COMP 480/580 — Probabilistic Algorithms and Data Structure Jan 24, 2019

Lecture 6

Lecturer: Anshumali Shrivastava *Scribe By: Gaurav Gupta*

1 Web Caching: Motivation

Suppose we wish to serve websites in response to requests. Some websites are popular and are requested very frequently, while others are not. Instead of loading every website from the server, we wish to identify popular websites and store (or cache) them locally to avoid web traffic and communication overhead. If we recieve many repeat requests for a particular page, then we want to store this page in our cache memory. Each time we store a webpage, we also want to keep a signature of each component so that when the HTTP server updates the webpage, we can update the cache. This type of system will save resources because the local server cache can be used by many users. This arrangement is called web caching. As an example, suppose many users from the Rice University network (or perhaps other users in Houston) visit the same page on amazon.com. This page will be loaded into a web cache located in Houston rather than loaded directly from Amazon each time.

In short, web caches result in **less web traffic, less congestion, less communication, and fewer dropped packets.**

However, there are obvious challenges. Remembering the recently-accessed Web pages for a large number of users will require a fast storage system, with efficient retrieval. The way to efficiently implement such a system is using a shared, distributed cache implemented at a large scale. That is, we need a performant cache that we can spread over multiple machines. This is precisely the solution provided by Akamai, the first company to implement large scale web caching. This also leads us to the next question of how to spread the web pages over a given number of machines.

We can use hashing to arrive at a simple solution for this problem. We use the webpage address as the key, we keep a reference to each machine as the value, and we set the hash table size equal to the number of machines. For a webpage, say amazon.com, and a hash function h, we store amazon.com using the machine associated with the hash table index $h(\text{amazon.com})$. Unfortunately, this system is not efficient if a server crashes or if we need to add more servers. Server crashes occur quite frequently and, and we regularly need to add new servers. Therefore, our simple solution will not work.

One obvious for this problem is to change the hash function and reallocate all the keys again. For example: say $h(x) = x \bmod 12$ is the hash function, addition or deletion of one machine changes it to $x \bmod 13$ or $x \bmod 11$. Unfortunately, the reallocation takes $O(n)$ time which is infeasible. We will show that there is a better solution for this problem.

Problem: Find a hashing scheme such that we require the minimum number of changes (reallocations) if we resize the table.

2 Consistent Hashing

The problem can be solved by hashing webpages not to machine IDs themselves but to a variable-size space between the machine IDs. More formally, consider a hashing scheme that hashes both web pages and machine IDs into the same range defined on a circular table. Refer to figure 1 for a visual. Each machine accepts web pages whose hash values lie in the white spaces to its left. In other words, to assign an object x to a machine, compute $h_i(x)$ and then traverse the circular table in the clockwise direction until you find the first machine's hash $h_m(y)$. Then assign webpage x to machine y.

However, this system has slow assignment times. The assignment for this system includes a search time (to traverse the circular table searching for a machine). To traverse the table, we need to probe the total number of buckets between two machines. The time complexity of this operation is linear in the number of machines. (It will be of the order of a fraction of the circular table range). A better set of solutions to search/insert items and machines is as follows.

Insert item x: Use Binary Search trees. Put the allocated indices of the servers in a binary search tree. Update the tree as needed. Given $h(x)$, we can find the successor (the next machine in a counterclockwise direction) in $log(n)$ time. If the item has no successor in the BST then return the machine with the smallest h_m value). Store x in the returned machine.

How to insert a new machine y?
Insert machine y: Given $h_m(y)$, find the successor of y in the BST (If it has no successor in the BST then return the machine with the smallest h_m value). Move all items whose value is less than y from the returned machine to the newly inserted machine y.

How to delete an existing machine y?
Delete machine y: Find the successor of y in the BST (if it has no successor in the BST then return the machine with the smallest h_m value). Move all items in y to the returned machine.

How to delete an existing item x?
Delete item x: Find the successor of x in the BST (if it has no successor in the BST then return the machine with the smallest h_m value). Delete x from the returned machine.

3 Load analysis

Given m items (m webpages in case of web-caching) and n machines/servers, the expected load of each machine is m/n. This means that, in expectation, the load is divided uniformly over the machines. When a machine is added or deleted the expected load becomes $\frac{m}{n+1}$ and $\frac{m}{n-1}$ respectively.

3.1 Max load of a machine

Argument: With high probablity, no machine owns more than $O(\frac{log\ n}{n})$ fraction of the total load.

Proof: Assuming the total load is 1 (in other words $m = 1$)
It is equivalent to prove that with probablity less than $\frac{1}{n}$, there exists an interval of size $\frac{2log\ n}{n}$ inside which no machine lands. In this case the load of the machine is more than $\frac{log\ n}{n}$.

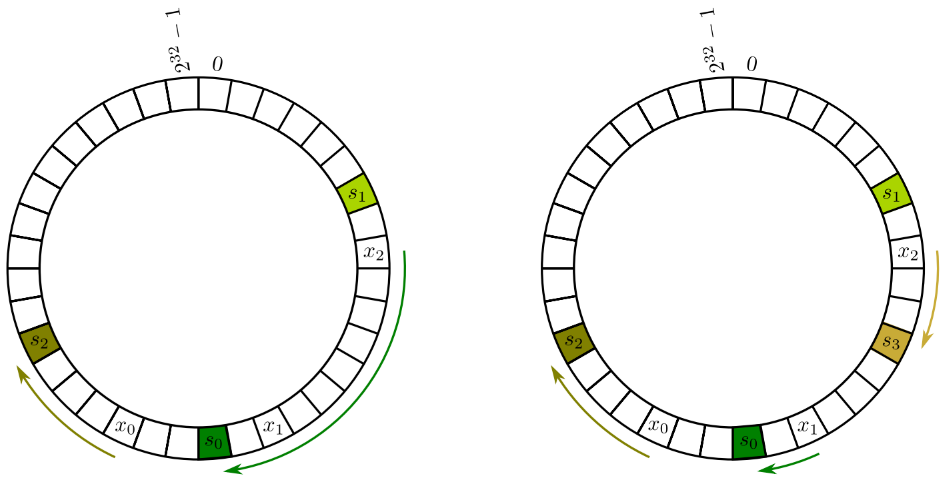

Figure 1: (Left) We use a cyclic table structure. For illustration there are $2^{32}-1$ partitions. The webpages assigned to a machine are those that hash to the range on the anticlockwise side of the machine's hash value. (Right) Inserted $S3$, hence a part of the load is shifted from $S0$ to $S3$. (Refrence: Tim Roughgarden's notes)

For an interval I of size $\frac{2log\ n}{n}$, the probability that no machine lies in this interval is

$$p = (1 - I)^n \tag{1}$$

$$p = (1 - 2\frac{log\ n}{n})^n \approx e^{-2\ log\ n} = \frac{1}{n^2} \tag{2}$$

If there are $k = \frac{n}{2log\ n}$ equal sized disjoint intervals, the probability that no machine lies in any one these intervals is

$$\bigcup_1^k p = \bigcup_1^k \frac{1}{n^2} \tag{3}$$

Using the probability union bound

$$\bigcup_1^k \frac{1}{n^2} \leq \sum_1^k \frac{1}{n^2} \tag{4}$$

$$\sum_1^k \frac{1}{n^2} = \frac{n}{2log\ n} \times \frac{1}{n^2} \leq \frac{1}{n} \tag{5}$$

or, with probbality $\geq 1 - \frac{1}{n}$, every interval of size $\frac{2log\ n}{n}$ contains at least one machine.

Question: Can we also say no machine is underloaded with high probablity?
Answer: No. There is a high probability that at least one interval contains two machines. To prove this, suppose the range is split equally into l intervals, each taking a fraction of $1/l$ of

the load. The probablty that no two of the n machines fall into same interval is

$$1 \times \frac{l-1}{l} \times \frac{l-2}{l} .... \times \frac{l-(n-1)}{l} = \frac{(l-1)!}{(l-n)!l^{n-1}} = \frac{(l)!}{(l-n)!l^{n}} \tag{6}$$

This probablity decreases as n grows. For example, for $l = 10000$ and $n = 100$ the probablity is 0.6085. Hence the probablity of two machines falling in the same interval is 0.3915.

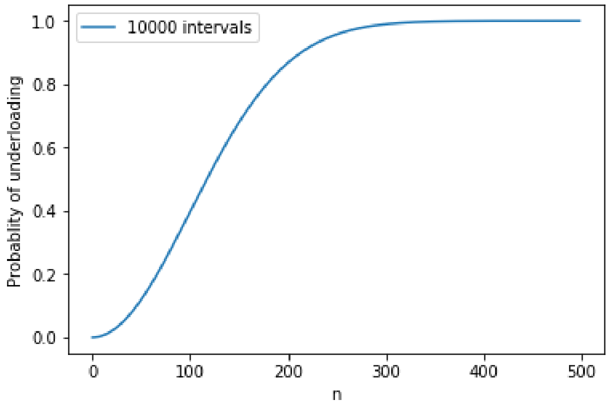

Figure 2: The plot shows how the probablity of underloading varies as we increase n.

This phenomenon is famously known as the **birthday paradox**. In the previous example, if we take $l = 365$ then we get the probability that no two people in a sample of n people share a birthday. This value is much smaller than one would think, even for small n. For instance, take $n = 23$ and observe that this probability is only 0.4927!

3.2 Reduce the variance of the workload

The expected load of a machine is $\frac{m}{n}$ and the max load is $O(m\frac{log\ n}{n})$. We can reduce the variance of workloads by creating multiple copies of each machine and hashing all of the copies.

If we create K copies (note: the hash value of each copy is different) of each machine, then the total load is the sum of K i.i.d random variables. Let $Y_1, Y_2, ...Y_K$ be the loads of all copies of a machine y. The expected load of Y_i is $\frac{m}{Kn}$ and the expected total load of machine y (the sum $Y_1 + Y_2 + ... + Y_K$) remains the same:

$$E(Y) = E(Y_1 + Y_2 + ...Y_K) = \sum_{i=1}^{K} E(Y_i) = K \times \frac{m}{Kn} = \frac{m}{n} \tag{7}$$

However, the variance will decrease. This can be proven using the weak law of large numbers from a previous lecture.

$$P(|\sum_{i=1}^{K} Y_i - E(Y)| > \epsilon) \longrightarrow 0 \tag{8}$$

As k increases, the tail shrinks and the sum of the i.i.d random variable becomes sharply concentrated around the mean. In other words, the variance decreses exponentially (by the Chernoff bound).

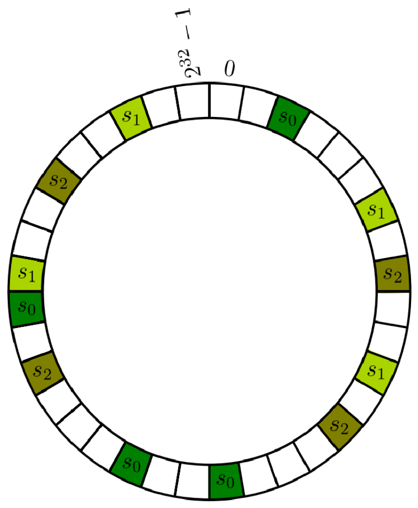

Figure 3: An ilustration of hashing 4 copies of each machine. (Refrence: Tim Roughgarden's notes)