Hashing and sketching

1 The age of big data

An age of big data is upon us, brought on by a combination of:

- *Pervasive sensing*: so much of what goes on in our lives and in the world at large is now digitally recorded.
- *Good algorithms* for dealing with data on this scale.

The algorithmic foundations of this area are still being built, and many of the most effective and widely-used methods were developed quite recently, in the past decade or two.

2 Storing data for fast lookup

How can large databases—such as the medical records of all patients under a particular provider, or the web pages crawled by a search engine, or the profiles of a social network's users—be stored so as to allow records to quickly be looked up, added, and deleted?

The formal setup is that each data item lies in some universe U that might be enormous, for instance, $\{$all possible social security numbers$\}$ or $\{$all possible names$\}$ or $\{$all valid URLs$\}$. We have a large collection of data from this universe, say $x_1, \ldots, x_n \in U$. We would like to store these in such a way that:

1. The amount of space used is $O(n)$, the best possible.
2. Some basic operations can be performed quickly, ideally in $O(1)$ time:
   - `lookup`(x): return record for x, if present
   - `insert`(x): add a new entry associated with x
   - `delete`(x): remove the entry associated with x

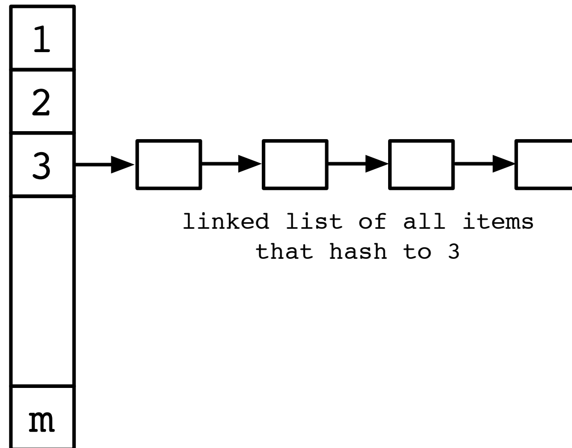
A traditional data structure for this problem is the *binary search tree*, in which the data is stored in lexicographic order in a tree structure of depth $O(\log n)$. Using clever ideas like the "red-black rule" for keeping the tree balanced, all lookups, inserts, and deletes can be handled in $O(\log n)$ time. Is there a faster—and perhaps even easier—way?

2.1 Hashing with chaining

The idea here is to use a table of size $m = O(n)$, call it $T[1 \cdots m]$, and to use a *hash function*

$$h : U \to [m]$$

that determines where each item gets stored. Item $x \in U$, for instance, ends up in $T[h(x)]$, which we describe by saying that it *hashes to bucket* $h(x)$. Now, it is quite possible that many of our data points land in the same bucket, so each entry $T[i]$ actually points to a linked list of items that hash to i.



The total amount of space used is $O(n)$, as required. Lookups, inserts, and deletes are very easy and, for an element x, take time proportional to the length of the linked list at $T[h(x)]$, which we will sometimes call the size of x's bucket. What we need, therefore, is a hash function that spreads items out so that none of the buckets is too heavily loaded. How can we pick such a function?

2.2 Picking a hash function

Say we are hashing people's names. An easy hash function is simply the very first character, which has 26 possible values. For a larger hash table, we could use two characters (size 676), or three (size 17576), or more. This scheme is convenient but is likely to produce very imbalanced buckets. How many names can you think of that begin with `xx`, for instance?

With care, one should be able to handcraft a much better hash function. But this will require advance knowledge of the distribution of words. This is because for any specific function that maps a large space U of names into a table of a fixed size $m = O(n)$, there is some collection of n names for which the function performs abysmally—for instance, sending all of them to the same bucket. This is just the pigeonhole principle at work. (Can you make a formal statement?)

An easier option is to use some randomness when choosing a hash function. Why would this help? Well, imagine that we have an enormous collection of hash functions, each mapping U to $[m]$, and that this collection has the following key property:

> For any given data set of n items, some of these functions might perform badly (as in inevitable, given the reasoning above) but *the vast majority perform well.*

If we pick a function at random from this collection, then we are very likely to get one that is good for the data we happen to have.

There are many ways of defining collections of hash functions that provably have this key property. Here's one.

- Take m to be a prime number. This is not too much of a constraint: we have efficient algorithms for testing primality, and if we keep picking random numbers $m \in [n, 2n]$ then after an average of $O(n)$ tries, we'll find one that is prime.
- Let's say m is $b + 1$ bits long, that is, $2^b \leq m < 2^{b+1}$.
- Write elements of U in binary. This representation will be $O(\log |U|)$ bits long, which we can break into b-bit chunks. In this way, any $z \in U$ becomes a vector

  $$z = (z^{(1)}, z^{(2)}, \ldots, z^{(k)})$$

  where each $z^{(i)}$ is a number in the range $[0, 2^b - 1]$ and $k = O((\log |U|)/b)$.

- A hash function is defined by numbers $a_1, \ldots, a_k \in \{0, 1, \ldots, m-1\}$:

$$h(z) = a_1 z^{(1)} + a_2 z^{(2)} + \cdots + a_k z^{(k)} \bmod m.$$

  By allowing all possible values of $a_1, \ldots, a_k$, we get a collection of m^k hash functions.

This collection of hash functions $\mathcal{H}$ has the following key properties, on account of which it is described as *universal*:

1. Pick any location i and any item $z \in U$. If we pick a hash function h at random from $\mathcal{H}$, then

$$\Pr(h(z) = i) = \frac{1}{m}.$$

2. Pick any distinct items $y, z \in U$. If we pick a hash function h at random from $\mathcal{H}$, then

$$\Pr(h(y) = h(z)) = \frac{1}{m}.$$

(Can you prove these? Remember that any when m is prime, any number in $\{1, 2, \ldots, m-1\}$ is invertible modulo m.)

We would expect these properties to hold if the hash function h were completely random, that is, if $\mathcal{H}$ were the collection of *all* functions from U to $[m]$, or put differently, if h were chosen by picking a random destination for each $x \in U$ separately. But completely random functions are impractical to use, because they take $O(|U|)$ space to write down. Instead, we have a family of very compact functions (each specified by just k numbers), which has a lot of the same behavior.

The properties of universal collections of hash functions, like the one above, has been studied with great care. We won't get into these details. Instead, we'll simply pretend that we are dealing with completely random hash functions (that is, that $\mathcal{H}$ consists of all functions), which is accurate enough for our purposes.

So, let's return to hashing with chaining. With a completely random hash function, what kind of lookup time do we expect? What is the size of the largest bucket? The answer, it turns out, is $O(\log n)$, and is obtained by thinking about the problem in a *balls and bins* framework.

2.3 Balls and bins

Suppose we have n bins and we throw n balls into these bins by picking a bin independently, and uniformly at random, for each ball.

Some bins will get no balls while others might get many balls. The average number of balls per bin is 1. But what is the size of the largest bin? Well, this depends. If we are unlucky, all the balls might fall in the same bin, in which case the answer is n. Or we could be extremely lucky and have every ball land in a separate bin, in which case the answer is 1. But neither alternative is at all likely. The most likely scenario—which occurs with probability at least $1 - 1/2^{100}$, say—is that the largest bin will have $O(\log n)$ balls.

To see this, number the bins $1, 2, \ldots, n$ and number the balls as well. Pick any bin, say bin i, and pick any k balls $S \subset [n]$, where k is a number between 1 and n. The probability that balls S all fall in bin i is $(1/n)^k$, since of the balls has exactly a $1/n$ chance of falling in the bin. Therefore,

$$\Pr(\text{bin } i \text{ gets} \geq k \text{ balls}) \;\leq\; \sum_{S \subset [n], |S|=k} \Pr(\text{balls } S \text{ fall in bin } i) \;=\; \binom{n}{k} \frac{1}{n^k}.$$

Now we can take a union bound over all the bins.

$$\Pr(\text{some bin gets} \geq k \text{ balls}) \;\leq\; \sum_{i=1}^{n} \Pr(\text{bin } i \text{ gets} \geq k \text{ balls}) \;\leq\; n \binom{n}{k} \frac{1}{n^k}.$$

For $k = O(\log n)$, this is less than $1/n^{100}$. We summarize this by saying the largest bin has size $O(\log n)$ *with high probability*, meaning (informally) that the alternative is extremely unlikely.

Returning to hashing with chaining, the balls-and-bins analysis shows that with random hash functions, the largest bucket will have $O(\log n)$ items in it, in which case the time for lookups, inserts, and deletes will be at most this much. This is reassuring, but is it possible to get even faster lookup?

2.4 The power of two choices

Let's return to the scenario with n balls and n bins, but this time, let's place the balls in a slightly more sophisticated way.

- For $i = 1, 2, \ldots n$
  - Pick *two* bins at random for the ith ball
  - Place the ball in whichever of them is less full

It can be shown that, with high probability, the largest bin now has only $O(\log \log n)$ balls in it. For instance, if n is a billion, then $\log n \approx 30$ and $\log \log n \approx 5$.

This immediately suggests a way to do hashing: pick two random hash functions $h_1, h_2 : U \to [n]$. When inserting a new element x, place it in either $T[h_1(x)]$ or $T[h_2(x)]$, whichever has the shorter linked list. And when looking up x, search both lists.

With this little modification, lookups, inserts, and deletes are down to $O(\log \log n)$. Is there any further scope for improvement?

2.5 A two-level hashing scheme

Here's an idea. So far we've been storing n items in a hash table of size n. What if we pick a bigger table, of size $m > n$. How large does m have to be to avoid collisions altogether, so that each linked list has at most one item? Is it enough to have m just slightly larger than n, like $2n$?

In the language of balls and bins, we now have n balls and m bins. The probability that two specific balls (say i and j) collide is $1/m$ (why?). So,

$$\Pr(\text{there is some collision}) \;\leq\; \sum_{i,j} \Pr(\text{balls } i \text{ and } j \text{ collide}) \;=\; \binom{n}{2}\frac{1}{m} \;\approx\; \frac{n^2}{2m}.$$

Setting $m = n^2$ makes this probability less than $1/2$. Of course, we'd like it to be considerably less likely, but the message is clear: we need a very big hash table, of size $O(n^2)$, to avoid collisions altogether. This space requirement is entirely impractical.

But the same idea becomes a lot more feasible if used for a *second* level of hashing. Here is the overall scheme:

- There is a primary hash function $h : U \to [n]$ that sends each item in U to a location in the table $T[1 \cdots n]$.
- Let's say we have n_i items that are hashed to location i. Instead of storing these in a linked list, store them in a second-level hash table, with hash function $h_i : U \to [n_i^2]$.

The lack of collisions makes for constant-time lookups, while the total space, $O(n)$ for the primary table and $O(n_1^2 + \cdots + n_n^2)$ for the secondary table, can be shown to be $O(n)$ with high probability.

2.6 Deterministic versus randomized hashing schemes

The hashing schemes we've considered all have strong guarantees about lookup times that make no assumptions about the specific data items being stored. As we saw earlier, this is made possible by the randomness in the choice of hash function: any deterministic choice is easily defeated.

As a consequence of randomization, some guarantees about effectiveness (lookup time and space usage) hold only *with high probability*. One formal way to define this is to say that these guarantees fail with probability at most $1/n^c$, for some constant $c > 0$. This implies, for instance, that the failure probability is less than $1/2^{100}$, provided n is larger than some fixed constant. An event this unlikely can be disregarded in practice.

For a concrete analogy, consider that there are much fewer than 2^{100} drops of water in the oceans. If you and your friend each independently picked one drop of water at random, anywhere on earth, is it plausible that you would pick exactly the same drop? Of course not. But it is a lot more probable than $1/2^{100}$.

3 Set membership using Bloom filters

Social networks like Facebook have billions of users, with new people joining all the time. How can they rapidly check whether a suggested username is already taken?

Formally, we can think of this as a *set membership* problem. There is a universe U of possible items: all possible usernames, for instance. We need to support two operations:

- `insert`(x): add an item $x \in U$
- `lookup`(x): check whether x has already been added, returning `true` or `false`

How can we implement these? Well, we could just use hashing with chaining, or one of the more sophisticated variants that we have seen. They are reasonable solutions, but the storage per item consists of the item itself plus pointers, and this can easily come out to a few hundred bits. We'd like something significantly more streamlined.

Here's an idea: use a hash table, but don't actually store any of the items. Thus the table $T[1 \cdots m]$ doesn't consist of pointers to linked lists, but instead just has Boolean entries, where $T[i]$ = `true` means "some item hashed to bucket i". The implementation works as follows, given a hash function $h : U \to [m]$:

- Initialize all entries of $T[1 \cdots m]$ to `false`
- `insert`(x): set $T[h(x)]$ = `true`
- `lookup`(x): return $T[h(x)]$

This seems a little too good to be true. Let's take a closer look.

There certainly cannot be *false negatives*: once an item is stored, any subsequent lookup of that item will return `true`. But there might be *false positives*. That is, it is possible for `lookup`(x) to return `true` even if x has not been stored. In the context of our social network example, a false positive isn't too bad—it just means that a suggested new username will be denied even though it is not in use—but we would like to reduce the probability of such mistake. How can we do this?

One strategy for a fix is simply to make m large. This will help, but we expressly want a small-space solution. So instead, we use the most common strategy for reducing error in randomized algorithms: *repetition*. The resulting data structure is called a *Bloom filter*.

The naive way to use repetition is to have k different tables, each of size m and each with its own hash function. A more compact strategy is to have the k different hash functions $h_1, \ldots, h_k : U \to [m]$, but to store everything in the same table.

- Initialize all entries of $T[1 \cdots m]$ to `false`
- `insert`(x): set entries $T[h_1(x)], \ldots, T[h_k(x)]$ to `true`

- lookup(x): return $T[h_1(x)] \wedge \cdots \wedge T[h_k(x)]$

In words, any $x \in U$ corresponds to k buckets $h_1(x), \ldots, h_k(x)$ in the table. To insert an item, the corresponding buckets are marked as occupied. And for lookup, an item is deemed to be present if and only if *all* of the corresponding buckets are occupied.

As before, this scheme has no false negatives. It might have false positives, but we can reduce the probability of these by setting k and m appropriately. As we will soon see, we can make the error probability less than 1% by choosing $m \approx 9.6n$, where n is the number of stored items: less than 10 bits of storage each!

Let's briefly analyze this failure probability. Suppose that after storing n items, the fraction of the table $T[1 \cdots m]$ that is still unoccupied is q. What would we expect q to be, roughly? Fix any bucket in the table. Assuming the hash functions are chosen completely at random,

$$\Pr(\text{this bucket is not hit by the first item}) = \left(1 - \frac{1}{m}\right)^k$$

$$\Pr(\text{this bucket is not hit by any of the } n \text{ items}) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m},$$

where we have used the approximation $e^z \approx 1 + z$ (for small $|z|$). Thus the expected value of q is about $e^{-kn/m}$, and by some more sophisticated arguments that we will not get into, it can be shown that q will be close to this value with high probability.

Now, suppose we are asked to look up an item x that has *not* been stored. What is the probability of making an error?

$$\begin{aligned}\Pr(\text{false positive on } x) &= \Pr(\text{buckets } h_1(x), \ldots, h_k(x) \text{ occupied}) \\ &= (1-q)^k \approx (1 - e^{-kn/m})^k.\end{aligned}$$

To make this failure probability less than a tolerance value ϵ, we can choose

$$\begin{aligned}k &= \frac{m}{n} \ln 2 \\ m &= \frac{1}{(\ln 2)^2} \cdot n \cdot \ln \frac{1}{\epsilon}\end{aligned}$$

When we plug in $\epsilon = 0.01$, we get the setting $m \approx 9.6n$ discussed earlier.

4 Similarity search using fingerprints

Many applications are centered around a large collection of documents: news articles, or web pages, or blog posts. A common scenario is that a query document is subsequently presented, and the goal is to find items in the collection that are *very similar* to this query. For instance, when crawling the web, it is essential to detect when the current page is a near-duplicate of a page that has already been processed. Or, when presenting results in information retrieval, documents that are very similar to each other should be clustered together so that they don't crowd out other alternatives.

There are several steps to defining this problem formally. First, what exactly is a document, and how is it represented? Let's fix an underlying vocabulary V, for instance the set of all English words. We will then allow a document to be any sequence of items from this vocabulary, that is, any element of V^*. A common simplification is to treat a document as a *bag of words*: that is, to model it by the set of words it contains and ignore the ordering of these words altogether. For a document $x \in V^*$, write

$$\text{words}(x) = \{w \in V : \text{word } w \text{ appears in } x\}.$$

Notice that this also disregards the number of occurrences of each word. For instance, $x =$ "`it was the best of times, it was the worst of times`" has

$$\text{words}(x) = \{\texttt{it, was, the, best, of, times, worst}\}.$$

Given two documents x and y, we need a measure of how similar they are. There are many sensible possibilities for this, but an especially popular choice is the *Jaccard coefficient*,

$$J(x, y) = \frac{|\text{words}(x) \cap \text{words}(y)|}{|\text{words}(x) \cup \text{words}(y)|}.$$

If x and y contain exactly the same words, though perhaps with differing multiplicities and orderings, then $J(x, y) = 1$. On the other hand, if they have no words in common, then $J(x, y) = 0$. Two documents can be considered near-duplicates when their Jaccard coefficient is very close to 1.

One way to answer a similarity query is to compute its Jaccard coefficient with every document in the collection and return the closest match. Is there a simpler and faster alternative? We would like to avoid dealing with documents in their entirety: can each document be replaced by a *fingerprint*, a short summary that contains enough information for estimating similarities? One such scheme is known as the *min-hash*.

Here's an idea for summarizing each document by just one word:

- Pick a random permutation (ordering) π of V. For instance, if V consists of animal names, then π might be (`horse, aardvark, jellyfish, dolphin, zebra,` ...).
- Represent any document x by

$$h_\pi(x) = \text{the word in } x \text{ that appears earliest in the ordering } \pi.$$

  For instance, if x is a story set in the ocean, then $h_\pi(x)$ may well be `dolphin`.

This representation does capture some information about x, but seems very inadequate. What can be said about it, exactly?

Pick any two documents x and y. Their hashes, $h_\pi(x)$ and $h_\pi(y)$, are equal if, of all the words in x or in y, the one that appears earliest in π lies in both x *and* y. Thus, when the permutation π is chosen at random,

$$\Pr(h_\pi(x) = h_\pi(y)) = \frac{|\text{words}(x) \cap \text{words}(y)|}{|\text{words}(x) \cup \text{words}(y)|} \;=\; J(x, y).$$

In short: checking whether $h_\pi(x) = h_\pi(y)$ is exactly like flipping a coin with heads probability $J(x, y)$. If we want to estimate $J(x, y)$, all we need to do is to flip such a coin several times and take the average.

Here, then, is the full min-hashing scheme.

- Pick k random permutations $\pi_1, \ldots, \pi_k$ of V.
- Represent each document x by a fingerprint in V^k:

$$h(x) = (h_{\pi_1}(x), \ldots, h_{\pi_k}(x))$$

- To estimate the similarity between documents x and y, use

$$J(x, y) \;\approx\; \frac{\#\text{ of positions on which } h(x) \text{ and } h(y) \text{ agree}}{k}.$$