These notes were written by Anthony Atkinson (asa15@rice.edu), Yassin Hafid (ymh4@rice.edu), and Atishay Jain (aej7@rice.edu) for the COMP 480 lecture by Dr. Anshumali Shrivastava on 12 September 2023.

1 Background

Consistent hashing is an approach to assigning objects to a handler using hashing that allows for easy insertion and deletion of objects and handlers. It ensures with high probability that one handler will not be overburdened with the objects it must handle relative to the load other handlers bear. Developments in this area have lead to a robust and efficient method for *web caching* - the local storage of internet content on a distributed system for rapid access - saving companies billions of dollars per year.

1.1 History

Web traffic is disproportionate to some websites than to others. For instance, popular websites like `gmail.com` and `amazon.com` are visited more frequently than other one-off websites. Similarly, web traffic is skewed by more popular searches. In order to service these frequent requests quickly, we need a system that identifies them and uses a cache to provide the requested content efficiently for a large number of users. Consider an instance where many users at Rice University are requesting the movie *Die Hard*. We can cache *Die Hard* on the Rice Network, which is more local to the requested location rather than storing it on the original server. Web caching can reduce the number of fetch and update operations, by storing popular websites local to the requester. Reducing latency in these operations is not just for convenience's sake - it can be worth billions of dollars in the case of `yahoo.com` and `google.com`.

Akamai Tech tried to solve this problem, by using a shared cache infrastructure in order to decide which machine would handle a specific request and then provide the response. However, handling network disruptions, such as system downtime or machine failures is difficult with a shared cache system like they used.

1.2 Motivation

Consider an experiment wherein you need to access a website, and there are 100 servers distributed near your location. It's inefficient to poll every server as a potential site since it would take time proportional to the number of servers, causing a severe lag in request service times if the requested content is not found immediately.

If we consider a hash function that maps a hash to each server, problems arise when there is server downtime, a crash (internal) or failure (external), or the addition of new machines. Such hash functions are relatively static, i.e., $\texttt{hashx} = ax + b \bmod R$, where R is the range of the hash values, such that reassigning elements given even this extremely simple function could at best require an easily-computable re-assignment of elements to its new server. Changing this function every time some event happens is infeasible - this would require reallocating every single item cached, of which there could be millions, also of significant storage sizes.

Consistent hashing is the solution to this problem. Instead of hashing webpages to the machines themselves, consider hashing both machines and objects in the same range. To initialize this system, hash with $\texttt{hash}_m$ the n machines into a large circular array. To insert or search for a webpage x, compute an item's hash with $\texttt{hash}_i(x)$ and traverse right in the array until a first machine's hash is located. If not already cached in machine Y, fetch x and cache it in Y. Otherwise, retrieve x directly from the cache at Y.

While this solves the issue of storing webpages for a variable number of machines, this leads to worst-case search and insertion times proportional to the range of the hash values (one unlucky user encounters a system set up where none of the servers are hashed anywhere except at the end of the array). Instead of performing a linear traversal of the circular array for the location of a hashed server, one could use a BST to search for the server that will handle the hashed item. Instead of $O(R)$ time complexity, we can sharply improve this by using BSTs for $O(\log n)$ worst-case time complexity.

2 How Consistent Hashing Works

Each machine is expected to bear a load of $\frac{m}{n}$ objects, where m is the total number of objects to be inserted/requested/cached and n is the total number of servers. This can be easily understood by the fact that there is an equal probability of an object being hashed into the domain of values handled by any server - not one server is favored in the hashing of objects to a location in the array. This assumes homogeneous capacities for each server.

Adding a machine does not change this expectation: the number of machines increases, so the number of objects that each must handle decreases. Expected load is now $\frac{m}{n+1}$.

In §3, we will show that, with high probability, no machines will bear more than a fraction $O(\frac{\log n}{n})$ of the total load.

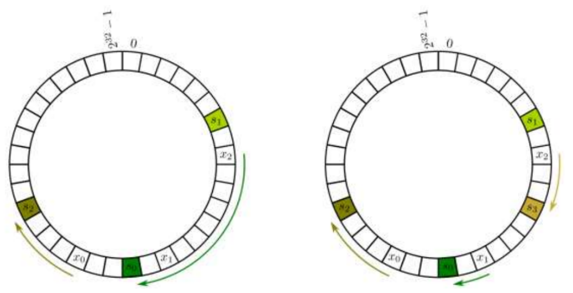

Figure 1: Consistent Hashing into a Circular Array. The s_i are the servers; the x_i are the items. Figure by Tim Roughgarden.

2.1 BST

In order to improve the search latency of consistent hashing, we can use Binary Search Trees (BSTs). The following describe implementations of BSTs used in consistent hashing:

- `insert(object o)` Compute the closest node values to $\texttt{hash}_i(o)$ in the BST. This can be done by traversing through the tree, comparing each node to $\texttt{hash}_i(o)$, and going to the right child (if $\texttt{hash}_i(o)$ is larger) or the left child (if $\texttt{hash}_i(o)$ is smaller). A global variable can be used to keep track of the closest node and can be returned. In the worst case, the algorithm traverses through a branch, with $O(\log n)$ time, where n is the number of machines (size of the tree).

- `insert(server s)` Perform a BST insertion with server s using its hash value $\texttt{hash}_s(s)$. Nodes are inserted at leaves (end nodes) in a BST, with $O(\log n)$ time in the worst case, n is the number of machines.

- `delete(object o)` Calculate the successor of $\texttt{hash}_i(o)$ in the BST. If there exists no successor, return machine with the smallest hash value. Delete e from the machine. Finding the successor is $O(\log n)$ time, where n is the number of machines.

- `delete(server s)` Perform a BST deletion, by finding the successor of the element of the server, and removing the pointer from it to the machine. Finding the successor is $O(\log n)$ time, where n is the number of machines.

All operations with regard to this implementation are $O(\log n)$. This is better than traversing the m buckets clockwise because in the worst case, it could take $O(m)$ time. In sparser arrays, m is much larger than n, and $n \leq m$.

2.2 Consequences of Inserting Servers

Initializing a new machine in consistent hashing can be done one of two ways:

- The new machine fills an empty cache with cache misses as they come in.
- The new machine will copy elements that belong to it from the previous server.

The first method is preferable since it does keep track of items internally nor copy data over the network. Instead, the pages that were once cached will have to be re-cached in their new assigned server when the requests cause a cache miss.

2.3 Consequences of Deleting Servers

There is potential for a cascading problem when servers are deleted. Suppose a server is removed or fails. The elements that it handled are now assigned to another server. The load that this new server must bear has increased, potentially significantly. If this server crashes, then both its load and that of the previous server are distributed onto another server, which will also likely fail from being overloaded, this time with an even greater load to pass on. This continues until many or potentially all servers are overloaded and the whole system must shut down. This is especially a problem with servers of heterogeneous capacities and will be explored in subsequent lectures.

3 Is Consistent Hashing Efficient?

To keep up with the volatile dynamics of real-world systems, i.e., frequent server shutdowns and reboots due to either internal or external factors, we need to ensure that consistent hashing treats each machine "fairly". That is, we desire that our hash function does not overburden any one machine with too many objects to store in proportion to its capacity. As we discuss in following lectures, the vanilla consistent hashing can mechanism can be modified to proportionately handle servers of heterogeneous capacities. We assume for now that each server has the same capacity.

The probability that one of the $1/f$ subsets of elements C_j does not contain one of the n servers s_i to handle or cache any of its elements is $\frac{1}{n}$ if we assume each subset contains $\frac{2\log n}{n}$. The complement of this statement is what we desire: with probability $P \geq 1 - \frac{1}{n}$, every array subset of size $\frac{2\log n}{n}$ contains at least one server to handle requests to its objects. The maximum load that a given server will bear, with the same probability as the above statement, is double the fraction of elements contained within a collection: $\frac{4\log n}{n}$. A proof of these claims follows.

However, we cannot claim that no machine will be underburdened. By the same line of reasoning as the birthday paradox, there is a high probability that two machines will actually fall in the same bin.

Claim: Consistent Hashing Does Not Overburden

We claim, *with high probability, that no machines own more than* $O\left(\frac{\log n}{n}\right)$ *fraction of the load/set of items*

We can show this by choosing some collection of elements C_j of the hash array that constitutes a fraction f of the full range R. There are n servers s_i we insert into the hash array by using an identifier.

$$P(\texttt{hash}(\mathrm{s_i}) \notin C_j) = 1 - f \implies P((\texttt{hash}(\mathrm{s_i}) \notin C_j)\forall i) = (1-f)^n$$

There are $1/f$ disjoint identical collections in this hash array, thus there are $1/f$ possible configurations of the system such that any one of these collections is the one not hashed into by all of the servers. We choose the fraction f of elements held in this collection $f = \frac{2\log n}{n}$:

$$\begin{aligned}
\frac{1}{f}(1-f)^n &= \frac{n}{2\log n}\cdot\left(1-\frac{2\log n}{n}\right)^n \\
&= \lim_{n>>1}\frac{n}{2\log n}\cdot\left(1-\frac{2\log n}{n}\right)^{\frac{n}{2\log n}\cdot 2\log n} \\
&\approx \frac{n}{2\log n}\cdot e^{-1\cdot 2\log n} \\
&= \frac{n}{2\log n}\cdot\frac{1}{n^2} \\
&= \frac{1}{2n\log n} \\
&\leq \frac{1}{n}
\end{aligned}$$

We can make the assumption that n is very large since we explicitly desire that for our hash function.

$$P((\texttt{hash}(\mathrm{s_i}) \notin C_j)\forall i, j) \leq \frac{1}{n} \implies P((\texttt{hash}(\mathrm{s_i}) \in C_j)\exists i, j) \geq 1 - \frac{1}{n}$$