Consistent Hashing

Overview

Consistent hashing, introduced in [1] is a hashing technique that assigns items (keys) to buckets in a way that makes it relatively inexpensive to rehash the collection of stored keys when a new hash bucket becomes available (or when a hash bucket needs to be removed).

Ranged Hash Functions

Notation. We use the following notation throughout:

- $\mathcal{I}$: a set of *items* or *hash keys*. Elements of this set $i \in \mathcal{I}$ will be hashed into buckets.
- $I = |\mathcal{I}|$: number of possible hash keys.
- $\mathcal{B}$: a set of *hash buckets*.

  Intuitively, a hash bucket corresponds not just to a logical partition of the hash keys, but also to a specific (physical) location for storing them.
- A subset $\mathcal{V} \subseteq \mathcal{B}$ is called a *view*.

  Intuitively, $\mathcal{B}$ represents a set of all possible locations (e.g., nodes in a computing cluster) available to store hashed data, while a view $\mathcal{V}$ represents the set of currently available locations.

Ranged hash function. A function of the form

$$f : 2^{\mathcal{B}} \times \mathcal{I} \longrightarrow \mathcal{B}$$

is called a *ranged hash function*.

Without loss of generality, we write $f_{\mathcal{V}}(i)$ instead of $f(\mathcal{V}, i)$.

Note: Essentially, for each hash key $i \in \mathcal{I}$ the ranged hash function specifies where it needs to be stored given a view $\mathcal{V}$ of available locations. Because of this, a *ranged hash function is* ***proper*** iff

$$f_{\mathcal{V}}(i) \in \mathcal{V}$$

for all $\mathcal{V} \subseteq \mathcal{B}$ and for all $i \in \mathcal{I}$. From now on, only *proper ranged hash functions are considered.*

Ranged hash function family. A group of ranged hash functions that are organized and behave in a similar manner.

Random ranged hash function. A ranged hash function drawn at random from a particular family $\mathcal{F}$ of ranged hash functions.

Properties of Ranged Hash Functions

General observation. The properties of ranged hash functions specified here allow us to quantify our notion of what "good" hash function behavior is in the presense of the need to scale the hashing procedure up and/or down in the number of available buckets all the time.

Balance. A ranged hash function family is *balanced* if for any view $\mathcal{V}$, a randomly chosen function from this family assigns $O(1/|\mathcal{V}|)$ items to each of the buckets $b \in \mathcal{V}$ with *high probability.*

Note: *balanced* families of ranged hash functions tend to assign the hash keys to different buckets roughly evenly, regardless of the set of buckets currently available.

Monotonicity. A ranged hash function f is *monotone* iff for all views $\mathcal{V}_1, \mathcal{V}_2 \subseteq \mathcal{B}$, if $\mathcal{V}_1 \subseteq \mathcal{V}_2$, then

$$f_{\mathcal{V}_2}(i) \in \mathcal{V}_1 \Longrightarrow f_{\mathcal{V}_1}(i) = f_{\mathcal{V}_2}(i).$$

Note: A ranged hash function is monotone if when new buckets become available, the only reassignments that happen in the function are the reassignments of hash keys to *new buckets.* No reassignment from one *old bucket* to another old bucket may occur.

Spread. Let $\mathcal{V}_1, \ldots \mathcal{V}_K$ be a set of views, such that $|\mathcal{V}_1 \cup \ldots \cup \mathcal{V}_K| = C$, and $|\mathcal{V}_i| \geq C/t$ for all views $\mathcal{V}_i$ and for some constant t.

The *spread* of a ranged hash function f on $\mathcal{V}_1, \ldots, \mathcal{V}_K$ on item $i \in \mathcal{I}$, denoted $\sigma_f(i)$ is

$$\sigma_f(i) = |\{f_{\mathcal{V}_j}(i)\}|, \text{ where } j = 1 \ldots K.$$

Note: Essentially, a spread of a ranged hash function on an item is the total number of different hash buckets it can be assigned to within a specific collection of "reasonable" views. A view is considered "reasonable" if it can see at least $\frac{1}{t}$ fraction of all "operational" buckets.

Load. Given a ranged hash function f and a bucket $b \in \mathcal{B}$, the load $\lambda_f(b)$ on the collection of views $\mathcal{V}_1, \ldots \mathcal{V}_K$ (defined as above) is

$$\lambda_f(b) = |\bigcup_{j=1}^{K} f_{\mathcal{V}_j}^{-1}(b)|.$$

Note: $f_{\mathcal{V}}^{-1}(b)$ is the set of items assigned to bucket b by function f in view $\mathcal{V}$.

What We Want

We want to describe a *family $\mathcal{F}$ of ranged hash functions* with the following properties:

1. $\mathcal{F}$ is **balanced**, i.e., any its function tends to distribute hash keys roughly evenly among the available hash buckets.
2. All functions in $\mathcal{F}$ are **monotone**, i.e., when adding new hash buckets, we would only need to rehash the keys that need to move into the new buckets.
3. $\mathcal{F}$ has a *controlled* **spread**, i.e., hash keys tend to be hashed to reasonably few different buckets in different "reasonable" views.
4. $\mathcal{F}$ has *controlled* **loads**.

Consistent Hashing

We construct a family of ranged hash functions that satisfies the above properties as follows.

Consider two functions,

$$r_{\mathcal{B}} : \mathcal{B} \longrightarrow [0, 1]$$

and

$$r_{\mathcal{I}} : \mathcal{I} \longrightarrow [0, 1]$$

which randomly map buckets and hash keys (respectively) to the unit interval.

We define $f_{\mathcal{V}}(i)$ as follows:

$$f_{\mathcal{V}}(i) = arg \min_{b \in \mathcal{V}}(|r_{\mathcal{B}}(b) - r_{\mathcal{I}}(i)|)$$

Figure 1 shows how the family of consistent ranged hash functions defined above works. We loop the unit interval, and consider the distance function $|r_{\mathcal{B}}(b) - r_{\mathcal{I}}(i)|$ modulo that looping. On the left of Figure 1, a view with four buckets is observed, with $r_{\mathcal{B}}$ mapping the buckets as follows:

bucket	$r_{\mathcal{B}}$
bucket 0	0.83
bucket 1	0.21
bucket 2	0.63
bucket 3	0.425

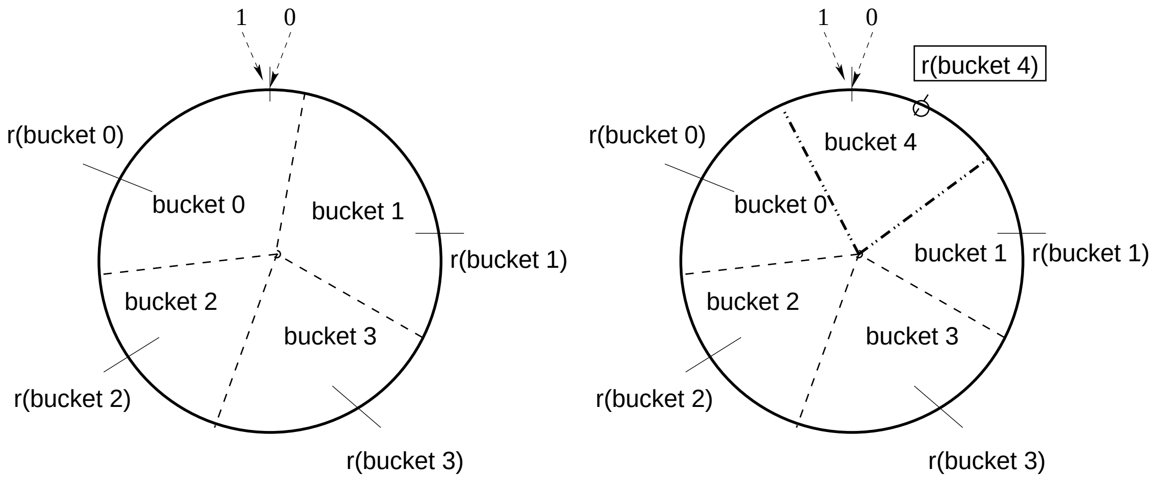

Figure 1: Simple consistent hashing explained. On the left (part a), a ranged hash function on a four-bucket is shown. On the right (part b), a fifth bucket is added to the view. Only the elements that need to be hashed into the new bucket are moved.

The dashed lines show the ranges of $r_{\mathcal{I}}$ that will hash to the specific buckets. The ranges are computed by bisecting each interval on the unit loop.

On the right of Figure 1, we observe what happens when a new bucket is added to the view. The new bucket, bucket 4, is hashed to 0.08. This causes a reassignment of parts of $r_{\mathcal{I}}$ ranges from bucket 0 and bucket 1 to bucket 4. The new boundaries are shown with dash-dotted lines.

We note that through this realingment, the set of hash keys assigned to buckets bucket 2 and bucket 3 remain intact. We also note, that **the only** hash key transfers that occur are from bucket 0 to bucket 4 and bucket 1 to bucket 4: buckets bucket 0 and bucket 1 keep all their hash keys that *do not* shift to bucket 4. Therefore, this reassignment is *monotone*.

Consistent Hashing: Improvements

From the example above, we also see a significant disadvantage of simple consistent hashing:

> When a new bucket is added, hash keys from only two other buckets will be reassigned to it!

This property means that in many situations, addinng new buckets will keep on reassigning the same keys, while keeping other buckets intact. It may also cause temporary disbalance in the key assignment to different buckets.

In order to create a better family of consistent range hash functions, we improve the initial criterion as follows:

- Let $\mathcal{B} = C$. We change $r_{\mathcal{B}}$ to now behave as follows:

$$r_{\mathcal{B}} : \mathcal{B} \longrightarrow [0,1]^{\log(C)}.$$

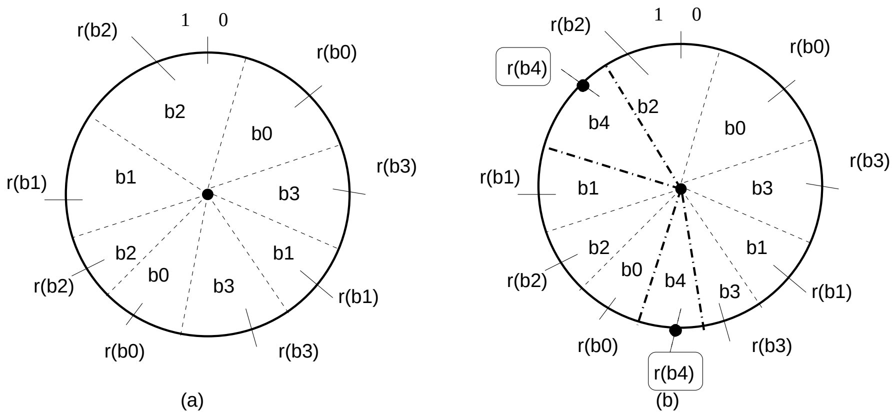

Figure 2: Improved consistent hashing example.

That is, the bucket assignment function now maps each bucket to $\log(C)$ *randomly selected* points on the unit interval.

- The ranged hash function $f_{\mathcal{V}}(i)$ retains is formal definition, except now, it treats each of the $log(C)$ values for a bucket b independently. A hash key i is assigned to bucket b if $r_{\mathcal{I}}(i)$ is *closest* to ***one of the values of*** $r_{\mathcal{B}}(b)$.

The immediate impact of this change is that now, instead of only one interval of hash keys associated with a bucket, the ranged hash function establishes $log(C)$ intervals. When a new bucket is added, a new collection of $log(C)$ values for this bucket is places on the unit interval (unit loop). The probability that more than two buckets will be affected is now high. This means that in general, we expect to transfer *smaller portions* from *a larger number of buckets* to the new bucket.

Figure 2 shows how the new hashing scheme works. $r_{\mathcal{B}}$ now returns two randomly selected values for each bucket[1]. In our example, the function looks roughly as follows:

bucket	$r_{\mathcal{B}}$	
b0	0.16	0.61
b1	0.38	0.76
b2	0.69	0.96
b3	0.26	0.47

Each bucket now controls two regions of hash key values on the unit loop (the second region for `b2` wraps around the 1-0 boundary).

On the right side of Figure 2, we show what happens when a fifth bucket, `b4` is added with $r_{\mathcal{B}}$ values of 0.51 and 0.89 mapped. The two regions for bucket `b4` now split the regions for buckets `b3` and `b0` (region around the 0.51 value) and buckets `b1` and `b2` (region around the 0.89 value). In this specific case, all four original buckets surrender (transfer) some of their keys to `b4`. In some other cases (e.g., if $r_{\mathcal{B}} = \{0.2, 0.51\}$) fewer than all buckets contribute, but the probability of only two buckets contributing is relatively low (in fact, for some functions $r_{\mathcal{B}}$ it will be 0).

[1]Since $\log(C)$ is assumed to be binary logarithm, a more correct example would have to show three values being selected, but we simplify the figure by selecting two values.

In [1], the number of values returned by $r_{\mathcal{B}}$ for a bucket is set to $k\log(C)$ for some constant k.

In [1] the following key theorem is stated and proved for the improved version of consistent hashing functions.

Theorem. The ranged hash function family $\mathcal{F}$ defined above has the following properties:

1. $\mathcal{F}$ is monotone.
2. **Balance:** For a fixed view $\mathcal{V}$,

$$P\left(f_{\mathcal{V}}(i) = b\right) \leq \frac{O(1)}{|\mathcal{V}|},$$

   (for $i \in \mathcal{I}, b \in \mathcal{V}$.
3. **Spread:** if the number of views $V = \rho C$ for some constant ρ, and the number of items $I = C$, then for $i \in \mathcal{I}$, $\sigma(i) \sim O(t\log(C))$ with probability $P > 1 - \frac{1}{C^{\Theta(1)}}$.
4. **Load:** if V and I are as above, then for $b \in \mathcal{B}$ $\lambda(b) \sim O(t\log(C))$ with probability $P > 1 - \frac{1}{C^{\Theta(1)}}$.

Implementation

In [1], a number of implementation details are provided that allow for the computations related to creation and reassignments of keys to be fast.

Mapping of unit loop segments to buckets. Use *balanced binary search tree.* For C buckets, we have $kC\log(C)$ intervals, and the depth of the search tree is $O(log(C))$.

To hash a key $i \in \mathcal{I}$, first we compute $r_{\mathcal{I}}(i)$[2]. Then we search for the interval containing $r_{\mathcal{I}}(i)$ in the binary search tree. Finally, the hash key is sent to the respective bucket.

To insert/remove bucket, $k\log(C)$ values are inserted into/removed from the balanced search tree (each at a cost of $O(\log(C))$) for a total complexity of $O(log^2(C))$.

Improving hash computation to O(1). Instead of keeping a single balanced search tree, we divide the unit interval into $kC\log(C)$ equal length segments, and create a balanced search tree for $r_{\mathcal{B}}(b)$ values for each interval separately.

Hashing in this case consists of two steps:

1. Determine the correct interval for $r_{\mathcal{I}}(i)$. This is done in time $O(1)$, because boundaries of the $kC\log(C)$ intervals are known implicitly (i.e., we can simply do $r_{\mathcal{I}}(i) cdot kC\log(C)$ to determine the interval it is in).

[2] Assumed to be O(1).

2. Look up the correct bucket for the hash value. The expected number of nodes in search tree for each interval is $O(1)$ (we have $O(kC\log(C))$ $r_{\mathcal{B}}$ values and $O(kC\log(C))$ intervals). Therefore, **on average**, each lookup will take $O(1)$ time.

This means that the expected time to compute a hash (amortized over time) is $O(1)$.

To facilitate it, size of subintervals needs to be shrunk every time more buckets are added.

References

[1] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin, Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web, in Proceedings, *Twenty Ninth Annual ACM Symposium on the Theory of Computing (STOC)*, El Paso, TX, USA, May 1997, pp. 654–663.