MIT 6.854/18.415: Advanced Algorithms	Spring 2016
Lecture 3 – February 10, 2016	
Prof. Ankur Moitra	*Scribe: Yinzhan Xu*

In this lecture we explain the consistent hashing scheme of [1] and some of its applications. In particular we discuss the application of random trees in consistent hashing and the setting in which clients do not have a consistent view of the web.

1 Consistent Hashing

1.1 Setup

There are m items such that each of them needs to be stored in one of the n distributed web caches. Here we aim to design an efficient hashing scheme that works well even if n changes. To start, recall the 2-universal hash scheme discussed in Lecture 1:

$$\mathcal{H} = \{h_{a,b} | a \in \{1, 2, \ldots, p-1\} \text{ and } b \in \{0, 1, \ldots, p-1\}\},$$

where

$$h_{a,b}(x) = (ax + b \bmod p) \bmod n.$$

Using a 2-universal family of hash functions, we can create a perfect hashing. However, the perfect hashing works well only if the number of available machines/web caches does not change during the process. When the number of web caches increases, there are two possible strategies to modify the hashing scheme:

1. Change the n in $h_{a,b}$ to $n+1$ to get $h'_{a,b}$.

   By doing so, we need to move *almost* all of the items to their new location determined by $h'_{a,b}$ (**Exercise: why?**). If we don't move an item x to its new place, then we cannot find x because $h'_{a,b}(x)$ no longer points to the machine/location in which x is stored.

2. Keep n unchanged and thus all the functions remain the same.

   In this case, the new machine is not in use. As a consequence, the load will not be distributed evenly among all available storage space in the system.

As you observed none of the above strategies satisfies our objectives. In fact we are looking for a strategy that does not incur too many *re-hashing* and in the meantime keep the load of all machines *almost balanced*. An elegant solution to handle this situation introduced in a work of Karger et al. [1].

1.2 Basic Idea

- Each cache is mapped to a random real number in the interval $[0, 1]$.
- Each item is mapped to a random real number in the interval $[0, 1]$.
- Store each item in the first cache on its right. If there is no cache on its right, then store the item in the cache with the smallest number. We can also imagine wrapping around the interval $[0, 1]$ and store each item in the first cache on its right on the circle.

See Figure 2 as an example of the consistent hashing scheme over the interval $[0, 1]$.

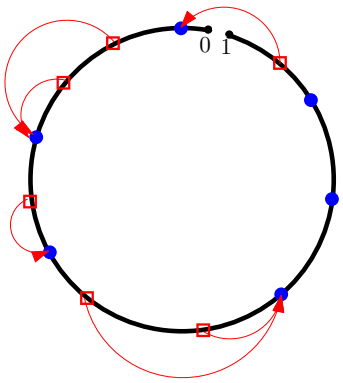

Figure 1: Blue disks denote the machines and red boxes denote the items. Arrows also show the machine to which each item is assigned.

1.3 Implementation

To dynamically maintain the machines and items, we need to maintain a Binary Search Tree (BST), whose keys are the values assigned to the machines. Let h_i and h_m be respectively the functions that we used to hash items and machines to the interval $[0, 1]$.

- **To insert an item x:**
  (a) Find the successor of $h_i(x)$ in the BST (if it has no successor in the BST then return the machine with the smallest h_m value)
  (b) Store x in the returned machine.
- **To delete an item x:**
  (a) Find the successor of $h_i(x)$ in the BST (if it has no successor in the BST then return the machine with the smallest h_m value)
  (b) Delete x in the returned machine.
- **To insert a new machine Y:** There may be some existing items that should be stored in the new machine Y, but these items now are all stored in the successor of $h_m(Y)$ (or the machine with the smallest h_m if $h_m(Y)$ is the largest value).

(a) Find the successor of $h_m(Y)$ in the BST (if it has no successor in the BST then return the machine with the smallest h_m value)

(b) Move all items whose h_i value is less than $h_m(Y)$ to the newly inserted machine Y.

- **To delete an existing machine Y:**

(a) Find the successor of $h_m(Y)$ in the BST (if it has no successor in the BST then return the machine with the smallest h_m value)

(b) Move all items in Y to the returned machine. In Lemma 1 we show that the number of items needed to move is not too much.

1.4 Bounds

Lemma 1. *With high probability, no machine owns more than $O(\frac{\log n}{n})$.*

Proof. Fix some interval I with length $\frac{2\log n}{n}$, then probability that no machine lands in I is

$$\left(1-\frac{2\log n}{n}\right)^n = \left(\left(1-\frac{2\log n}{n}\right)^{\frac{n}{2\log n}}\right)^{2\log n} \approx \frac{1}{n^2}.$$

Equally split $[0,1]$ to $\frac{n}{2\log n}$ such intervals. By union bound, the probability that every one of these intervals contains at least a machine is at least $1-\frac{n}{2\log n}\cdot\frac{1}{n^2} > 1-\frac{1}{n}$. Therefore, with probability at least $1-\frac{1}{n}$, each machine owns an interval of length at most $\frac{4\log n}{n}$. □

However, note that the size of the smallest length can be about $1/n^2$. To see this, split the $[0,1]$ interval into equally n^2 parts, with each part is of length $1/n^2$. By Birthday Paradox, some machines will fall into the same interval with high probability. Thus, with high probability, the size of the smallest interval assigned to a machine is $O(\frac{1}{n^2})$.

Lemma 2. *When a machine is added, the expected number of items that move to the newly added machine is $\frac{m}{n+1}$.*

Proof. The only items that move are those assigned to the new machine. Since the process is as if we map all the machines to the interval from the beginning, by symmetry, in expected $\frac{m}{n+1}$ items assign to the $(n+1)^{\text{th}}$ machine. □

Note that the bound cannot be improved because when $n=1$, all the items should be assigned to the new machine.

2 Random Trees

2.1 Setup

The actual motivation of random trees is to relieve the hotspot on web. Assume that there is a single root server that stores all the pages. If all the requests go to the root server, it cannot handle

all of them and it could crash due to the traffic. Somehow it is necessary to share the load among a set of machines. Therefore the solution is to use a set of proxy caches $\mathcal{C}$ so that the requests can be distributed among them and handled by different machines.

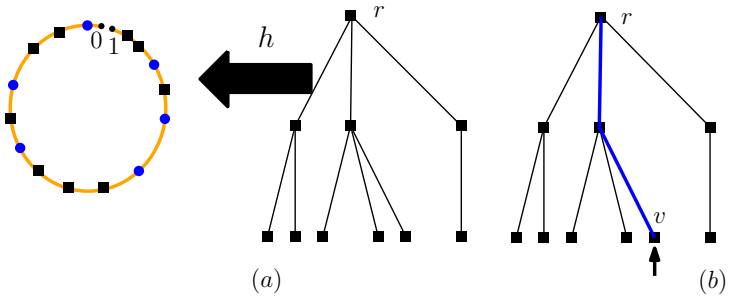

Figure 2: (a) shows a random tree and mapping of its virtual nodes to the interval $[0, 1]$. The blue circles in $[0, 1]$ denote the proxy caches. In (b) a random leaf has been selected to response a request. The leaf-to-root path is shown in blue.

2.2 Implementation

1. Choose a d-ary tree with n virtual nodes V. The root server is located in the root of this d-ary tree. We also choose a consistent hash function $h : V \mapsto \mathcal{C}$ which maps the virtual nodes to the set of available (proxy) caches.

2. For each request of a page, the algorithm does the following:

   (a) Choose a random leaf v in the random d-ary tree (note that the leaf also defines a path to the root node in the tree.

   (b) Ask the virtual nodes in the vr path one by one. For each node u in the path: if the cache $h(u)$ contains the requested page, then return the page; otherwise, propagate the request to the parent of u on the path and increment the page's counter on the local cache (i.e. $h(u)$) that missed the request.

   (c) For any *local cache*, if the counter for a page reaches a fixed threshold denoted by q, then the cache stores the page (note that each node will get the page in its way from r back to the leaf v).

At the beginning all pages are only on the root server. As it goes and by receiving the requests, the popular pages will spread downward in the tree. This propagation of pages among the caches in the tree guarantees that no local cache gets too many requests for any page.

2.3 Analysis

Lemma 3. *Each cache that is not mapped to a leaf on the random tree is asked for the same page at most* $O(dq\frac{\log n}{\log\log n})$ *times with high probability.*

Proof. Consider the children of a non-leaf virtual node v, if all of them have been asked for the same page for at least q times, $h(v)$ will not be asked for this page any more. Therefore, v will only be asked for the same page for at most $O(dq)$ times.

Note that (intuitively) the consistent hashing scheme is similar to the "balls and bins" process and there we showed that if for each ball we draw a bin (uniformly) at random then with high probability the maximum load of the bins is $O(\frac{\log n}{\log \log n})$. In our application, balls translate to the virtual node in the random tree and bins are the proxy caches in the system. Thus by similar argument we can show that with high probability each cache is responsible for at most $O(\frac{\log n}{\log \log n})$ virtual nodes.

Hence, each cache that is not mapped to any leaf nodes with high probability will be asked for the same page $O(dq\frac{\log n}{\log \log n})$ times. □

2.4 Inconsistent World

In reality, not everyone can know the whole set of the proxy caches. Here we modify the model so that it captures what happens in the real world applications. Let $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_\ell$ be ℓ views such that $\mathcal{C}_i \subset \mathcal{C}$ and $\frac{|\mathcal{C}_i|}{|\mathcal{C}|} \geq \frac{1}{t}$ for each i, where t is a fixed constant. Now each view i uses its own consistent hash function $h_i : V \mapsto \mathcal{C}_i$. More precisely, the random tree for each view $\mathcal{C}_i$ is the same, and the underlying hashing of virtual nodes to the interval $[0, 1]$ is the same for all views. The only difference is that $h_i(v)$ returns the first cache in $\mathcal{C}_i$ on the right of the virtual node v (which may not be the same as the first cache in $\mathcal{C}$ which is on the right of v).

However, we can still bound the number of times that a non-leaf node receives a request for a specific page.

Lemma 4. *If there are at most n total requests, then with high probability, each cache is asked for $O(dqt^2 \log^2(n\ell))$ times.*

Proof. Consider a fixed cache c. In part (a), we show that over all views the number of virtual nodes mapped to c is $O(t \log n)$ with high probability. Each virtual node has at most d children and in part (b) we show that over all views each of them is mapped to $O(t \log n)$ different caches with high probability. Finally, since the number of times a cache propagate the request for a page to its parent before storing the page is q, the total number of times that c is asked for any page is bounded by $O(t \log n \cdot d \cdot t \log n \cdot q) = O(dqt^2 \log^2 n)$.

Next we complete the proof by formally proving the bounds on the number of different virtual nodes mapped to a single cache and the number of different caches a single virtual node is mapped to (over all ℓ views).

Proof of part (a)**.** Fix some cache c and let it be mapped to x in the interval $[0, 1]$. Consider the interval I of length $\frac{10t \log n\ell}{n}$ whose right end point is x. Fix some view $\mathcal{C}_i$. The probability that no cache in view $\mathcal{C}_i$ is mapped to a point in I is at most

$$\left(1 - \frac{10t \log(n\ell)}{n}\right)^{|\mathcal{C}_i|} \leq \left(1 - \frac{10t \log(n\ell)}{n}\right)^{\frac{n}{t}} \approx \frac{1}{(n\ell)^{10}}.$$

By union bound over all views, the probability that each of the ℓ views maps at least a cache to a

point in interval I is at least

$$1 - \ell \cdot \frac{1}{(n\ell)^{10}} \geq 1 - \frac{1}{n^{10}}.$$

This implies that the cache c is not responsible for any virtual node outside interval I with probability at least $1 - \frac{1}{n^{10}}$. Then we can use Chernoff bound to conclude that each cache is responsible for at most $O(t \log(n\ell))$ virtual nodes, with high probability (More precisely, using Chernoff bound, we show that the number of virtual nodes that are mapped to an interval of size $\frac{10t \log(n\ell)}{n}$ is $O(t \log(n\ell))$ with high probability).

Proof of part (b)**.** Let j be a virtual node and suppose that it is mapped to x in the interval $[0, 1]$. Consider the interval I of length $\frac{10t \log n\ell}{n}$ whose left end point is x. Similarly to part (a), we can show that with high probability, each view $\mathcal{C}_i$ maps at least one cache to the interval I, so any cache outside interval I is never responsible for virtual node j. Then by Chernoff bound, we can show that each virtual node can only be mapped to $O(t \log(n\ell))$ caches.

□

References

[1] Karger, David, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web." In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, pp. 654-663. ACM, 1997.