CMSC 420: Lecture 20
Bloom Filters

Filtering: In previous lectures, we have discussed randomized algorithms, that is, algorithms that use a random number generator to compute results. In those data structures (skip lists and treaps), the random number generator affected the running time, but the answers were always correct. Today, we will study a data structure in which the random choices may actually affect the correctness of the answer. We can make the probability of an erroneous answer as small as we like (with a modest increase in the running time).

The data structure we will consider is used in an application called *filtering*. Abstractly, imagine that we are given a large set X of n items drawn from a *universe* U. We want to answer set-membership queries of the form, "Is a given element x in X?". There are many applications of this data structure. Here are a couple:

- The head of IT at your new company tells you that too many users are picking passwords that are easy to crack (like "password123"). You are given a list of one million easily breakable passwords. When a user creates a new password, you want to check whether it is in this list, and if so, tell them to pick again.
- Search engines (Google Chrome or Microsoft Bing) maintain a large list of URL's of malicious sites used in cyber-attacks. You are given a large list of URL's of known malicious sites, and you need to check each connection request against this list.

Of course, this is just a special case of the dictionary problem. (We will allow insertions into X, but no deletions. Also, there are no associated values, just keys. We just want a "yes/no" answer.)

We know many efficient dictionary data structures. For example, hashing takes $O(n)$ words of space and $O(1)$ (expected) query time. Let us imagine, however, that n is very large, and we would like something much more space-efficient. Our goal will be a data structure that uses only around $O(n)$ *bits*, rather than $O(n)$ *words*.

False Positive/Negative: A *false positive* is when you erroneously report that $x \in X$, even though it isn't. A *false negative* occurs when you report that $x \notin X$, even though it is. False negatives are potentially very costly (allowing a user to use a weak password or allowing a connection to a malicious site).

We will present a new randomized data structure for this problem, called a *Bloom filter*. This data structure is very fast, it never suffers from false negatives, but it may (with low probability) suffer a false positive. That is, if $x \in X$, we are guaranteed to say "yes." If $x \notin X$, we will almost always say "no," but may occasionally say "yes" by mistake.

Bloom Filter: This data structure was invented in 1970 by Burton Howard Bloom. Recall that we wish to store a set $X \subseteq U$, where $|X| = n$. Our approach will be to use a collection hash functions, but unlike standard hashing, when collisions occur, we do not bother to resolve them. Formally, given a pair of positive integers m and k (to be determined later) a *Bloom filter* consists of a bit array `B[0..m-1]` together with k hash functions $h_1, \ldots, h_k$. Each hash function maps an element of U to an index $\{0, 1, \ldots, m-1\}$ into the array B. We'll denote the value of the ith hash function on argument x as $h(i, x)$.

The algorithm is really amazingly simple. Initially, all the bits of `B` are set to 0. To insert an element, we apply each of our k hash functions, and set all of these bits to 1. When we wish

to find an element, we repeat the hashing process. We know that if it was inserted, then each of the k hash entries must be 1. So, if even one of these has 0, we know (for sure) that x is not in the set. See the code block below and the examples in Fig. 1.

Bloom Filter Operations

```
void insert(Key x) {                        // insert x into the Bloom filter
    for (i = 0; i < k; i++) {               // hash it k times
        B[h(i,x)] = 1                       // ...and set each bit to 1
    }
}

boolean find(Key x) {                       // find x in the the Bloom filter
    for (i = 0; i < k; i++) {               // hash it k times
        if (B[h(i,x)] == 0) return false // failed even once? - not found
    }
    return true                             // succeeded all k times
}
```

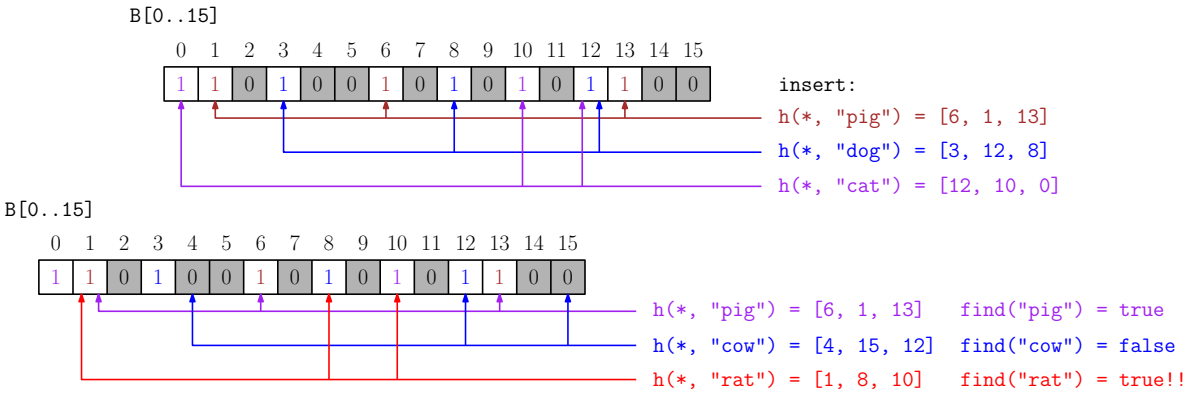

Fig. 1: Insertion and finding in a Bloom filter with $m = 16$ and $k = 3$.

A purely practical point to make is that the k hash functions do not depend on each other. So, if you are running on a typical modern machine with multiple cores, they can be computed in parallel. (This is not true for almost all of the data structures we have seen so far. Each step follows in sequence from the previous.)

Partial Correctness: Consider what happens during `find(x)`. If x was already inserted, we set all k of its hashed bits to 1, and when we search again, they will still be 1, so we are guaranteed to report that x is present. (No false negatives.)

On the other hand, if all k hashed entries contain a 1, it might indeed be the case that x was inserted, and this is why all these bits are set. But, it could also have happened by coincidence. When other keys were inserted, they happened to hit all k of these bits. As a result, we will incorrectly report that x is present when it really is not. (We may have a false positive.)

Controlling the False-Positive Rate: It is not surprising that the probability that things go wrong will depend on the values of k and m. As m gets larger, it becomes less likely that two hash functions will accidentally collide, but this increases our space. As k gets larger, it

becomes less likely that two different keys will coincidentally hit the same bit positions, but we fill in more entries of `B` with 1's, and so coincidental matches are more likely. Need to find a good balance.

The analysis will require that we delve into a bit of probability theory. We will make use of a few basic facts:

- If $|z|$ is small $1 + z \approx e^z$.
- If ℓ independent events each have a probability p of occurring, then the probability that they all occur is p^ℓ.

Let $n = |X|$. Let's make the assumption that our hash functions are mutually independent and uniformly random. The probability of hitting any one of the m bits of the B vector is exactly $\frac{1}{m}$, and therefore the probability of missing it is $1 - \frac{1}{m}$. So, for any $x \in U$ and any i and j, where $1 \le i \le k$ and $0 \le j \le m - 1$, we have

$$\Pr[h(i, x) \neq j] \;=\; 1 - \frac{1}{m}.$$

Suppose that after inserting all of our keys, we find that some entry of the bit vector is 0. This means every one of kn attempts at hitting it failed. (There are n keys inserted and each applies k different hash functions.) Clearly, this occurs with probability

$$\Pr[B[j] = 0] \;=\; \left(1 - \frac{1}{m}\right)^{kn}.$$

We may assume that m is large, meaning that $\frac{1}{m}$ is small, an hence $1 - \frac{1}{m} \approx e^{-1/m}$. Therefore

$$\Pr[B[j] = 0] \;\approx\; (e^{-1/m})^{kn} \;=\; e^{-kn/m}.$$

Since we will use this value repeatedly, let's define $p = e^{-kn/m}$. (Later, we'll discover that the best choice is $p = 1/2$, so think of it intuitively as just a coin flip.)

Since the probability that any bit of B is zero is p, it follows that the expected number of 0-bits is just mp. We will employ (without proof) an important fact from probability theory, which states that when dealing with a large number of independent trials, the actual number of times an event occurs is very close to its expected value. (This can be formally quantified using something called the *Chernoff bounds*. Since this is not a theory course, we'll just keep things simple, and assume the "ideal case" that the number of 0-bits in B is exactly mp.)

So, what is the probability of a false positive when we search for a key x? In order for a false positive to happen, it must be the case that all of the bits that x hashes to, that is, $h(1, x), \ldots, h(k, x)$, have already all been set to 1 by the other keys. Since we have $B[j] = 0$ with probability p, we have $B[j] = 1$ with probability $1 - p$. In order to get a false positive, this unfortunate event must reoccur k times in a row. Each of these is an independent event. Therefore, the probability of this happening is the k-fold product of $(1 - p)$. Letting $\Pr[\text{FP}]$ denote the probability of a false positive, we have

$$\Pr[\text{FP}] \;=\; (1 - p)^k.$$

Let's assume that n and m are fixed (based on the amount of storage we can devote, given the size of our set). The question is what is the best choice for k to minimize the false-positive

rate. We claim that this will happen when $p = 1/2$. To show this, let's take the logarithm of this quantity. (Remember the that log function is monotonically increasing, so minimizing the log is equivalent to minimizing the original.)

$$\ln(\Pr[\text{FP}]) \;=\; \ln(1-p)^k \;=\; k\ln(1-p).$$

Given our definition p, we can equivalently express $\ln p = -k(n/m)$, or in other words, $k = -(m/n)\ln p$. Plugging this in, we find that the log of the false-positive probability is

$$\ln(\Pr[\text{FP}]) \;=\; -\frac{m}{n}\ln p \cdot \ln(1-p).$$

Since we assume that n and m are fixed, we just need to minimize the quantity $-\ln p \cdot \ln(1-p)$. Observe that this quantity is symmetric about $p = 1/2$. It stands to reason that the minimum value will be achieved at the symmetric point where $p = (1-p) = 1/2$. (This is not immediately obvious. To be rigorous, you should take the derivative with respect to p and set it to zero. Alternatively, just enter the function into any online graphing software, and you'll see right away that it is minimized at $p = 1/2$.)

By setting $p = 1/2$ and solving, we conclude that (for fixed values of n and m) the optimum number of hash functions is $k = (\ln 2)(m/n)$. This gives a false positive rate of

$$\Pr[\text{FP}] \;=\; (1-p)^k \;=\; \left(\frac{1}{2}\right)^{(\ln 2)(m/n)} \;\approx\; (0.61850)^{m/n}. \tag{1}$$

Adjusting the Bit-Vector Size: Rather than figuring out the false positive rate, suppose that the data structure designer tells us that they can tolerate a false-positive probability of $\delta > 0$. Solving Eq. (1) for m, we have the following.

$$m \;=\; \left\lceil \frac{\lg(1/\delta)}{\ln 2} n \right\rceil \;=\; O(n\log(1/\delta)).$$

Setting the table size to this value guarantees the desired false-positive rate. So, for example, if we wanted to achieve a 1% false-positive rate for a set of size n, we could do this with a Bloom filter involving $10n$ bits and 7 hash functions. (Assuming that each key is a 32-bit word, this is much smaller than the space needed to store the individual keys explicitly.)