Lab 4: Consistent Hashing

Monday, October 1
CompSci 531, Fall 2018
Outline

• Review Hashing

• Motivation: Caching Webpages

• Consistent Hashing
Symbol-table problem

Symbol table $S$ holding $n$ records:

Record $x$:

Operations on $S$:
- $\text{INSERT}(S, x)$
- $\text{DELETE}(S, x)$
- $\text{SEARCH}(S, k)$

How should the data structure $S$ be organized?
Direct-access table

**Idea:** Suppose that the keys are drawn from the set \( U \subseteq \{0, 1, \ldots, m-1\} \), and keys are distinct. Set up an array \( T[0 \ldots m-1] \):

\[
T[k] = \begin{cases} 
  x & \text{if } x \in K \text{ and } key[x] = k, \\
  \text{NIL} & \text{otherwise.}
\end{cases}
\]

Then, operations take \( \Theta(1) \) time.

**Problem:** The range of keys can be large:
- 64-bit numbers (which represent 18,446,744,073,709,551,616 different keys),
- character strings (even larger!).
Hash functions

Solution: Use a hash function $h$ to map the universe $U$ of all keys into \{0, 1, \ldots, m-1\}:

When a record to be inserted maps to an already occupied slot in $T$, a collision occurs.
Average-case analysis of chaining

We make the assumption of \textit{simple uniform hashing}:

- Each key $k \in S$ is equally likely to be hashed to any slot of table $T$, independent of where other keys are hashed.

Let $n$ be the number of keys in the table, and let $m$ be the number of slots.

Define the \textit{load factor} of $T$ to be

$$\alpha = \frac{n}{m}$$

= average number of keys per slot.
Outline

• Review Hashing

• Motivation: Caching Webpages

• Consistent Hashing
Caching Webpages

• The usual model:
Caching Webpages

• Reality:
Caching Webpages

Can we cut out the server bottleneck?
Caching Webpages

• The usual model:

Just cache this locally.
Caching Webpages – Advantages

• Users get much faster response times from webpages.

• Overall network congestion is decreased.

• Server load is decreased.

• It’s a win win!
  • Well…except that it costs space. Maybe too much for one device.
Caching Webpages

• Better yet, couldn’t *multiple* users/devices share a common cache of recent urls?

• **Problem:** Who stores what? When we try to visit google.com, how do we know which device in our local network has the page in cache?

• **Solution:** Hashing!
Caching Webpages

Set $h(x)$ to be something like $\text{MD5}(x) \mod n$

The expected load on any machine will just be $m/n$, if there are $m$ webpages cached.
Caching Webpages

• **Problem:** What happens if we add or take away a device from this caching scheme?

• We could just set $h(x)$ to be something like $\text{MD5}(x) \mod (n+1)$.

• But then we have to move almost all $m$ cached pages between devices.

• For a problem at this scale on the internet, devices can come and go too often for this to be even remotely feasible.
Outline

• Review Hashing

• Motivation: Caching Webpages

• Consistent Hashing
Consistent Hashing

• We want a way to increase or decrease the number of “buckets” in our hash table without needing to shuffle a lot of data.

• **Key idea:** Don’t hash to machines directly. Hash to values, and also hash the names of the machines.

• To lookup a page, find the active machine whose hash value is closest (to the right) to the hash value of the page.
Consistent Hashing

Figure 2: Each element of the array above is a bucket of the hash table. Each object is assigned to the first cache server on its right.

Figure 3: (Left) We glue 0 and 2 together, so that objects are in a bucket assigned to the cache server that is closest in the clockwise direction. This solves the problem of the last object being to the right of the last cache. (Right) Adding a new cache server. Object $x_2$ moves from $s_0$ to $s_3$.

The key idea is: in addition to hashing the names of all objects (URLs) like before, we also hash the names of all the cache servers. The object and cache names need to be hashed to the same range, such as 32-bit values.

To understand which objects are assigned to which caches, consider the array shown in Figure 2, indexed by the possible hash values. (This array might be very big and it exists only in our minds; we'll discuss the actual implementation shortly.) Imagine that we've already hashed all the cache server names and made a note of them in the corresponding buckets. Given an object $x$ that hashes to the bucket $h(x)$, we scan buckets to the right until we find a bucket $h(s)$ to which an object hashes. (We wrap around the array, if need be.) We then designate $s$ as the cache responsible for the object.

This approach to consistent hashing can also be visualized on a circle, with points on the circle corresponding to the possible hash values (Figure 3(left)). Caches and objects both hash to points on this circle; an object is stored on the cache server that is closest in the clockwise direction. Thus $n$ caches partition the circle into $n$ segments, with each cache responsible for all objects in one of these segments.

This simple idea leads to some nice properties. First, assuming reasonable hash functions,
Consistent Hashing

Figure 2: Each element of the array above is a bucket of the hash table. Each object \( x \) is assigned to the first cache server \( s \) on its right.

Figure 3: (Left) We glue 0 and 232 together, so that objects are in a cache server that is closest in the clockwise direction. This solves the problem of the last object being to the right of the last cache. (Right) Adding a new cache server \( s_3 \). Object \( x_2 \) moves from \( s_0 \) to \( s_3 \).

The key idea is: in addition to hashing the names of all objects (URLs) \( x \), like before, we also hash the names of all the cache servers \( s \). The object and cache names need to be hashed to the same range, such as 32-bit values.

To understand which objects are assigned to which caches, consider the array shown in Figure 2, indexed by the possible hash values. (This array might be very big and it exists only in our minds; we’ll discuss the actual implementation shortly.) Imagine that we’ve already hashed all the cache server names and made a note of them in the corresponding buckets. Given an object \( x \) that hashes to the bucket \( h(x) \), we scan buckets to the right until we find a bucket \( h(s) \) to which some cache name hashes. (We wrap around the array, if need be.) We then designate \( s \) as the cache responsible for the object \( x \).

This approach to consistent hashing can also be visualized on a circle, with points on the circle corresponding to the possible hash values (Figure 3(left)). Caches and objects both hash to points on this circle; an object is stored on the cache server that is closest in the clockwise direction. Thus \( n \) caches partition the circle into \( n \) segments, with each cache responsible for all objects in one of these segments.

This simple idea leads to some nice properties. First, assuming reasonable hash functions,
Consistent Hashing

• With m webpages and n machines, we still have expected load $m/n$ per machine.

• Now we only have to move $m/n$ pages, in expectation, when we add or remove a machine.

• Note that we can even do this in lazy execution!
Consistent Hashing

• **Problem.** How do we actually implement the “find the active machine whose hash value is closest (to the right) to the hash value of the page” idea?

• **Solution.** Talk in small groups for 4 minutes about what you would do.

• Maintain a binary search tree on the machines, sorted by hash values. Then given the hash value of a page, we can find it’s machine in $O(\log(n))$ time, assuming the tree is balanced.
  • Note – one should use a red and black tree, as the tree will be changing frequently and needs to stay balanced.
Consistent Hashing

For example, suppose you want to know to what machine you should cache a page hashed to 5.
Consistent Hashing

• Unfortunately, we have introduced another problem.

• In *expectation*, the load per machine should still be m/n. But expectations aren’t everything...
Consistent Hashing

• These two distributions have the same expectation...

• But different variance. Recall that the variance of a random variable $X$ is $\mathbb{E}[(X - \mathbb{E}[X])^2] = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$. 
Consistent Hashing

• Let $X$ be the random variable for the load on a machine after caching $m$ pages on $n$ machines using our consistent hashing scheme.

• **Problem.** $X$ has substantially higher variance than one would typically expect in hashing applications. Why?

• **Solution.** Talk in groups for 4 minutes about what you would do.
Consistent Hashing

• The standard idea is to create multiple *logical* machines for each physical machine. Everything is as before, except multiple logical machines actually get stored on the same device.

• Another general purpose idea for reducing variance in hashing is to use multiple hash functions.
Consistent Hashing - Akamai

Market Capitalization = 12.3 billion USD
Consistent Hashing - Akamai