CS 7800 / 4810 — Advanced Data Structures

Lecture Notes: Consistent Hashing

---

1 Problem Setup

There are m items such that each of them needs to be stored in one of n distributed web caches.

2 Recall: Hash Functions

2.1 Universal Hashing

We are given a set $\mathcal{H}$ of hash functions mapping from $\mathcal{U} \to \{0, 1, 2, \ldots, m-1\}$ such that $\forall\, x, y \in \mathcal{U}$, where $x \neq y$:

$$|\{h_a \in \mathcal{H} \mid h_a(x) = h_a(y)\}| = \frac{|\mathcal{H}|}{m}.$$

That is, the probability that x and y collide is $\frac{1}{m}$, if we choose h_a uniformly at random from $\mathcal{H}$.

2.2 2-Wise Independent Hashing

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

where

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

Using a 2-wise independent family of hash functions, we can create a perfect hashing scheme.

3 Motivation for Consistent Hashing

Perfect hashing only works well if the number of machines does not change during the process. If the number of machines changes, we face two bad options:

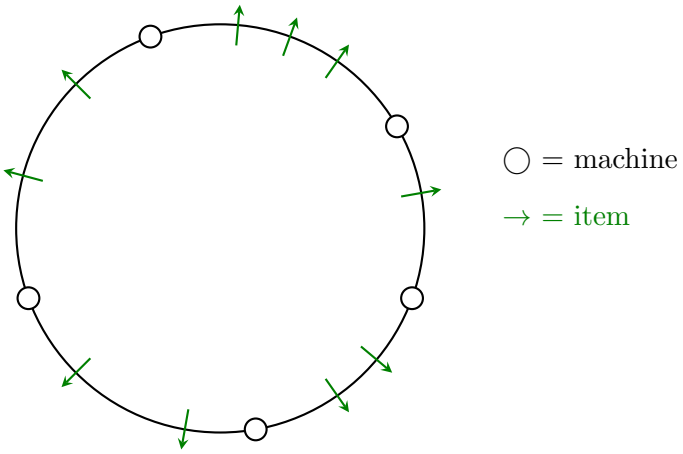
1. **Change n to n':** Update the modulus in $h_{a,b}$ from n to n' to get $h'_{a,b}$. By doing so, we need to move *almost all* items to their new locations.
2. **Keep n unchanged:** No items need to move, but the new machine is not used. This creates load imbalance.

Key Idea

We need a strategy that does not incur too many re-hashings and, in the meantime, keeps the load of all machines almost balanced.

4 Consistent Hashing: Basic Idea

- Each machine 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 machine on its right (clockwise on the ring). If no cache is on the right, then store the item in the cache with the smallest number (i.e., wrap around).



5 Implementation

To dynamically maintain 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 use to hash items and machines to the interval $[0, 1)$.

5.1 Insert an Item x

1. Find the successor of $h_i(x)$ in the BST.
2. If no successor, then find the smallest h_m value.
3. Store x in the returned machine.

5.2 Delete an Item x

1. Find the successor of $h_i(x)$ in the BST.
2. If no successor, then find the smallest h_m value.
3. Delete x in the returned machine.

5.3 Insert a New Machine Y

There may be some existing items that should be stored in the new machine Y, but these items are all currently stored in the successor of $h_m(Y)$.

1. Find the successor of $h_m(Y)$ in the BST.
2. Move all items whose h_i value is less than $h_m(Y)$ to the newly inserted machine Y.

5.4 Delete an Existing Machine Y

1. Find the successor of $h_m(Y)$ in the BST.
2. Move all items in Y to the returned machine.

6 Bounds

Lemma 6.1. *With high probability, no machine owns more than* $O\left(\frac{\log n}{n}\right)$ *of the interval* $[0, 1)$.

Proof. Consider a particular machine M and the interval it owns (i.e., the arc from the preceding machine to M on the ring). This interval is large only if no other machine landed in that region.

Fix an interval I of length $\ell = \frac{c \log n}{n}$ for a constant $c > 0$ to be chosen later. Each of the remaining $n - 1$ machines lands in I independently with probability ℓ. The probability that *none* of them lands in I is

$$(1-\ell)^{n-1} \le e^{-\ell(n-1)} = e^{-\frac{c\log n}{n}\cdot(n-1)} \le e^{-(c-o(1))\log n} = \frac{1}{n^{c-o(1)}}.$$

There are n machines, so by a union bound the probability that *any* machine owns an interval of length $\ge \ell$ is at most

$$n \cdot \frac{1}{n^{c-o(1)}} = \frac{1}{n^{c-1-o(1)}}.$$

Choosing c to be a sufficiently large constant (e.g., $c = 3$) makes this probability $O(1/n)$. Therefore, with high probability every machine owns an interval of length at most $O\left(\frac{\log n}{n}\right)$. □

Lemma 6.2. *With high probability, the size of the smallest interval assigned to a machine is* $O\left(\frac{1}{n^2}\right)$.

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

$$\Pr[\text{no machine lands in } I] = \left(1 - \frac{2\log n}{n}\right)^n = \left(\left(1 - \frac{2\log n}{n}\right)^{\frac{n}{2\log n}}\right)^{2\log n} \approx \frac{1}{n^2}.$$

Equally split $[0, 1)$ into $\frac{n}{2\log n}$ such intervals. By a union bound:

$$\Pr[\text{every interval contains at least 1 machine}] \ge 1 - \frac{n}{2\log n}\cdot\frac{1}{n^2} > 1 - \frac{1}{n}.$$

Therefore, with high probability, each machine owns an interval of length at most $\frac{4\log n}{n}$. □

Lemma 6.3. *When a machine is added, the expected number of items that move to the newly added machine is*

$$\frac{m}{n+1}.$$

Proof. Suppose there are currently n machines on the ring and a new machine Y is inserted at position $h_m(Y)$. By the symmetry of consistent hashing, the new machine's position is uniformly random on $[0, 1)$ and is independent of the existing n machine positions.

After insertion, the ring is partitioned into $n + 1$ intervals by the $n + 1$ machines. The interval owned by Y is the arc from the machine immediately preceding Y (its predecessor on the ring) to Y itself. By symmetry of the $n + 1$ uniformly random positions, the expected length of any single machine's interval is $\dfrac{1}{n+1}$.

Each of the m items is hashed to a uniformly random position on $[0, 1)$ independently. An item moves to Y if and only if it falls in the interval now owned by Y. Since this interval has expected length $\dfrac{1}{n+1}$, by linearity of expectation:

$$\mathbb{E}[\text{items moved}] = m \cdot \mathbb{E}[\text{length of } Y\text{'s interval}] = \frac{m}{n+1}.$$

Note that before the insertion of Y, all of these items were stored in the successor of $h_m(Y)$, which is the only machine from which items need to be redistributed. □

7 Random Trees

7.1 Motivation

The actual motivation of random trees is to relieve the *hot spots* on the web. If only a root server is handling all the requests, it will quickly become the bottleneck. The goal is to use a set of proxy caches so that requests can be distributed among them.

7.2 Concrete Use Case: Content Distribution Networks

Consider a popular news website (e.g., the New York Times) that hosts millions of pages. On a normal day, traffic is spread across many articles. But when a major event breaks — say, an election result or a natural disaster — a single page may suddenly receive millions of requests per second.

Naïve approach (single origin server): Every request goes to the origin. The server is overwhelmed and the page becomes unavailable precisely when demand is highest.

Flat consistent hashing: We distribute pages across n caches using the ring. This balances load across pages, but it does not help when one page is *far more popular* than others. The single cache responsible for the hot page still becomes a bottleneck.

The Hot-Spot Problem

Consistent hashing balances load *across pages*, but not *within a single hot page*. We need a mechanism that **replicates popular content** across multiple caches automatically and proportionally to demand.

Random trees solve this by organizing the caches into a logical tree and letting popular pages "trickle down" toward the leaves as demand grows. Concretely:

1. A client requesting a page enters the tree at a random leaf.
2. The request walks *up* toward the root until some cache on the path has the page.

3. Each cache along the way counts how often the page is requested. Once a threshold q is reached, the cache stores a local copy.

The effect is adaptive replication: a page requested once stays only at the root; a page requested thousands of times is replicated across many caches near the leaves, spreading the load. Crucially, unpopular pages consume *no* extra storage, and the system adapts dynamically as popularity shifts.

7.3 Implementation

- Choose a d-ary tree with n virtual nodes V.
- The root server is located at the root of this d-ary tree.
- We choose a consistent hash function $h\colon V \to C$ (mapping virtual nodes to caches).

7.4 Request Protocol

For each request of a page:

1. Choose a random leaf v in the random d-ary tree.
2. Walk the v-to-root path one node at a time.
3. For each node u in the path:
   - If the cache $h(u)$ contains the requested page, return the page.
   - Otherwise, go to the parent of u on the v-to-root path and increment the page's counter on the local cache (i.e., $h(u)$) that missed the request.
4. For any local cache, if the counter for a page reaches a fixed threshold denoted by q, then the cache stores the page.

Remark 7.1. At the beginning, all pages are only on the root server. As requests arrive, the popular pages will spread downward in the tree. This guarantees that no local cache gets too many requests for any page.

Lemma 7.1. *Each cache that is not mapped to a leaf on the random tree is asked for the same page at most*

$$O\left(dq \cdot \frac{\log n}{\log \log n}\right)$$

times, with high probability.

8 Space Lower Bound for Approximate Membership (Filters)

We now turn to a fundamental question in the design of space-efficient data structures: how much space is needed to represent a set *approximately*? The following lower bound, due to Carter, Floyd, Gill, Markowsky, and Wegman [**?**], shows that any filter must use at least $n \log_2(1/\varepsilon)$ bits.

8.1 Problem Setup

Definition 8.1 (Approximate Membership / Filter)**.** Let U be a finite universe with $|U| = u$ and let $S \subseteq U$ with $|S| = n$. An *approximate membership tester* (filter) with false-positive rate ε is a data structure D that implements a function $\text{query}_D : U \to \{0, 1\}$ satisfying:

1. **No false negatives:** For all $x \in S$, $\text{query}_D(x) = 1$.
2. **Bounded false positives:** For a uniformly random $y \in U \setminus S$, $\Pr[\text{query}_D(y) = 1] \leq \varepsilon$.

The question is: how many bits must D use to achieve false-positive rate ε for a set of n elements?

8.2 Information-Theoretic Lower Bound

Theorem 8.1 (Carter et al., 1978)**.** *Any (static) filter storing a set S of n elements from a universe U with false-positive rate ε must use at least*

$$n \log_2 \left(\frac{1}{\varepsilon} \right)$$

bits in expectation (over the random choice of S), assuming $u \gg n/\varepsilon$.

Proof. The argument is a counting/information-theoretic one. We count the number of *distinct behaviors* a filter can exhibit, and show that if the filter uses too few bits, there are not enough distinct representations to handle all possible input sets.

Step 1: Counting the effective false-positive sets. Fix a filter representation D. The behavior of D is completely determined by the set

$$F_D = \{x \in U \mid \text{query}_D(x) = 1\}.$$

Since D has no false negatives, we must have $S \subseteq F_D$. The false-positive rate constraint says that the number of non-members accepted is at most $\varepsilon(u - n)$ in expectation, so

$$|F_D| \leq n + \varepsilon(u - n) \leq n + \varepsilon u.$$

For a fixed set F_D of this size, how many input sets S of size n can "hide" inside F_D? At most

$$\binom{|F_D|}{n} \leq \binom{n + \varepsilon u}{n}.$$

Step 2: Counting the number of possible input sets. The total number of possible input sets of size n from U is $\binom{u}{n}$.

Step 3: Pigeonhole argument. If the filter uses b bits, there are at most 2^b distinct representations. Each representation corresponds to some F_D, and each F_D can "cover" at most $\binom{n+\varepsilon u}{n}$ input sets. Therefore, to cover all possible input sets:

$$2^b \cdot \binom{n + \varepsilon u}{n} \geq \binom{u}{n}.$$

Rearranging:

$$b \geq \log_2 \frac{\binom{u}{n}}{\binom{n+\varepsilon u}{n}}.$$

Step 4: Simplifying the bound. Using the standard estimate $\binom{N}{k} \approx (N/k)^k$ for $N \gg k$:

$$\frac{\binom{u}{n}}{\binom{n+\varepsilon u}{n}} \approx \frac{(u/n)^n}{((n+\varepsilon u)/n)^n} = \left(\frac{u}{n+\varepsilon u}\right)^n.$$

When $u \gg n/\varepsilon$ (i.e., the universe is much larger than the set), we have $n + \varepsilon u \approx \varepsilon u$, so

$$\left(\frac{u}{\varepsilon u}\right)^n = \left(\frac{1}{\varepsilon}\right)^n.$$

Therefore:

$$b \geq n \log_2\left(\frac{1}{\varepsilon}\right).$$

□

Remark 8.1. This bound is *tight* for static filters: there exist constructions (e.g., using perfect hashing to store $\lceil \log_2(1/\varepsilon) \rceil$-bit fingerprints) that achieve $(1 + o(1))\, n \log_2(1/\varepsilon)$ bits.

8.3 Implications

The $n \log(1/\varepsilon)$ Lower Bound

Every bit of false-positive rate reduction costs exactly 1 bit per element. Halving ε requires storing one additional bit per element — regardless of the data structure used.

To put this in perspective:

- A **Bloom filter** with optimal parameters uses $\approx 1.44\, n \log_2(1/\varepsilon)$ bits — about 44% above the lower bound.
- **Quotient filters**, **cuckoo filters**, and **XOR filters** also store fingerprints of $\lceil \log_2(1/\varepsilon) \rceil$ bits and come closer to the bound (within $1 + O(1/\log(1/\varepsilon))$ factor).
- For **dynamic** filters (supporting insertions and deletions), Lovett and Porat (2010) and Kuszmaul and Walzer (2024) showed an additional $\Omega(n)$-bit overhead is unavoidable beyond the static lower bound, i.e., any dynamic filter requires $n \log_2(1/\varepsilon) + \Omega(n)$ bits.

8.4 Connection to Exact Membership

It is instructive to compare with the exact case ($\varepsilon = 0$). An exact membership tester (dictionary) must distinguish all $\binom{u}{n}$ possible sets, so it requires at least $\log_2 \binom{u}{n} \approx n \log_2(u/n)$ bits. When $\varepsilon > 0$, multiple input sets map to the same filter representation (since the filter is allowed to accept some non-members), and the lower bound drops from $n \log_2(u/n)$ to $n \log_2(1/\varepsilon)$ — completely independent of the universe size u.

References

[1] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman. Exact and approximate membership testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC)*, pages 59–65, 1978.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] S. Lovett and E. Porat. A space lower bound for dynamic approximate membership data structures. *SIAM Journal on Computing*, 42(6):2182–2196, 2013.

[4] W. Kuszmaul and S. Walzer. Space lower bounds for dynamic filters and value-dynamic retrieval. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1153–1164, 2024.