Bloom Filters and Cuckoo Filters

Sam McCauley
Applied Algorithms, Fall 2024

Goals for Today

The first section of this course was about time and space. We learned about how saving space can help with running time. In particular, smaller space often means better cache efficiency.

Today, we are using randomization to take this idea one step further, and talk about two methods to *compress* data.

You may have learned in the past (for example, in CS 361) that data cannot be losslessly compressed in the worst case. But as with all of the algorithms we learn in this class, we do want good worst-case bounds. So if the goal is to obtain worst-case compression bounds, our compression can't be lossless: we must lose something along the way.

A Bloom filter answers *membership queries*: it stores a set S, and for a query q, it answers $q \in S$ (q is in the set), or $q \notin S$ (q is not in the set).

Bloom filters are incredibly useful because they compress data while giving *guarantees* on exactly what is lost—and what isn't.

First guarantee: No false negatives. A Bloom filter guarantees that the answer $q \notin S$ is always correct: we can be completely sure that q is not in the set S.

Equivalently, we can frame this guarantee in terms of queries to members of S. If we query some $q \in S$, then we are guaranteed that the Bloom filter will answer $q \in S$.

Second guarantee: Bounded false positive rate. Consider a query $q \notin S$. The Bloom filter is guaranteed to only (incorrectly) return $q \in S$ with probability at most ε. This ε is called the *false positive rate*.

Bloom filters allow us to choose any ε we want. The smaller the value of ε is, the fewer mistakes our filter makes. But, in exchange, the compression will be less effective for small ε; that is to say, our filter will take up more space.

Formal Definitions

Let us be a bit more careful with our vocabulary. We will refer to any data structure meeting the requirements above—no false negatives, and a tunable false positive rate ε—as a "filter." We will reserve the term "Bloom filter" for the specific filter invented by Burton Bloom.

Let's more formally define what we mean by "filter." A filter is a data structure that stores a set S of size n. Each element of S must be from a universe U.[1] The filter answers membership queries: for some element $q \in U$, a query asks "is $q \in S$?" The filter either answers "yes, $q \in S$" or "no, $q \notin S$."

A filter has two guarantees. First, no false negatives: if the query $q \in S$, then the filter must answer "yes, $q \in S$" with probability 1. Second, a false positive guarantee: if $q \notin S$, the filter must answer "yes, $q \in S$" with probability at most ε.

A filter must be space-efficient. The Bloom filter uses $1.44n\log_2(1/\varepsilon)$ bits of space. The cuckoo filter uses $2n\log_2(1/\varepsilon+1)$ bits of space; we will show how to reduce this to $1.05n\log_2(1/\varepsilon+1)+3.15n$ bits. Both are very space-efficient, but the cuckoo filter is up to 27% smaller for sufficiently small ε.

Being Specific with Space Usage. Since we are compressing data, every bit counts. With this in mind, the space of a filter is usually measured in *bits* (not bytes or words of space). Furthermore, we will explicitly track constants when discussing space rather than using O notation.

History and Discussion

The Bloom filter was invented by Burton H. Bloom while he was a graduate student at MIT in 1970. His original publication [1] focused on practical performance, and he did not attempt a theoretical analysis of the data structure.

Today we will also discuss the cuckoo filter [3], a more recent filter that gives the same guarantees as a Bloom filter, but provides better space usage as well as better cache performance. This effect is most dramatic when ε is small (that is to say, when the data is not too compressed—the filter takes up a relatively large amount of space).

The Bloom filter is used in an incredibly large number of contexts. It is fairly likely that a recent operation you performed on your computer was sped up using a Bloom filter. The ideas that make up the Bloom filter have been the foundation for further work on compressed data structures. For example, next lecture we will be learning about the Count-Min Sketch, which uses many of the same ideas to compress much more aggressively.

[1]For example, U may be "all possible strings," or it could be "all 64-bit integers." The only reason we need U is that we need to be able to hash elements—whatever hash we choose must be able to hash all elements from U.

Uses

Cache-efficiency. First, let us discuss using the Bloom filter as a way to improve cache-efficiency. Let's say that we are storing a large amount of data—in fact such a large amount that it needs to be stored in relatively slow memory (like RAM).[2] Furthermore, assume that we frequently make queries to items not actually stored. For example, it may be that we are inserting elements into a database. Each inserted key x is preceded by a query "is $x \in S$" to ensure that there are no two items with the same key.

These queries are very expensive because of the size of the database. Using the vocabulary we learned in the first part of the class, they require at least one cache miss.[3]

The idea is to maintain a filter on S to avoid these queries. The filter is very small; in fact so small that it fits in cache. This means that it can be accessed without any cache misses.

Why can we do this? The heart of this approach is the first filter guarantee. If we are looking for data that *is* stored, we definitely want to take the cost to access the data. The filter does this: if we query a $q \in S$, the filter is guaranteed to always give the correct answer. So to queries in the set, we access the data just like we would have without a filter.

How much does it save us? The second filter guarantee gives us the performance. Let's say we have a query $q \notin S$. Then with probability $1 - \varepsilon$, we do not need to access the data at all, saving us cache misses (as well as some computation). To put a finer point on it: let's say I make X queries to the data, all for things that are not actually stored. Without a filter, this takes X accesses to the full dataset, taking at least X cache misses (and possibly many more). With a filter, this only takes εX accesses to the dataset. For example, if we use a cuckoo filter with $\varepsilon = 1/64$, we can store the dictionary with only 1 byte of space per word (likely to fit in cache), and the number of expensive accesses decreases by $1 - 1/64 \approx 98.4\%$.

Approximate Sets. In the above example, we wanted to simply improve query performance. The set S—and in fact, an entire database—was stored as a backup.

But for some applications, we don't want to store all of S; storing the Bloom filter alone is enough. In these applications it's not a big deal if an answer is incorrect, i.e. if the filter answers $q \in S$ incorrectly.

The classic example of this use case is a spell checker. Let's say we want to build a spell checker for a text application, but we don't have room to store an entire dictionary.[4] Instead, we can store the

[2] Remember, as we talked about earlier, that memory lower in the cache hierarchy is larger, but is much slower.

[3] In fact, frequently these items will be stored using a B-tree (we haven't discussed in detail, but in short a B tree is a binary-search-tree-like data structure for efficient external-memory searching), in which case each query requires $\Theta(\log_B n)$ cache misses—very expensive indeed!

[4] This seems silly by modern standards, but in fact this was the one of the original use cases for a Bloom filter! In the 1970s,

dictionary words in a filter. If we query a word that is spelled correctly, the filter is guaranteed to correctly recognize it as spelled correctly. If we query an incorrectly-spelled word, the filter marks it as misspelled with probability $\geq 1 - \varepsilon$. For example, if we use a cuckoo filter with $\varepsilon = 1/64$, we can store the dictionary with only 1 byte of space per word, and still have our spell checker identify misspelled words 98.4% of the time.

Bloom Filters

A Bloom filter consists of $k = \log_2 1/\varepsilon$ hash functions, which I will denote using $h_1, h_2, \ldots, h_k$, and an array A of $m = (\log_2 e)nk \approx 1.44nk$ bits. Let $A[i]$ denote the ith bit of A. For each $i = 1, \ldots, k$, $h_i : U \to \{0, m-1\}$ (that is to say, h_i maps an element from the universe of possible elements U to a slot in the hash table).

Notice that m and k, as defined above, may not be integers. For m we can just round up; this only means that the data structure requires an extra bit of space. For k we have a more difficult problem: if we round down we will not achieve the desired ε; if we round up we will take an extra bit of space for every stored element. In that sense rounding up is the safer strategy.

Building a Bloom Filter. We begin by setting all bits of A to 0: $A[i] = 0$ for all i.

For each $x \in S$, we go through each hash $h_i = h_1, h_2, \ldots, h_k$, and set $A[i] = 1$. Figure 1 shows an example result of this process.

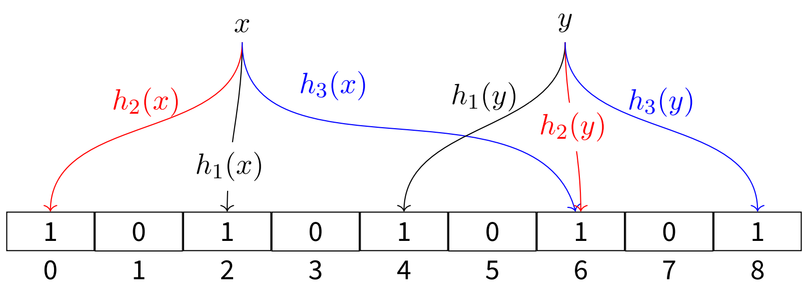

Figure 1: Inserting two elements x and y into a Bloom filter with $\varepsilon = 1/8$. We have three hash functions, and (rounding up) the array is of length $m = 9$ bits. Each hash value is shown by arrows; for example $h_2(x) = 2$. Note that $h_3(x) = 6$ and $h_2(y) = 6$, so when setting the bits for y, we already have $A[h_2(y)] = 1$; in this case setting $A[6] = 1$ does not change the data structure.

storing a dictionary of all English words in ASCII was completely infeasible. For example, the 2019 Scrabble dictionary requires 3MB just to list all the words. Whereas the Macintosh II, which did not come out until 17 years after the Bloom filter was invented, came with 1MB of RAM and a 40MB hard disk.

This ensures that the Bloom filter satisfies the following crucial invariant.

Invariant 1. A Bloom filter storing a set S using hashes $h_1, \ldots h_k$ satisfies $A[h_i(x)] = 1$ for all $x \in S$ and all $i \in \{1, \ldots, k\}$.

Querying a Bloom filter. For a query q, we go through each hash $h_i = h_1, h_2, \ldots h_k$. If $A[h_i(q)] = 0$ for any h_i, the Bloom filter returns "$q \notin S$." Otherwise, we have $A[h_i(q)] = 1$ for all h_i, and the Bloom filter returns "$q \in S$." Two examples of this can be seen in Figures 2 and 3 below.

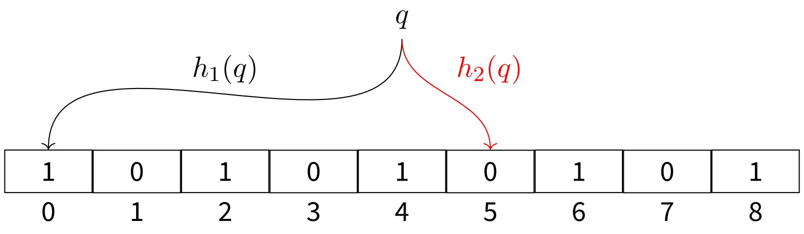

Figure 2: An example query to an element not in the set. Since $A[h_2(q)] = 0$, the Bloom filter returns "$q \notin S$."

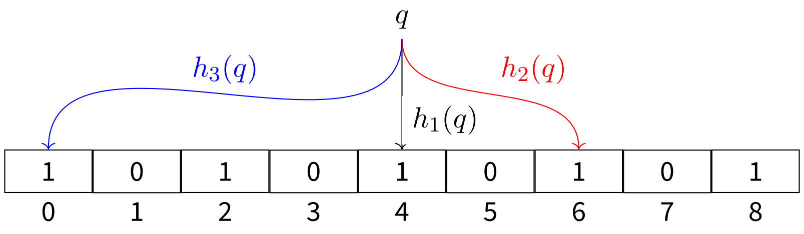

Figure 3: An example false positive query. Since $A[h_i(q)] = 1$ for all i, the Bloom filter returns "$q \in S$." Comparing to Figure 1, we see that q is not actually in S; q is a false positive.

Discussion. Notice that since we build the Bloom filter by hashing items one by one, we can easily extend this strategy to insert new elements into S (so long as no more than n items are stored in total).

However, we cannot delete items, as doing so could inadvertently cause false negatives (by violating Invariant 1). For example, in Figure 1, if we were to delete y by setting $A[h_i(y)] = 0$ for all $i \in \{1, 2, 3\}$, then we would have $A[6] = 0$. But that means that if we queried for x (which is still in S), the Bloom filter would see that $A[h_3(x)] = 0$, and incorrectly answer "$x \notin S$".

Analysis

For this analysis, we will assume that all hash functions h_i are completely random: any $x \in U$ is mapped to any hash slot $s \in \{0, \ldots, m-1\}$ with probability $1/m$. We will discuss in class that implementations like Murmurhash or cryptographic hash functions (though not all hash implementations), generally behave like this in practice.

First, let's consider the first guarantee: that if $q \in S$, the Bloom filter returns "$q \in S$" correctly. By Invariant 1, if $q \in S$, then $A[h_i(q)] = 1$ for all $i \in \{1, \ldots k\}$. This means that the query algorithm always returns "$q \in S$."

Now let's consider the false positive rate. I am not going to give a formal proof in these notes; instead I will give the high-level form of this argument. A full analysis of this data structure is surprisingly difficult due to the fact that the value stored in each bit of A is not quite independent: for example, if $A[1] = 1$, then the probability that $A[2] = 1$ is slightly smaller. The recommended course textbook on probability contains the full analysis [4, pages 109–111].

Whether or not a query q is positive is determined by which bits in A are set. So let's begin by analyzing that, for some i, $A[i] = 1$. A bit of A is 0 if no element hashes to it. An element hashes to a particular bit with probability $1/m$; so *no* element hashes to a given bit with probability $(1 - 1/m)^{nk} \approx 1/e^{nk/m} = 1/e^{1/\ln 2} = 1/2$. Therefore, $A[i] = 1$ with probability approximately $1/2$.

As mentioned above, for any i, j, the event that $A[i] = 1$ and the event that $A[j] = 1$ are not quite independent—but for simplicitly let's assume that they are.[5] Then, the probability that for all $i \in \{1, \ldots, k\}$, $A[h_i(q)] = 1$ is $1/2^k = 1/2^{\log_2 1/\varepsilon} = 1/(1/\varepsilon) = \varepsilon$. This gives us a false positive rate of at most ε.

Cuckoo Filter

The cuckoo filter is a more recent method of creating a filter, invented by Fan et al. in [3]. In this section we start by describing and analyzing the cuckoo filter. We will discuss the advantages and disadvantages of the cuckoo filter vs the Bloom filter in Section .

Rather than a bit array (as used in the Bloom filter), the cuckoo filter uses a hash table with properties similar to cuckoo hashing.

A cuckoo filter consists of k hash functions[6] denoted by $h_1, h_2, \ldots, h_k$, a fingerprint hash function f, and a hash table T of m slots, where each slot has room for $\log_2(1 + 1/\varepsilon)$ bits. (We will assume that

[5]The textbook analysis does not make this assumption.

[6]Unlike in a Bloom filter, k will be a small constant that does not depend on ε. We'll usually use $k = 2$. Furthermore, we will usually be using partial-key cuckoo hashing, in which case we only need one of these hashes.

ε is set so that $\log_2(1+1/\varepsilon)$ is an integer.) For now, let's have $k = 2$ hash functions, and $m = 2n$ slots. These parameters correspond to the cuckoo hash table we discussed in class. As mentioned, with these parameters, inserting a new element in the table takes $O(1)$ expected time, and $O(\log n)$ time with probability at least $1 - 1/n^2$, and a rebuild happens with probability at most $1/n^2$ [5].

The hash functions $h_1, \ldots, h_k$ take in an element from the universe U and output a number from 0 to $m-1$. The fingerprint hash takes in an element from U and outputs a number from 1 to $1/\varepsilon$; it cannot output 0. Let's assume that $1/\varepsilon + 1 = 2^f$ for some integer f; then the fingerprint can be stored in f bits.

Inserting into a Cuckoo Filter To begin, we mark each slot of T as empty by storing a value of 0. To insert an element x, we check to see if $T[h_1(x)]$ is empty, i.e. it stores the value 0. If it is, we store $f(x)$ in $T[h_1(x)]$. Otherwise, we check $T[h_2(x)]$, and so on until we finally check $T[h_k(x)]$. (Note that $f(x) > 0$, so an empty slot always stores 0, and a nonempty slot always stores a number in $\{1, \ldots 1/\varepsilon\}$.)

If $T[h_i(x)]$ is not empty for any i from 1 to k, then we cuckoo: choose some $i \in \{1, \ldots, k\}$.[7] Let's say that x_1 is the element stored in $T[h_i(x)]$. Then we store $f(x)$ in $T[h_i(x)]$ and "cuckoo" x_1 to another slot: if there is an $i_1 \in \{1, \ldots k\}$ with $i_1 \neq i$, then we store $f(x_1)$ in $T[h_{i_1}(x_1)]$. Otherwise, $T[h_{i_1}(x_1)]$ is full for all i_1; again we pick an $i_2 \in \{1, \ldots k\}$, and we "cuckoo" the element stored in $T[h_{i_1}(x_1)]$ to another slot, leaving room to store $f(x_1)$ in $T[h_{i_1}(x_1)]$.

If we cuckoo more than $O(\log n)$ elements during a single insertion, we rebuild the filter. The classic cuckoo hashing analysis shows that, after n inserts, we only need to rebuild with probability $O(1/n)$.[8]

Example inserts are given in Figures 4 and 5.

Invariant 2. For every $x \in S$, there exists an $i \in \{1, \ldots, k\}$ such that $f(x)$ is stored in $T[h_i(x)]$.

Partial-key Cuckoo Hashing. We cheated in the above insertion algorithm! When all of the slots for x are full, I want to move x_1 to another slot to make room for x. But this is a filter: I don't know what x_1 is; all I know is that there are some fingerprint bits stored in $T[h_i(x)]$. How can I find the other slot for x_1?

[7] It doesn't matter how you choose this first i. You can always choose $i = 1$, or you can choose it at random.

[8] The detailed analysis of this is outside the scope of this class (though I recommend [6] for an excellent, though very short, explanation using only information-theoretic arguments). But here is some intuition for the case with $k = 2$ and $m = 2n$. Since $k = 2$, to "cuckoo" an element we move it to its opposite slot. Since half of the slots are full, this opposite slot is full with probability $1/2$. The probability that we see k elements without finding an empty slot is then $1/2^k$, and we have $1/2^{\log n} = 1/n$. Note that, as in the Bloom filter case, I am incorrectly treating these probabilities as independent—this argument is much more a mnemonic for the correct bound than a proof.

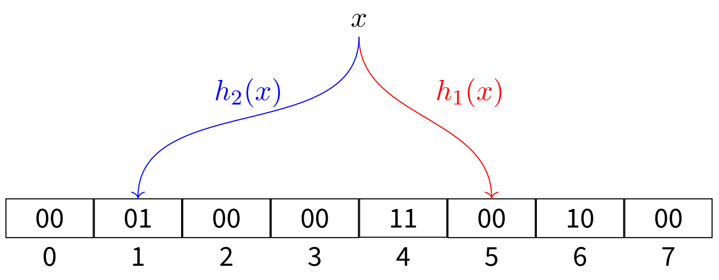

Figure 4: A cuckoo filter with $\varepsilon = 1/3$ and $k = 2$. Each slot has room for $\log_2(1 + 1/\varepsilon) = 2$ bits. Since slot $h_1(x) = 5$ does not have a stored fingerprint, we can store $f(x)$ in slot 5.

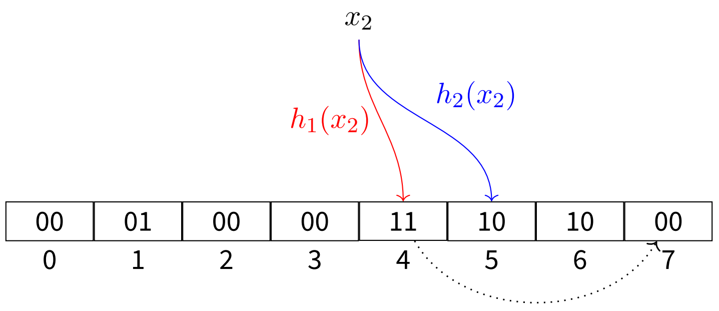

Figure 5: A cuckoo filter with $\varepsilon = 1/3$ and $k = 2$. Each slot has room for $\log_2(1 + 1/\varepsilon) = 2$ bits. In this example both slots are full, so we "cuckoo" a fingerprint the fingerprint in slot $h_1(x_2) = 4$ to its other slot. We use partial-key cuckoo hashing to calculate the new slot using an XOR: $4 \wedge h(11) = 7$ (in this case, $h(11) = 3$). After moving the fingerprint, 11 will be stored in slot 7, leaving room for $f(x_2)$ to be stored in slot 4.

If $k = 2$ and m is a power of 2, there is a nice trick that works well in practice, called "partial-key cuckoo hashing." Rather than two hashes $h_1(x)$ and $h_2(x)$, we use one hash $h_1(x)$, plus an additional hash h that maps a *fingerprint* to a value in $\{1, \ldots m - 1\}$. Then we set $h_2(x) = h_1(x) \wedge h(f(x))$ (where $\wedge$ means XOR, as in C).[9] For this strategy to work, we need $h(f(x))$ to never be zero. (Otherwise, an element may have both hashes map to the same slot, which creates issues.)

One benefit of this strategy is that XOR is its own inverse. That is to say, for any a and b, $a \wedge b \wedge b = a$. This means that $h_1(x) = h_2(x) \wedge h(f(x))$.

Thus, if we need to "cuckoo" an element stored in slot s, we take the $\log_2(1 + 1/\varepsilon)$ bits stored in that slot (let's call these bits f_s), and XOR them to obtain a new slot $s' = s \wedge h(f_s)$. If s' is empty we store f_s in s'; otherwise we cuckoo s' using the same method.

[9] In [2], Eppstein shows that omitting h, and instead taking an XOR with the fingerprint itself, does not affect the theoretical performance. But in my experience, this leads to serious issues with the parameters we will use in the assignment.

Querying a cuckoo filter. To query an element q, we go through each hash $h_i = h_1, h_2, \ldots h_k$. If $T[h_i(q)] = f(q)$, the cuckoo filter returns "$q \in S$." Otherwise, $T[h_i(q)] \neq f(q)$ for all h_i, in which case the filter returns "$q \notin S$."

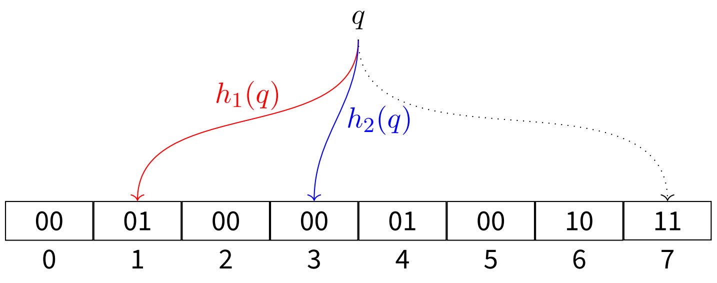

Figure 6: Querying a cuckoo filter with $\varepsilon = 1/3$ and $k = 2$. Let's say that $f(q) = 11$. Neither slot hashed to by q contains 11, so the filter returns "$q \notin S$." If q had hashed to slot 7 (say, if $h_1(q) = 7$), then the filter would have returned "$q \in S$".

Discussion. As implied above, a cuckoo filter can handle inserting new elements (so long as no more than n elements are stored in the filter simultaneously—if too many elements are stored, we are likely to be forced to rebuild frequently). Unlike a Bloom filter, a cuckoo filter can also handle deletes under certain circumstances (we won't discuss these circumstances in this class).

Analysis

For this analysis, we will assume that all hash functions h_i are completely random: any $x \in U$ is mapped to any hash slot $s \in \{0, \ldots, m-1\}$ with probability $1/m$. Furthermore, we will assume the same of the fingerprint hash f: any $x \in U$ is mapped to a given fingerprint $f_x \in \{1, \ldots, 1/\varepsilon\}$ with probability ε. We will soon discuss in class how many practical (though not all!) implementations, like Murmurhash or cryptographic hash functions, generally behave like this in practice.

Aside: Union Bound. The union bound is a very simple but very useful bound in analyzing randomized algorithms. Like linearity of expectation, it has the benefit of always working: even for events that are not independent.

Theorem 1. Let X and Y be random events. Then

$$\Pr(X \text{ or } Y) \leq \Pr(X) + \Pr(Y).$$

More generally, if $X_1, X_2, \ldots, X_k$ are any random events, then

$$\Pr(X_1 \text{ or } X_2 \text{ or } \ldots \text{ or } X_k) \leq \sum_{i=1}^{k} \Pr(X_k).$$

The union bound gives a *simple upper bound* on the probability of an event. It can save us a huge amount of work compared to a more exact analysis!

Here's one example. Let's say I have 10 students in a course, and I randomly assign each student an ID between 1 and 100, each with probability $1/100$ (these IDs do not need to be unique). Can you upper bound the probability that some student has ID 1?

First, let's give an exact analysis. (The idea of this example is that this analysis is a little difficult, but gives an exact answer—the union bound method will be much easier.) The probability that at least one student has ID 1 is

$$1 - \Pr(\text{no student has ID 1}).$$

The probability that a single student has an ID other than 1 is $99/100$. Thus, the probability that all 10 students have an ID other than 1 is $(99/100)^{10}$. Thus, the probability that at least one student has ID 1 is $1 - (99/100)^{10}$. Plugging this into a scientific calculator, we obtain a probability of 9.56%.

Now, let's do the same analysis with the union bound. The probability that a given student has ID 1 is 1/100. Thus, the probability that any student has ID 1 is the sum, over all 10 students, of 1/100. This gives us a value of $10/100 = 10\%$.

The union bound analysis was much simpler, but gave almost the same result! (Union bound will not always give such a close approximation of the true result—it does best with events that occur with fairly low probability.)

Meeting the Filter Guarantees. First, let's consider the first guarantee: that if $q \in S$, the cuckoo filter returns "$q \in S$" correctly. By Invariant 2, if $q \in S$, then there exists a hash h_i such that $T[h_i(q)] = f(q)$. This means that the query algorithm will always correctly answer "$q \in S$."

Now, let's consider the false positive rate. A query $q \notin S$ is a false positive if, for some h_i, $T[h_i(q)] = f(q)$. Let's examine each hash h_1 and h_2 individually.

Let's start with h_1. Since we are storing n elements in $2n$ slots, the probability that $T[h_1(q)]$ contains a fingerprint is $1/2$. If $T[h_1(q)]$ contains the fingerprint of some element x, the probability that $f(x) = f(q)$ is ε. Therefore, the probability that $T[h_1(q)]$ contains a fingerprint $f(x) = f(q)$ is $\varepsilon/2$.

Exactly the same analysis holds for h_2: the probability that $T[h_2(q)]$ contains a fingerprint $f(x) = f(q)$ is $\varepsilon/2$.

A query q is a false positive if $T[h_2(q)]$ contains $f(q)$, OR if $T[h_1(q)]$ contains $f(q)$. Therefore, we can use the union bound to say that q is a false positive with probability $\varepsilon/2 + \varepsilon/2 \leq \varepsilon$.

Improving the Cuckoo Filter. You may have noticed that the above analysis has total space usage $2n \log_2(1 + 1/\varepsilon)$. This is worse than the Bloom filter, which requires $1.44n \log_2(1/\varepsilon)$ bits of space!

Further theory and experiments (i.e. in [3]) have shown that if we set $k = 4$, we obtain good performance even if the hash table T has only $1.05n$ slots, so long as we extend the fingerprints to have $\log_2(1 + 4/\varepsilon)$ bits. This gives a total of $1.05n \log_2(1 + 4/\varepsilon) \approx 1.05n \log_2(1/\varepsilon) + 2n$ bits of space.

An alternate method, which we discussed in class, extends the hash table so that there is room for *four* fingerprints in each hash slot (while still keeping $k = 2$). Insertion proceeds largely as before: we only cuckoo if both hash slots contain two fingerprints. The invariant is that, for each $x \in S$, at least *one* of the fingerprints in one of the hash slots must be $f(x)$. In this case, we need to increase the length of the output of f to be $\log_2(1 + 8/\varepsilon)$ bits.[10] Now, we can set $m = 1.05n/4$ for total space usage $1.05n/4 \times (4 \log_2(1 + 8/\varepsilon)) \approx 1.05n \log_2(1/\varepsilon) + 3.15n$ bits.

Note that while the above analysis seems to indicate that using slots to store more elements, rather than just setting $k = 4$, seems like a bad idea (it's more complicated and takes up more space), it's actually fairly popular in practice. Part of the reason for this is that hashing is expensive: it's a big cost to hash each item four times instead of just twice (plus the fingerprint on top). It's also unclear how to make partial-key cuckoo hashing work with $k = 4$, while with $k = 2$ it works immediately.

Comparison of the Two Filters

To begin, let us look at what happens each time we make a query. In a Bloom filter, we must hash the query $\log_2(1/\varepsilon)$ times, and see if the corresponding bit is set to 1. In a cuckoo filter, we must calculate the fingerprint of the query and then hash the query twice, and see if the fingerprint stored at the corresponding table entry is equal to the query's fingerprint.

What does this mean in terms of cost? In both cases, testing the actual table entry is cheap (it's just an equality test). There are two costly aspects: hashing the query, and the cache misses incurred when accessing the table. The Bloom filter performs $\log_2(1/\varepsilon)$ hashes and likely[11] will incur $\log_2(1/\varepsilon)$ cache

[10] In short: we can repeat the same analysis as in the two-hash version. But now, for a query q, $f(q)$ is compared to 8 different slots in the hash table. Taking the union bound over the 8 slots, the probability of a false positive is at most $8 \cdot 1/2^{\log_2 1+8/\varepsilon} \leq \varepsilon$.

[11] I say "likely" because it is possible that some of the bits we look at are very close to each other. But for most elements, each hash will require a cache miss.

misses; meanwhile the cuckoo filter performs 3 hashes and incurs 2 cache misses. This means that the cuckoo filter is substantially more efficient for each query, particularly for small ε.

And yet, the Bloom filter is still almost certainly the more popular of the two filters we learned about today, despite the extra cost mentioned above. The main reason why is simplicity. A Bloom filter only requires a bit array along with some hash functions. Even on a limited system it can be implemented easily. And, many implementations already exist. The cuckoo filter is simple enough to be efficient when implemented, but requires a little more work: we need to allocate a hash table (rather than just a bit array), and we need to write a method to efficiently "cuckoo" elements around the table.

Looking at the space bounds alone, we immediately get a further tradeoff: the Bloom filter requires $1.44n \log_2(1/\varepsilon)$ bits of space, whereas the cuckoo filter requires $1.05n \log_2(1/\varepsilon) + 3.15n$ bits of space. This means that the Bloom filter is more space efficient if $\varepsilon > 2.85\%$; otherwise the cuckoo filter is more space efficient.

So which should we use? If performance is not too important—you just want a filter that works—then a Bloom filter is probably good enough for your use case. But if query performance is important, a cuckoo filter may be worth implementing, as it hashes fewer times and incurs fewer cache misses. Finally, if space is very important and ε is much less than 2.85%, using a cuckoo filter may give significant savings over the space required by a Bloom filter.

So as with many of the algorithms we've learned in this class, while a simple version is likely good enough for many use cases, a more involved algorithm can eke out significant gains—when the extra complexity is worth it.

References

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

[2] David Eppstein. Cuckoo filter: Simplification and analysis. In *Scandinavian Symposium and Workshops on Algorithm Theory (SWAT)*, volume 53, pages 8:1–8:12, 2016.

[3] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *International Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.

[4] Michael Mitzenmacher and Eli Upfal. *Probability and computing: randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[5] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[6] Mihai Pătraşcu. Cuckoo hashing. http://infoweekly.blogspot.com/2010/02/cuckoo-hashing.html, February 2, 2010. Unpublished blog post.