Chapter 7

Hashing

The basic idea of hashing is that we have keys from a large set U, and we'd like to pack them in a small set M by passing them through some function $h : U \to M$, without getting too many **collisions**, pairs of distinct keys x and y with $h(x) = h(y)$. Where randomization comes in is that we want this to be true even if the adversary picks the keys to hash. At one extreme, we could use a random function, but this will take a lot of space to store.[1] So our goal will be to find functions with succinct descriptions that are still random enough to do what we want.

The presentation in this chapter is based largely on [MR95, §§8.4-8.5] (which is in turn based on work of Carter and Wegman [CW77] on universal hashing and Fredman, Komlós, and Szemerédi [FKS84] on $O(1)$ worst-case hashing); on [PR04] and [Pag06] for cuckoo hashing; and [MU05, §5.5.3] for Bloom filters.

7.1 Hash tables

Here we review the basic idea of **hash tables**, which are implementations of the **dictionary** data type mapping keys to values. The basic idea of hash tables is usually attributed to Dumey [Dum56].[2]

[1] An easy counting argument shows that almost all functions from U to M take $|U| \log |M|$ bits to represent, no matter how clever you are about choosing your representation. This forms the basis for **algorithmic information theory**, which *defines* an object as random if there is no way to reduce the number of bits used to express it.

[2] Caveat: This article is pretty hard to find, so I am basing this citation on its frequent appearance in later sources. This is generally a bad idea that would not really be acceptable in an actual scholarly publication.

U	Universe of all keys
$S \subseteq U$	Set of keys stored in the table
$n = \|S\|$	Number of keys stored in the table
M	Set of table positions
$m = \|M\|$	Number of table positions
$\alpha = n/m$	Load factor

Table 7.1: Hash table parameters

Suppose we want to store n elements from a universe U of in a table with **keys** or **indices** drawn from an index space M of size m. Typically we assume $U = [|U|] = \{0 \dots |U| - 1\}$ and $M = [m] = \{0 \dots m - 1\}$.

If $|U| \leq m$, we can just use an array. Otherwise, we can map keys to positions in the array using a **hash function** $h : U \to M$. This necessarily produces **collisions**: pairs (x, y) with $h(x) = h(y)$, and any design of a hash table must include some mechanism for handling keys that hash to the same place. Typically this is a secondary data structure in each bin, but we may also place excess values in some other place. Typical choices for data structures are linked lists (**separate chaining** or just **chaining**) or secondary hash tables (see §7.3 below). Alternatively, we can push excess values into other positions in the same hash table (**open addressing** or **probing**) or another hash table (see §7.4).

For all of these techniques, the cost will depend on how likely it is that we get collisions. An adversary that knows our hash function can always choose keys with the same hash value, but we can avoid that by choosing our hash function randomly. Our ultimate goal is to do each search in $O(1 + n/m)$ expected time, which for $n \leq m$ will be much better than the $\Theta(\log n)$ time for pointer-based data structures like balanced trees or skip lists. The quantity n/m is called the **load factor** of the hash table and is often abbreviated as α.

All of this only works if we are working in a RAM (random-access machine model), where we can access arbitrary memory locations in time $O(1)$ and similarly compute arithmetic operations on $O(\log|U|)$-bit values in time $O(1)$. There is an argument that in reality any actual RAM machine requires either $\Omega(\log m)$ time to read one of m memory locations (routing costs) or, if one is particularly pedantic, $\Omega(m^{1/3})$ time (speed of light + finite volume for each location). We will ignore this argument.

We will try to be consistent in our use of variables to refer to the different parameters of a hash table. Table 7.1 summarizes the meaning of these variable names.

7.2 Universal hash families

A family of hash functions H is **2-universal** if for any $x \neq y$, $\Pr[h(x) = h(y)] \leq 1/m$ for a uniform random $h \in H$. It's **strongly 2-universal** if for any $x_1 \neq x_2 \in U$, $y_1, y_2 \in M$, $\Pr[h(x_1) = y_1 \wedge h(x_2) = y_2] = 1/m^2$ for a uniform random $h \in H$. Another way to describe strong 2-universality is that the values of the hash function are uniformly distributed and pairwise-independent.

For $k > 2$, **k-universal** usually means **strongly k-universal**: Given distinct $x_1 \ldots x_k$, and any $y_1 \ldots y_k$, $\Pr[h(x_i) = y_i) \forall i] = m^{-k}$. This is equivalent to uniform distribution and k-wise independence. It is possible to generalize the weak version of 2-universality to get a weak version of k-universality ($\Pr[h(x_i)$ are all equal$] \leq m^{-(k-1)}$), but this generalization is not as useful as strong k-universality.

7.3 FKS hashing

The FKS hash table, named for Fredman, Komlós, and Szemerédi [FKS84], is a method for storing a static set S so that we never pay more than constant time for search (not just in expectation), while at the same time not consuming too much space. The assumption that S is static is critical, because FKS chooses hash functions based on the elements of S.

If we were lucky in our choice of S, we might be able to do this with standard hashing. A **perfect hash function** for a set $S \subseteq U$ is a hash function $h : U \to M$ that is injective on S (that is, $x \neq y \to h(x) \neq h(y)$ when $x, y \in S$). Unfortunately, we can only count on finding a perfect hash function if m is large:

Lemma 7.3.1. *If H is 2-universal and $|S| = n$ with $n^2 \leq m$, then there is a perfect $h \in H$ for S.*

Proof. We'll do the usual collision-counting argument. For all $x \neq y$, we have $\delta(x, y, H) \leq |H|/m$. So $\delta(S, S, H) \leq n(n-1)|H|/m$. The Pigeonhole Principle says that there exists a particular $h \in H$ with $\delta(S, S, h) \leq n(n-1)/m < n^2/m \leq 1$. But $\delta(S, S, h)$ is an integer, so it can only be less than 1 by being equal to 0: no collisions. □

Essentially the same argument shows that if $n^2 \leq \alpha m$, then $\Pr[h$ is perfect for $S] \geq 1 - \alpha$. This can be handy if we want to find a perfect hash function and not just demonstrate that it exists.

Using a perfect hash function, we get $O(1)$ search time using $O(n^2)$ space. But we can do better by using perfect hash functions only at the second level of our data structure, which at top level will just be an ordinary hash table. This is the idea behind the Fredman-Komlós-Szemerédi (FKS) hash table [FKS84].

The short version is that we hash to $n = |S|$ bins, then rehash perfectly within each bin. The top-level hash table stores a pointer to a header for each bin, which gives the size of the bin and the hash function used within it. The i-th bin, containing n_i elements, uses $O(n_i^2)$ space to allow perfect

To analyze universal hash families, it is helpful to have some notation for counting collisions. We'll mostly be doing counting rather than probabilities because it saves carrying around a lot of denominators. Since we are assuming uniform choices of h we can alway get back probabilities by dividing by $|H|$.

Let $\delta(x, y, h) = 1$ if $x \neq y$ and $h(x) = h(y)$, 0 otherwise. Abusing notation, we also define, for sets X, Y, and H, $\delta(X, Y, H) = \sum_{x \in X, y \in Y, h \in H} \delta(x, y, h)$, with e.g. $\delta(x, Y, h) = \delta(\{x\}, Y, \{h\})$. Now the statement that H is 2-universal becomes $\forall x, y : \delta(x, y, H) \leq |H|/m$; this says that only a fraction of $1/m$ of the functions in H cause any particular distinct x and y to collide.

If H includes all functions $U \to M$, we get equality: a random function gives $h(x) = h(y)$ with probability exactly $1/m$. But we might do better if each h tends to map distinct values to distinct places. The following lemma shows we can't do too much better:

Lemma 7.2.1. *For any family H, there exist x, y such that $\delta(x, y, H) \geq \frac{|H|}{m}\left(1 - \frac{m-1}{|U|-1}\right)$.*

Since $1 - \frac{m-1}{|U|-1}$ is likely to be very close to 1, we are happy if we get the 2-universal upper bound of $|H|/m$.

Why we care about this: With a 2-universal hash family, chaining using linked lists costs $O(1+s/n)$ expected time per operation. The reason is that the expected cost of an operation on some key x is proportional to the size of the linked list at $h(x)$ (plus $O(1)$ for the cost of hashing itself). But the expected size of this linked list is just the expected number of keys y in the dictionary that collide with x, which is exactly $s\delta(x, y, H) \leq s/n$.

hashing. The total size is $O(n)$ as long as we can show that $\sum_{i=1}^{n} n_i^2 = O(n)$. The time to do a search is $O(1)$ in the worst case: $O(1)$ for the outer hash plus $O(1)$ for the inner hash.

Theorem 7.3.2. *The FKS hash table uses $O(n)$ space.*

Proof. Suppose we choose $h \in H$ as the outer hash function, where H is some 2-universal family of hash functions. Compute:

$$\begin{aligned}\sum_{i=1}^{n} n_i^2 &= \sum_{i=1}^{n} (n_i + n_i(n_i - 1)) \\ &= n + \delta(S, S, h).\end{aligned}$$

Since H is 2-universal, we have $\delta(S, S, H) \leq |H|s(s-1)/n$. But then the Pigeonhole principle says there exists some $h \in H$ with $\delta(S, S, h) \leq \frac{1}{|H|}\delta(S, S, H) \leq n(n-1)/n = n - 1$. This gives $\sum_{i=1}^{n} n_i^2 \leq n + (n-1) = 2n - 1 = O(n)$. □

If we want to find a good h quickly, increasing the size of the outer table to n/α gives us a probability of at least $1 - \alpha$ of getting a good one, using essentially the same argument as for perfect hash functions.

7.4 Cuckoo hashing

Goal: Get $O(1)$ search time in a dynamic hash table at the cost of a messy insertion procedure. In fact, each search takes only two reads, which can be done in parallel; this is optimal by a lower bound of Pagh [Pag01], which shows a matching upper bound for static dictionaries. **Cuckoo hashing** is an improved version of this result that allows for dynamic insertions.

Cuckoo hashing was invented by Pagh and Rodler [PR04]; the version described here is based on a simplified version from notes of Pagh [Pag06] (the main difference is that it uses just one table instead of the two tables—one for each hash function—in [PR04]).

7.4.1 Structure

We have a table T of size n, with two separate, independent hash functions h_1 and h_2. These functions are assumed to be k-universal for some sufficiently large value k; as long as we never look at more than k values at once, this means we can treat them effectively as random functions. In practice, using crummy hash functions seems to work just fine, a common property of hash

tables. There are also specific hash functions that have been shown to work with particular variants of cuckoo hashing [PR04, PT12]. We will avoid these issues by assuming that our hash functions are actually random.

Every key x is stored either in $T[h_1(x)]$ or $T[h_2(x)]$. So the search procedure just looks at both of these locations and returns whichever one contains x (or fails if neither contains x).

To insert a value $x_1 = x$, we must put it in $T[h_1(x_1)]$ or $T[h_2(x_1)]$. If one or both of these locations is empty, we put it there. Otherwise we have to kick out some value that is in the way (this is the "cuckoo" part of cuckoo hashing, named after the bird that leaves its eggs in other birds' nests). We do this by letting $x_2 = T[h_1(x_1)]$ and writing x_1 to $T[h_1(x_1)]$. We now have a new "nestless" value x_2, which we swap with whatever is in $T[h_2(x_2)]$. If that location was empty, we are done; otherwise, we get a new value x_3 that we have to put in $T[h_1(x_3)]$ and so on. The procedure terminates when we find an empty spot or if enough iterations have passed that we don't expect to find an empty spot, in which case we rehash the entire table. This process can be implemented succinctly as shown in Algorithm 7.1.

```
1  procedure insert(x)
2      if T(h1(x)) = x or T(h2(x)) = x then
3          return
4      pos ← h1(x)
5      for i ← 1...n do
6          if T[pos] = ⊥ then
7              T[pos] ← x
8              return
9          x ⇄ T[pos]
10         if pos = h1(x) then
11             pos ← h2(x)
12         else
13             pos ← h1(x)
14     If we got here, rehash the table and reinsert x.
```

Algorithm 7.1: Insertion procedure for cuckoo hashing. Adapted from [Pag06]

A detail not included in the above code is that we always rehash (in theory) after m^2 insertions; this avoids potential problems with the hash functions used in the paper not being universal enough. We will avoid this

issue by assuming that our hash functions are actually random (instead of being approximately n-universal with reasonably high probability). For a more principled analysis of where the hash functions come from, see [PR04]. An alternative hash family that is known to work for a slightly different variant of cuckoo hashing is tabulation hashing, as described in §7.2.2; the proof that this works is found in [PT12].

7.4.2 Analysis

The main question is how long it takes the insertion procedure to terminate, assuming the table is not too full.

First let's look at what happens during an insert if we have a lot of nestless values. We have a sequence of values $x_1, x_2, \ldots$, where each pair of values x_i, x_{i+1} collides in h_1 or h_2. Assuming we don't reach the loop limit, there are three main possibilities (the leaves of the tree of cases below):

1. Eventually we reach an empty position without seeing the same key twice.

2. Eventually we see the same key twice; there is some i and $j > i$ such that $x_j = x_i$. Since x_i was already moved once, when we reach it the second time we will try to move it back, displacing x_{i-1}. This process continues until we have restored x_2 to $T[h_1(x_1)]$, displacing x_1 to $T[h_2(x_1)]$ and possibly creating a new sequence of nestless values. Two outcomes are now possible:

   (a) Some x_ℓ is moved to an empty location. We win!

   (b) Some x_ℓ is moved to a location we've already looked at. We lose! We find we are playing musical chairs with more players than chairs, and have to rehash.

Let's look at the probability that we get the last, *closed loop* case. Following Pagh-Rodler, we let v be the number of distinct nestless keys in the loop. Since v includes x_1, v is at least 1. We can now count how many different ways such a loop can form.

There are at most v^3 choices for i, j, and ℓ, m^{v-1} choices of cells for the loop, and n^{v-1} choices for the non-x_1 elements of the loop. For each non-x_i element, its initial placement may determined by either h_1 or h_2; this gives another 2^{v-1} choices.[4] This gives a total of $v^3(2nm)^{v-1}$ possible closed loops starting with x_1 that have v distinct nodes.

[4] The original analysis in [PR04] avoids this by alternating between two tables, so that we can determine which of h_1 or h_2 is used at each step by parity.

Since each particular loop allows us to determine both h_1 and h_2 for all v of its elements, the probability that we get exactly these hash values (so that the loop occurs) is m^{-2v}. Summing over all closed loops with v elements gives a total probability of $v^2(2nm)^{v-1}m^{-2v} = v^3(2n/m)^{v-1}m^{-v-1} \leq v^3(2n/m)^{v-1}m^{-2}$.

Now sum over all $v \geq 1$. We get $m^{-2}\sum_{v=1}^{n} v^3(2n/m)^{v-1} < m^{-2}\sum_{v=1}^{\infty} v^3(2n/m)^{v-1}$. The series converges if $2n/m < 1$, so for any fixed $\alpha < 1/2$, the probability of any closed loop forming is $O(m^{-2}$. Since the cost of hitting a closed loop is $O(n+m) = O(m)$, this adds only $O(m^{-1})$ to the insertion complexity.

Now we look at what happens if we don't get a closed loop. This doesn't force us to rehash, but if the path is long enough, we may still pay a lot to do an insertion.

It's a little messy to analyze the behavior of keys that appear more than once in the sequence, so the trick used in the paper is to observe that for any sequence of nestless keys $x_1 \ldots x_p$, there is a subsequence of size $p/3$ with no repetitions that starts with x_1. This will be either the sequence S_1 given by $x_1 \ldots x_{j-1}$—the sequence starting with the first place we try to insert x_1–or S_2 given by $x_1 = x_{i+j-1} \ldots x_p$, the sequence starting with the second place we try to insert x_1. Between these we have a third sequence S_3 where we undo some of the moves made in S_1. Because $|S_1| + |S_3| + |S_2| \geq p$, at least one of thee subsequences has size $p/3$. But $|S_3| \leq |S_1|$, so it must be either S_1 or S_2.

We can then argue that the probability that we get a sequence of v distinct keys in either S_1 or S_2 most $2(n/m)^{v-1}$ (since we have to hit a nonempty spot, with probability at most n/m, at each step, but there are two possible starting locations), which gives an expected insertion time bounded by $\sum 3v(n/m)^{v-1} = O(1)$, assuming n/m is bounded by a constant less than 1. Since we already need $n/m \leq 1/2$ to avoid the bad closed-loop case, we can use this here as well.

An annoyance with cuckoo hashing is that it has high space overhead compared to more traditional hash tables: in order for the first part of the analysis above to work, the table must be at least half empty. This can be avoided at the cost of increasing the time complexity by choosing between d locations instead of 2. This technique, due to Fotakis *et al.* [FPSS03], is known as **d-ary cuckoo hashing**; for suitable choice of d it uses $(1+\epsilon)n$ space and guarantees that a lookup takes $O(1/\epsilon)$ probes while insertion takes $(1/\epsilon)^{O(\log\log(1/\epsilon))}$ steps in theory and appears to take $O(1/\epsilon)$ steps in experiments done by the authors.

7.5 Practical issues

For large hash tables, local probing schemes are faster, because it is likely that all of the locations probed to find a particular value will be on the same virtual memory page. This means that a search for a new value usually requires one cache miss instead of two. **Hopscotch hashing** [HST08] combines ideas from linear probing and cuckoo hashing to get better performance than both in practice.

Hash tables that depend on strong properties of the hash function may behave badly if the user supplies a crummy hash function. For this reason, many library implementations of hash tables are written defensively, using algorithms that respond better in bad cases. See `http://svn.python.org/view/python/trunk/Objects/dictobject.c` for an example of a widely-used hash table implementation chosen specifically because of its poor theoretical characteristics.

7.6 Bloom filters

See [MU05, §5.5.3] for basics and a formal analysis or `http://en.wikipedia.org/wiki/Bloom_filter` for many variations and the collective wisdom of the unwashed masses. The presentation here mostly follows [MU05].

7.6.1 Construction

Bloom filters are a highly space-efficient randomized data structure invented by Burton H. Bloom [Blo70] that store sets of data, with a small probability that elements not in the set will be erroneously reported as being in the set.

Suppose we have k independent hash functions $h_1, h_2, \ldots, h_k$. Our memory store A is a vector of m bits, all initially zero. To store a key x, set $A[h_i(x)] = 1$ for all i. To test membership for x, see if $A[h_i(x)] = 1$ for all i. The membership test always gives the right answer if x is in fact in the Bloom filter. If not, we might decide that x is in the Bloom filter anyway.

7.6.2 False positives

The probability of such **false positives** can be computed in two steps: first, we estimate how many of the bits in the Bloom filter are set after inserting n values, and then we use this estimate to compute a probability that any fixed x shows up when it shouldn't.

If the h_i are close to being independent random functions,[5] then with n entries in the filter we have $\Pr[A[i] = 1] = 1 - (1 - 1/m)^{kn}$, since each of the kn bits that we set while inserting the n values has one chance in m of hitting position i.

We'd like to simplify this using the inequality $1 + x \le e^x$, but it goes in the wrong direction; instead, we'll use $1 - x \ge e^{-x-x^2}$, which holds for $0 \le x \le 0.683803$ and in our application holds for $m \ge 2$. This gives

$$\begin{aligned}\Pr[A[i] = 1] &\le 1 - (1 - 1/m)^{kn} \\ &\le 1 - e^{-k(n/m)(1+1/m)} \\ &= 1 - e^{-k\alpha(1+1/m)} \\ &= 1 - e^{-k\alpha'}\end{aligned}$$

where $\alpha = n/m$ is the load factor and $\alpha' = \alpha(1 + 1/m)$ is the load factor fudged upward by a factor of $1 + 1/m$ to make the inequality work.

Suppose now that we check to see if some value x that we never inserted in the Bloom filter appears to be present anyway. This occurs if $A[h_i(x)] = 1$ for all i. Since each $h_i(x)$ is assumed to be an independent uniform probe of A, the probability that they all come up 1 conditioned on A is

$$\left(\frac{\sum A[i]}{m}\right)^k. \tag{7.6.1}$$

We have an upper bound $\mathrm{E}\left[\sum A[i]\right] \le m\left(1 - e^{-k\alpha'}\right)$

So let's assume for simplicity that our false positive probability is exactly $(1 - e^{-k\alpha'})^k$. We can choose k to minimize this quantity for fixed α' by doing the usual trick of taking a derivative and setting it to zero; to avoid weirdness with the k in the exponent, it helps to take the logarithm first (which doesn't affect the location of the minimum), and it further helps to take the derivative with respect to $x = e^{-\alpha' k}$ instead of k itself. Note that when we do this, $k = -\frac{1}{\alpha'} \ln x$ still depends on x, and we will deal with this by applying this substitution at an appropriate point.

[5] We are going sidestep the rather deep swamp of how plausible this assumption is and what assumption we should be making instead; however, it is known [KM08] that starting with two sufficiently random-looking hash functions h and h' and setting $h_i(x) = h(x) + ih'(x)$ works.

Compute

$$\begin{aligned}\frac{d}{dx}\ln\left((1-x)^k\right) &= \frac{d}{dx}k\ln(1-x)\\ &= \frac{d}{dx}\left(-\frac{1}{\alpha'}\ln x\right)\ln(1-x)\\ &= -\frac{1}{\alpha'}\left(\frac{\ln(1-x)}{x}-\frac{\ln x}{1-x}\right).\end{aligned}$$

Setting this to zero gives $(1-x)\ln(1-x) = x\ln x$, which by symmetry has the unique solution $x = 1/2$, giving $k = \frac{1}{\alpha'}\ln 2$.

In other words, to minimize the false positive rate for a known load factor α, we want to choose $k = \frac{1}{\alpha'}\ln 2 = \frac{1}{\alpha(1+1/m)}\ln 2$, which makes each bit one with probability approximately $1 - e^{-\ln 2} = \frac{1}{2}$. This makes intuitive sense, since having each bit be one or zero with equal probability maximizes the entropy of the data.

The probability of a false positive is then $2^{-k} = 2^{-\ln 2/\alpha'}$. For a given maximum false positive rate ϵ, and assuming optimal choice of k, we need to keep $\alpha' \le \frac{\ln^2 2}{\ln(1/\epsilon)}$ or $\alpha \le \frac{\ln^2 2}{(1+1/m)\ln(1/\epsilon)}$.

Alternatively, if we fix ϵ and n, we need $m/(1+1/m) \ge n \cdot \frac{\ln(1/\epsilon)}{\ln^2 2} \approx 1.442n\lg(1/\epsilon)$, which works out to $m \ge 1.442n\lg(1/\epsilon) + O(1)$. This is very good for constant ϵ.

Note that for this choice of m, we have $\alpha = O(1/\ln(1/\epsilon))$, giving $k = O(\log\log(1/\epsilon))$. So for polynomial ϵ, we get $k = O(\log\log n)$. This means that not only do we use little space, but we also have very fast lookups (although not as fast as the $O(1)$ cost of a real hash table).

7.6.3 Comparison to optimal space

If we wanted to design a Bloom-filter-like data structure from scratch and had no constraints on processing power, we'd be looking for something that stored an index of size $\lg M$ into a family of subsets $S_1, S_2, \ldots S_M$ of our universe of keys U, where $|S_i| \le \epsilon|U|$ for each i (giving the upper bound on the false positive rate)[6] and for any set $A \subseteq U$ of size n, $A \subseteq S_i$ for at least one S_i (allowing us to store A).

Let $N = |U|$. Then each set S_i covers $\binom{\epsilon N}{n}$ of the $\binom{N}{n}$ subsets of size n. If we could get them to overlap optimally (we can't), we'd still need a minimum of $\binom{N}{n} \Big/ \binom{\epsilon N}{n} = (N)_n/(\epsilon N)_n \approx (1/\epsilon)^n$ sets to cover everybody, where the approximation assumes $N \gg n$. Taking the log gives $\lg M \approx n \lg(1/\epsilon)$, meaning we need about $\lg(1/\epsilon)$ bits per key for the data structure. Bloom filters use $1/\ln 2$ times this.

There are known data structures that approach this bound asymptotically; see Pagh *et al.* [PPR05]. These also have other desirable properties, like supporting deletions and faster lookups if we can't look up bits in parallel. As far as I know, they are not used much in practice.

7.6.4 Applications

Bloom filters are popular in networking and database systems because they can be used as a cheap test to see if some key is actually present in a data structure that it's otherwise expensive to search in. Bloom filters are particular nice in hardware implementations, since the k hash functions can be computed in parallel.

An example is the **Bloomjoin** in distributed databases [ML86]. Here we want to do a join on two tables stored on different machines (a join is an operation where we find all pairs of rows, one in each table, that match on some common key). A straightforward but expensive way to do this is to send the list of keys from the smaller table across the network, then match

[6] Technically, this gives a weaker bound on false positives. For standard Bloom filters, assuming random hash functions, each key individually has at most an ϵ probability of appearing as a false positive. The hypothetical data structure we are considering here—which is effectively deterministic—allows the set of false positives to depend directly on the set of keys actually inserted in the data structure, so in principle the adversary could arrange for a specific key to appear as a false positive with probability 1 by choosing appropriate keys to insert. So this argument may underestimate the space needed to get make the false positives less predictable. On the other hand, we aren't charging the Bloom filter for the space needed to store the hash functions, which could be quite a bit if they are genuine random functions.

them against the corresponding keys from the larger table. If there are n_s rows in the smaller table, n_b rows in the larger table, and j matching rows in the larger table, this requires sending n_s keys plus j rows. If instead we send a Bloom filter representing the set of keys in the smaller table, we only need to send $\lg(1/\epsilon)/\ln 2$ bits for the Bloom filter plus an extra ϵn_b rows on average for the false positives. This can be cheaper than sending full keys across if the number of false positives is reasonably small.

7.6.5 Counting Bloom filters

It's not hard to modify a Bloom filter to support deletion. The basic trick is to replace each bit with a counter, so that whenever a value x is inserted, we increment $A[h_i(x)]$ for all i and when it is deleted, we decrement the same locations. The search procedure now returns $\min_i A[h_i(x)]$ (which means that it principle it can even report back multiplicities, though with some probability of reporting a value that is too high). To avoid too much space overhead, each array location is capped at some small maximum value c; once it reaches this value, further increments have no effect. The resulting structure is called a **counting Bloom filter**, due to Fan *et al.* [FCAB00].

We'd only expect this to work if our chances of hitting the cap is small. Fan *et al.* observe that the probability that the m table entries include one that is at least c after n insertions is bounded by

$$\begin{aligned} m\binom{nk}{c}\frac{1}{m^c} &\le m\left(\frac{enk}{c}\right)^c\frac{1}{m^c} \\ &= m\left(\frac{enk}{cm}\right)^c \\ &= m(ek\alpha/c)^c. \end{aligned}$$

(This uses the bound $\binom{n}{k} \le \left(\frac{en}{k}\right)^k$, which follows from Stirling's formula.)

For $k = \frac{1}{\alpha}\ln 2$, this is $m(e\ln 2/c)^c$. For the specific value of $c = 16$ (corresponding to 4 bits per entry), they compute a bound of $1.37 \times 10^{-15} m$, which they argue is minuscule for all reasonable values of m (it's a systems paper).

The possibility that a long chain of alternating insertions and deletions might produce a false negative due to overflow is considered in the paper, but the authors state that "the probability of such a chain of events is so low that it is much more likely that the proxy server would be rebooted in the meantime and the entire structure reconstructed." An alternative way of dealing with this problem is to never decrement a maxed-out register; this never produces a false negative, but may cause the filter to slowly fill up with maxed-out registers, producing a higher false-positive rate.

A fancier variant of this idea is the **spectral Bloom filter** [CM03], which uses larger counters to track multiplicities of items. The essential idea here is that we can guess that the number of times a particular value x was inserted is equal to $\min_{i=1}^{m} A[h_i(x)]$), with some extra tinkering to detect errors based on deviations from the typical joint distribution of the $A[h_i(x)]$ values. A fancier version of this idea gives the count-min sketches of the next section.

7.6.6 Count-min sketches

Count-min sketches are designed for use in **data stream computation**. In this model, we are given a huge flood of data—far too big to store—in a single pass, and want to incrementally build a small data structure, called a **sketch**, that will allow us to answer statistical questions about the data after we've processed it all. The motivation is the existence of data sets that are too large to store at all (network traffic statistics), or too large to store in fast memory (very large database tables). By building a sketch we can make one pass through the data set but answer queries after the fact, with some loss of accuracy.

An example of a problem in this model is that we are presented with a sequence of pairs (i_t, c_t) where $1 \leq i_t \leq n$ is an *index* and c_t is a *count*, and we want to construct a sketch that will allows us to approximately answer statistical queries about the vector a given by $a_i = \sum_{t,i[t]=i} c_t$. The size of the sketch should be polylogarithmic in the size of a and the length of the stream, and polynomial in the error bounds. Updating the sketch given a new data point should be cheap.

A solution to this problem is given by the **count-min sketch** of Cormode and Muthukrishnan [CM05] (see also [MU05, §13.4]). This gives approximations of a_i, $\sum_{i=\ell}^{r} a_i$, and $a \cdot b$ (for any fixed b), and can be used for more complex tasks like finding **heavy hitters**—indices with high weight. The easiest case is approximating a_i when all the c_t are non-negative, so we'll start with that.

7.6.6.1 Initialization and updates

To construct a count-min sketch, build a two-dimensional array c with depth $d = \lceil \ln(1/\delta) \rceil$ and width $w = \lceil e/\epsilon \rceil$, where ϵ is the error bound and δ is the probability of exceeding the error bound. Choose d independent hash functions from some 2-universal hash family; we'll use one of these hash function for each row of the array. Initialize c to all zeros.

The update rule: Given an update (i_t, c_t), increment $c[j, h_j(i_t)]$ by c_t for $j = 1 \ldots d$. (This is the *count* part of count-min.)

7.6.6.2 Queries

Let's start with **point queries**. Here we want to estimate a_i for some fixed i. There are two cases, depending on whether the increments are all non-negative, or arbitrary. In both cases we will get an estimate whose error is

linear in both the error parameter ϵ and the ℓ_1-norm $\|a\|_1 = \sum_i |a_i|$ of a. It follows that the relative error will be low for heavy points, but we may get a large relative error for light points (and especially large for points that don't appear in the data set at all).

For the non-negative case, to estimate a_i, compute $\hat{a}_i = \min_j c[j, h_j(i)]$. (This is the *min* part of coin-min.) Then:

Lemma 7.6.1. *When all c_t are non-negative, for $\hat{a}_i$ as defined above:*

$$\hat{a}_i \geq a_i, \tag{7.6.2}$$

and

$$\Pr\left[\hat{a}_i \leq a_i + \epsilon\|a\|_1\right] \geq 1 - \delta. \tag{7.6.3}$$

Proof. The lower bound is easy. Since for each pair (i, c_t) we increment each $c[j, h_j(i)]$ by c_t, we have an invariant that $a_i \leq c[j, h_j(i)]$ for all j throughout the computation, which gives $a_i \leq \hat{a}_i = \min_j c[j, h_j(i)]$.

For the upper bound, let I_{ijk} be the indicator for the event that $(i \neq k) \wedge (h_j(i) = h_j(k))$, i.e., that we get a collision between i and k using h_j. The 2-universality property of the h_j gives $\mathrm{E}\left[I_{ijk}\right] \leq 1/w \leq \epsilon/e$.

Now let $X_{ij} = \sum_{k=1}^{n} I_{ijk} a_k$. Then $c[j, h_j(i)] = a_i + X_{ij}$. (The fact that $X_{ij} \geq 0$ gives an alternate proof of the lower bound.) Now use linearity of expectation to get

$$\begin{aligned}
\mathrm{E}\left[X_{ij}\right] &= \mathrm{E}\left[\sum_{k=1}^{n} I_{ijk} a_k\right] \\
&= \sum_{k=1}^{n} a_k \,\mathrm{E}\left[I_{ijk}\right] \\
&\leq \sum_{k=1}^{n} a_k (\epsilon/e) \\
&= (\epsilon/e)\|a\|_1.
\end{aligned}$$

So $\Pr\left[c[j, h_j(i)] > a_i + \epsilon\|a\|_1\right] = \Pr\left[X_{ij} > e\,\mathrm{E}\left[X_{ij}\right]\right] < 1/e$, by Markov's inequality. With d choices for j, and each h_j chosen independently, the probability that every count is too big is at most $(1/e)^{-d} = e^{-d} \leq \exp(-\ln(1/\delta)) = \delta$. □

Now let's consider the general case, where the increments c_t might be negative. We still initialize and update the data structure as described in §7.6.6.1, but now when computing $\hat{a}_i$, we use the median count instead of the minimum count: $\hat{a}_i = \mathrm{median}\left\{c[j, h_j(i)] \mid j = 1 \ldots n\right\}$. Now we get:

Lemma 7.6.2. *For $\hat{a}_i$ as defined above,*

$$\Pr\left[a_i - 3\epsilon\|a\|_1 \le \hat{a}_i \le a_i + 3\epsilon\|a\|_1\right] > 1 - \delta^{1/4}. \tag{7.6.4}$$

Proof. We again define the error term X_{ij} as above, and observe that

$$\begin{aligned}
\mathrm{E}\left[|X_{ij}|\right] &= \mathrm{E}\left[\left|\sum_k I_{ijk} a_k\right|\right] \\
&\le \sum_{k=1}^{n} |a_k \,\mathrm{E}\left[I_{ijk}\right]| \\
&\le \sum_{k=1}^{n} |a_k|(\epsilon/e) \\
&= (\epsilon/e)\|a\|_1.
\end{aligned}$$

Using Markov's inequality, we get $\Pr\left[|X_{ij}|\right] > 3\epsilon\|a\|_1] = \Pr\left[|X_{ij}| > 3e\,\mathrm{E}\left[X_{ij}\right]\right] < 1/3e < 1/8$. In order for the median to be off by more than $3\epsilon\|a\|_1$, we need $d/2$ of these low-probability events to occur. The expected number that occur is $\mu = d/8$, so applying the standard Chernoff bound (5.2.1) with $\delta = 3$ we are looking at

$$\begin{aligned}
\Pr\left[S \ge d/2\right] &= \Pr\left[S \ge (1+3)\mu\right] \\
&\le (e^3/4^4)^{d/8} \\
&\le (e^{3/8}/2)^{\ln(1/\delta)} \\
&= \delta^{\ln 2 - 3/8} \\
&< \delta^{1/4}
\end{aligned}$$

(the actual exponent is about 0.31, but 1/4 is easier to deal with). This immediately gives (7.6.4). □

One way to think about this is that getting an estimate within $\epsilon\|a\|_1$ of the right value with probability at least $1-\delta$ requires 3 times the width and 4 times the depth—or 12 times the space and 4 times the time—when we aren't assuming increments are non-negative.

Next, we consider inner products. Here we want to estimate $a \cdot b$, where a and b are both stored as count-min sketches using the same hash functions. The paper concentrates on the case where a and b are both non-negative, which has applications in estimating the size of a join in a database. The method is to estimate $a \cdot b$ as $\min_j \sum_{k=1}^{w} c_a[j,k] \cdot c_b[j,k]$.

For a single j, the sum consists of both good values and bad collisions; we have $\sum_{k=1}^{w} c_a[j,k] \cdot c_b[j,k] = \sum_{k=1}^{n} a_i b_i + \sum_{p \neq q, h_j(p)=h_j(q)} a_p b_q$. The second term has expectation

$$\begin{aligned}\sum_{p \neq q} \Pr\left[h_j(p) = h_j(q)\right] a_p b_q &\leq \sum_{p \neq q} (\epsilon/e) a_p b_q \\ &\leq \sum_{p,q} (\epsilon/e) a_p b_q \\ &\leq (\epsilon/e) \|a\|_1 \|b\|_1.\end{aligned}$$

As in the point-query case, we get probability at most $1/e$ that a single j gives a value that is too high by more than $\epsilon \|a\|_1 \|b\|_1$, so the probability that the minimum value is too high is at most $e^{-d} \leq \delta$.

7.6.6.3 Finding heavy hitters

Here we want to find the heaviest elements in the set: those indices i for which a_i exceeds $\phi \|a\|_1$ for some constant threshold $0 < \phi \leq 1$.

The easy case is when increments are non-negative (for the general case, see the paper), and uses a method from a previous paper by Charikar *et al.* [CCFC04]. Because $\|a\|_1 = \sum_i a_i$, we know that there will be at most $1/\phi$ heavy hitters. But the tricky part is figuring out which elements they are.

Instead of trying to find the elements after the fact, we extend the data structure and update procedure to track all the heavy elements found so far (stored in a heap), as well as $\|a\|_1 = \sum c_t$. When a new increment (i, c) comes in, we first update the count-min structure and then do a point query on a_i; if $\hat{a}_i \geq \phi \|a\|_1$, we insert i into the heap, and if not, we delete i along with any other value whose stored point-query estimate has dropped below threshold.

The trick here is that the threshold $\phi \|a\|_1$ only increases over time (remember that we are assuming non-negative increments). So if some element i is below threshold at time t, it can only go above threshold if it shows up again, and we have a probability of at least $1 - \delta$ of including it then. This means that every heavy hitter appears in the heap with probability at least $1 - \delta$.

The total space cost for this data structure is the cost of the count-min structure plus the cost of the heap; this last part will be $O((1+\epsilon)/\phi)$ with reasonably high probability, since this is the maximum number of elements that have weight at least $\phi \|a\|_1/(1+\epsilon)$, the minimum needed to get an apparent weight of $\phi \|a\|_1$ even after taking into account the error in the count-min structure.