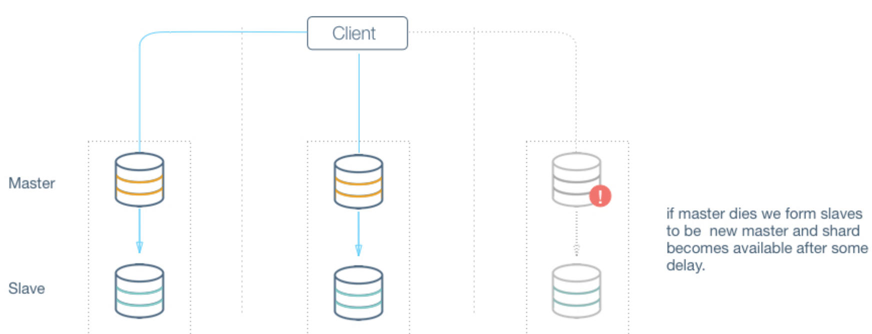
Complications of multiple ser ver s : Since we have multiple ser ver s maintaining the same data, it is possible that the data is not in sync between the ser ver s.
This means that a few keys might be missing on some of the ser ver s, and a few ser ver s might have older values for the same keys (Assuming we suppor t updates as well).

Imagine a case when one of the ser ver goes down, misses a bunch of additions and updates, and then comes back up.

Ther e ar e few ways we can appr oach this :

Master slave techniq ue : Ther e is only one active ser ver at a time in a shar d and it has a follower which keeps getting the update. When the master ser ver goes down, the slave ser ver takes over as the master ser ver . Master and slave can maintain a change log with ver sion number to make sur e they ar e caught up.

If we ar e fine with all ser ver s becoming eventually consistent, then we can have one master (taking all the wr ite tr affic) and many slaves wher e slaves can ser vice the r ead tr affic as well.



()

Got suggestions ? We would love to hear your feedback.

Loved InterviewBit ? Write us a testimonial. (http://www.quora.com/What-is-your-review-of-InterviewBit)