COMP 480/580 — Probabilistic Algorithms and Data Structure Jan 29, 2018

Lecture 7: Caching with SPOCA

Lecturer: Anshumali Shrivastava *Scribe By: Alexander Jiang*

1 Problem Context: Yahoo! Video Platform

Yahoo!'s video platform has over 20 million videos, and users make about 800,000 unique video requests per day (out of 30 million total requests per day). Yahoo!'s front-end servers can only store 500 unique videos in memory (100,000 unique videos on disk).

⇒ The low ratio of unique requests to total requests and the low amount of unique videos that can be stored on each server make it difficult to decide what to cache in the servers (to maximize cache hits = requests for videos that can be served from the cache).

Goal Design a **request router** that maps video filenames to server IDs. This router will also be used to decide where to cache videos.

Requirements:

1. **Stateless Addressing**: Servers can be added and deleted at any time (e.g. adding/removing servers to handle spikes in demand, servers failing, etc.).
2. **Efficiency**: Since each request will generate a call to the request router, the routing algorithm must be efficient.
3. **Heterogeneous pool of servers**: Servers have different capacities (server i has capacity c_i), and each server's load should be proportional to its capacity.

In the previous lecture, we saw a hashing scheme that meets the first two requirements (in particular, the first requirement of being able to handle dynamic addition/deletion of servers): consistent hashing.

2 Consistent Hashing Recap

Consistent hashing almost meets all of the requirements above: consistent hashing handles dynamic addition/deletion of servers (a.k.a. machines), and is efficient ($O(\log n)$ worst-case search time, where n is the number of machines).

Why not just hash videos to server IDs? We can hash objects to a fixed number of servers, but at large scales (e.g. Yahoo! Video Platform), servers will frequently fail and be replaced. In a "simple" hashing scheme where video IDs are hashed to machine IDs, you would have to reallocate all objects each time a server is added or removed. This is infeasible (and in general, we want to reduce the number of transfers of objects between machines that are needed to handle added/deleted machines).

In consistent hashing, if a server is added or deleted, we only need to transfer objects between two servers (the server that is being added or deleted and one other server), which means adding/removing servers doesn't add a lot of network traffic.

Consistent Hashing Idea Hash machines and objects to the same (circular) hash range. Then map objects to machines based on their hash locations.

Details

- To place object q, start at $h(q)$ and go to the right until you land in a slot that a machine m_i was hashed to, and place q in machine m_i (i.e. $h(m_i)$ is the first machine hash value to the right of $h(q)$).
- If a machine is removed, remove it from the range, and transfer all objects that were mapped to the removed machine to the next machine to the right in the range.
- If a machine is added, hash it into the range, and transfer objects from the next machine to the right in the range to the newly added machine.
- Since traversing the hash range (to figure out which machine is closest to the right) is potentially slow (i.e. $O(R)$ worst-case time where R is the hash range size), store the machine hash locations $h(m_i)$ in a binary search tree, which allows us to find the next machine to the right of a given hash location in $O(\log n)$ worst-case time.

2.1 Issues with Consistent Hashing

Though consistent hashing allows for stateless addressing (i.e. being able to add or remove servers at will), it has two major flaws that prevent it from being used for the Yahoo! Video Platform problem described above.

1. **Domino Effect**

   Even though the load is balanced in expectation, one server failing means that all of its load is transferred to the next server to the right in the hash range. This transfer might overload the next server, and then that server would fail, causing a chain reaction of failures.

2. **Non-Proportional Distribution**

   As mentioned above, consistent hashing assumes each server has equal capacity (i.e. if servers do not have the same capacity, consistent hashing will not give higher-capacity servers proportionally higher loads).

2.2 Proposal: Proportional Consistent Hashing

In the previous lecture, we proposed a modification to consistent hashing to reduce the variance of the load on each machine: make an equal number of "clones" of each machine, then apply consistent hashing (except mapping an object to any "clone" of the machine is the same as mapping the object to the real machine).

Proportional Consistent Hashing Idea Make a different number of "clones" of each machine in proportion to its capacity.

For example, if we have three servers (A, B, and C) with capacities 2, 3, and 4, respectively, then make 2 clones of server A, 3 clones of server B, and 4 clones of server C. This should achieve proportional distribution: in expectation, each clone will have equal load, so the expected loads

are proportional. This also somewhat eliminates the domino effect of failures, as when one server fails, each of its clones transfers its load to the next machine clone to the right, and these clones might not belong to the same real machine, making cascading failures due to overloading less likely.

However, the SPOCA paper doesn't explore this potential solution, as this variation of consistent hashing isn't usually found in books/literature.

3 SPOCA Algorithm

SPOCA stands for Stateless, Proportional, Optimally-Consistent Addressing.

SPOCA Idea Keep repeatedly hashing $h(v), h(h(v)), h(h(h(v))), ...$ until you land into a valid address range for a machine (machine range size = machine capacity). This means that, even if you add or remove machines, you can quickly route videos to machines (assuming you don't have to try too many different locations, see the Performance Analysis Below).

Details

1. Create an address space with total capacity C greater than the total capacity of the servers (usually $C = 2\sum_i c_i$ i.e. twice the sum of the server capacities).

2. Allocate a contiguous address range to each server: first, try to allocate the address range of size c_i that starts at $h(s_i)$ (i.e. $h(s_i)$ to $h(s_i) + c_i$). If that address range isn't free, try to allocate a c_i-sized range starting at $h(h(s_i))$ (and if that isn't free, then try the range starting at $h(h(h(s_i)))$, and so on).

3. To hash a video v, look at address $h(v)$. If this address is in the range of an active server, then the video request should be routed to that server (and add the video to the server's cache). If not, try $h(h(v))$, then $h(h(h(v)))$, and so on.

4. To add a new server s_j, allocate an address range for s_j as above. Then if any video v is now routed to server s_j, add that video to s_j's cache, and find where the video was previously cached by repeatedly applying the hash function until you land in another server's address range (assuming there was a previous request for v, which can be checked with a Bloom filter).

5. To delete a server, mark its address range as free, and for each video in its cache, repeatedly apply the hash function (i.e. look in addresses $h(v), h(h(v)), h(h(h(v))), ...$) until you land in the range of an active server, and add the video to that server's cache.

3.1 Performance Analysis

Why not just map servers to address ranges sequentially? Such a mapping would need to be updated as the servers are added/deleted dynamically, and gaps in the mapping (from deleted servers) would need to be handled carefully to utilize the entire address space well.

What if there isn't enough contiguous space to add a new server? This is a potential problem. We can make this less likely by choosing the initial address space to be large enough (but making the initial address space too large could mean that you have to try many times to hash a video to a machine).

How many locations will we need to look in before landing in a server's range (when routing a request for a video)?

Note that the probability of a hashed value landing in a specific server's range equals the server's capacity divided by the total hash range size (assuming h is a universal hash function and that the load is evenly distributed over the hash range). Thus if the total hash range size is twice the total server capacity, the probability that any given hash lands in an empty space is $1/2$, which means:

$E[\text{number of tries to land in any server's range}] = 2$

This is constant expected time for insertion! (Compare to $O(\log n)$ worst-case time with consistent hashing.) Additionally, the distribution of the number of tries needed to land in a server's range has sharp tails (i.e. the probability decays exponentially as you move further from the mean).

In addition, the $O(\log n)$ worst-case insertion time with consistent hashing comes with some disadvantages: binary search trees are difficult to update and access in parallel.

4 Handling Popular Videos: Zebra System

For a popular video, caching the video on one server isn't enough: there's too much demand for one server to handle. So we need to route requests for a popular video to multiple servers, each with the popular video in their cache. But how do we determine which videos are popular?

Idea Use Bloom filters (this is a set membership problem: is this specific video in the set of popular videos?)

Challenges:

- The popularity of a video can change rapidly (demand for a specific video could rapidly increase or decrease over a few hours, or even in minutes or seconds).
- All videos gain views over time, so you need some way to decay popularity over time. But Bloom filters don't support deletion.

Solution Use multiple Bloom filters (the Zebra system). Insert new "popular" videos (determined by some metric e.g. views in the past hour) into the first (i.e. most recent) filter. Every so often, "forget" the oldest filter (i.e. delete the last filter in a list and add a new filter to the front of the list). Check if a video is popular by checking for the video in the union of all of the Bloom filters.

For example, you could use three filters and "forget" the last filter every 150 seconds (this was done for the Yahoo! Video Platform).

5 Results

In the Yahoo! Video Platform, the Zebra system was used to select a cluster of servers to handle a given request (based on factors like popularity and physical locality), and the SPOCA algo-

rithm was used to determine which server among the cluster would serve the request (potentially from its cache).

This system decreased cache miss percentage from 9.7% to 0.4% over about two weeks, saving $350 million in equipment costs alone over five years (power costs were also reduced significantly).

References

[1] Ashish Chawla, Benjamin Reed, Karl Juhnke, and Ghousuddin Syed. 2011. Semantics of caching with SPOCA: a stateless, proportional, optimally-consistent addressing algorithm. In Proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIXATC'11). USENIX Association, Berkeley, CA, USA, 33-33.