CMSC 420: Bloom Filters

Justin Wyss-Gallifent

December 1, 2022

1 Introduction .............................................. 2
2 Definition .................................................. 2
3 Insertion ..................................................... 3
   3.1 Algorithm ............................................. 3
   3.2 Time Complexity ...................................... 3
4 Search ....................................................... 4
   4.1 Algorithm ............................................. 4
   4.2 Time Complexity ...................................... 4
5 Why No Deletion? .......................................... 4
6 Math! .......................................................... 5
   6.1 Probability of a False Positive ....................... 5
   6.2 An Approximation Formula ............................. 6
   6.3 Minimizing False Positives ............................ 7
7 Tuning and Rebuilding ...................................... 8
1 Introduction

All of our data structures so far have been exact, meaning that they accurately reflect the truth of whether an item is or is not in the set. Oftentimes however it’s sufficient to sacrifice a bit of precision for speed. Probabilistic data structures generally do exactly this, and Bloom filters are one of the most used of these.

The reason they are called “filters” is that their primary use is as a filter before some other process, but they are in fact structures which contain data.

2 Definition

Loosely speaking a Bloom filter is a method of storing data such that search and insert are very fast. If you notice that delete is not included you’d be right. We can’t delete from a bloom filter!

The trade-off is that search is imprecise in the following sense:

• If the item is in the set then search will tell us that it is. It will not lie - there are no false negatives.

• If the item is not in the set then there is a small chance that search will tell us that it is. In other words there might be false positives.

You might wonder why this might be useful, or even acceptable, so here are a few examples:

• Suppose a system needs to check if a URL is a threat and therefore needs extra security checks. We’d prefer to get “yes” answers very quickly and a small number of false positives are acceptable. Chrome uses this.

• Suppose a database needs to check if a password is weak and inform the user that they should change it. We’d prefer to get “yes” answers very quickly and a small number of false positives are acceptable.

Definition 2.0.1. Given a set $S$ of keys, a Bloom filter is composed of a bit array $B$ composed of $m$ bits (think of a list indexed $B[0]$ through $B[m-1]$ but we’ll just write a string) and a set of $k$ hash functions $h_1,...,h_k : K \rightarrow \mathbb{Z}_m$. These hash functions do no sort of collision management, they are basically just very fast functions which behave as “randomly” as possible.
3 Insertion

3.1 Algorithm
When a key $x$ is to be inserted we set:

\[ B[h_1(x)] = ... = B[h_k(x)] = 1 \]

**Example 3.1.** Here is a really trivial example. Suppose our set of keys is $\mathbb{Z}_{10}$, our bit array has 20 bits, hence is indexed $B[0]$ through $B[19]$, and we use three hash functions:

\[
\begin{align*}
  h_1(x) &= x \mod 20 \\
  h_2(x) &= 3x \mod 20 \\
  h_3(x) &= 7x \mod 20
\end{align*}
\]

The bit array starts as:

\[ B = 000000000000000000000000 \]

To insert the key $x = 1$ we calculate $h_1(1) = 1$, $h_2(1) = 3$, and $h_3(1) = 7$ and so we assign bits 1,3,7 to 1:

\[ B = 010100010000000000000000 \]

To insert the key $x = 4$ we calculate $h_1(4) = 4$, $h_2(4) = 12$, and $h_3(4) = 8$ and so we assign bits 4,2,8 to 1:

\[ B = 010110010100100000000000 \]

To insert the key $x = 7$ we calculate $h_1(7) = 7$, $h_2(7) = 1$, and $h_3(7) = 9$ and so we assign bits 7,1,9 to 1. Note that 7 and 1 were already set:

\[ B = 010110010100100000000000 \]

3.2 Time Complexity
Since $k$ is fixed the calculation is really only dependent on the hash functions. Since hash functions are typically very fast this then implies that insertion is, too.

If we want a \( \Theta \) time complexity then we can overlook the hashing speed and say that the time complexity of insertion as a function of the number of elements $n$ in the Bloom filter is \( \Theta(1) \).
4 Search

4.1 Algorithm

To search for a key \( x \) we simply check if:

\[
B[h_1(x)] = ... = B[h_k(x)] = 1
\]

We return “yes” if so.

\textbf{Note 4.1.1.} Notice that it’s entirely possible that for a key \( x \) that the bits \( B[h_1(x)], ..., B[h_k(x)] \) might equal 1 because they were turned on for other keys. Such an event would yield a false positive.

On the other hand a key is certainly not in the Bloom filter iff at least one if \( B[h_1(x)], ..., B[h_k(x)] \) equals 0. Thus there are no false negatives.

\textbf{Example 4.1.} Following off the previous example to check if the key \( x = 2 \) is in the set we observe:

\[
\begin{align*}
B[h_1(2)] &= B[2] = 1 \\
B[h_2(2)] &= B[6] = 0 \\
B[h_3(2)] &= B[14] = 0
\end{align*}
\]

Since at least one of them is 0 we know that \( x = 2 \) is not in the set. This “no” is definite.

To check if the key \( x = 4 \) is in the set we observe:

\[
\begin{align*}
B[h_1(4)] &= B[8] = 1 \\
B[h_2(4)] &= B[12] = 1 \\
B[h_3(4)] &= B[8] = 1
\end{align*}
\]

Since all three are 1 we believe that \( x = 6 \) is in the set. Observe that this might be a false positive and we have no way of knowing.

\textbf{4.2 Time Complexity}

Same as for insertion.

5 Why No Deletion?

There is no deletion basically because the only reasonable way to delete would be to hash the key and then set those bits to 0. While this is possible, the issue with this is that it starts introducing false negatives because when deleting a key \( x \) we might zero out a bit for another key \( x' \) and then searching for \( x' \) would say it’s not in the set.
There are other probabilistic ways to manage this such as keeping a separate bloom filter of deleted keys but these can not only introduce further issues but slow down a data structure which we wanted to be fast.

6 Math!

6.1 Probability of a False Positive

The primary issue of course is the issue of false positives so let’s consider what the probability of false positives is.

We will assume the hash functions act “randomly” in the sense that taken over all possible $x \in S$ we have an equal chance of each $h_i(x)$ being each bit.

Suppose we insert one element. The probability of a hash function setting a specific bit to 1 is $1/m$ and hence the probability that a specific bit is not 1 is:

$$1 - \frac{1}{m}$$

If we have $k$ hash functions and they are independent from one another then the probability that a specific bit is not 1 is:

$$\left(1 - \frac{1}{m}\right)^k$$

If we insert $n$ elements then the probability that a specific bit is not 1 is:

$$\left(1 - \frac{1}{m}\right)^{kn}$$

Hence the probability that it is 1 is:

$$1 - \left(1 - \frac{1}{m}\right)^{kn}$$

For a given $x$ we get a false positive if all $B[h_1(x)] = \ldots = B[h_k(x)] = 1$. Since the probability that each is 1 is given above, and they are independent, then the probability that all of them are 1 is:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$$

Example 6.1. Suppose we use a bit array with $m = 200$ bits and $k = 5$ hash functions.

If we have inserted $n = 10$ keys into the Bloom filter then the chance of getting a false positive (say, in a query), equals:
\[ p = \left( 1 - \left( 1 - \frac{1}{m} \right)^k \right)^n = \left( 1 - \left( 1 - \frac{1}{200} \right)^{(5)(10)} \right)^5 \approx 0.000535433414 \]

So about 0.05\%, which is pretty low.

If we have inserted \( n = 100 \) keys into the Bloom filter then the chance of getting a false positive (say, in a query), equals:

\[ p = \left( 1 - \left( 1 - \frac{1}{m} \right)^k \right)^n = \left( 1 - \left( 1 - \frac{1}{200} \right)^{(5)(100)} \right)^5 \approx 0.65347 \]

So about 65.347\%, which is pretty high.

We could fix this by using more hash functions or a larger bit array.

\[
\text{Note 6.1.1. In the real world what generally happens is that we make a prediction on } n \text{ and a decision on } p \text{ and then we choose } m \text{ and } k \text{ in order to make this happen. These choices are typically system-related and constrained by how much memory we want to use for } B \text{ and how many hash functions we want to create. This is called tuning the Bloom filter.}
\]

\subsection*{6.2 An Approximation Formula}

There is a slick approximation for our formula. Recall from calculus that:

\[
\lim_{m \to \infty} \left( 1 - \frac{1}{m} \right)^m = \frac{1}{e}
\]

It follows that for large \( m \) that:

\[
\left( 1 - \frac{1}{m} \right)^m \approx \frac{1}{e}
\]

Thus for large \( m \) we have:

\[
\left( 1 - \frac{1}{m} \right)^k = \left( \left( 1 - \frac{1}{m} \right)^m \right)^{k/m} \approx \left( \frac{1}{e} \right)^{k/m}
\]

As a consequence:

\[
p = \left( 1 - \left( 1 - \frac{1}{m} \right)^k \right)^n \approx \left( 1 - \left( \frac{1}{e} \right)^{k/m} \right)^n \approx \left( 1 - e^{-kn/m} \right)^k
\]
6.3 Minimizing False Positives

Consider the slick approximation above:

\[ p \approx \left( 1 - e^{-kn/m} \right)^k \]

We can of course minimize false positives by increasing the value of \( m \) which is the size of \( B \). We see in the formula above that as \( m \) increases the probability decreases and limits to 0.

But what would happen if we fixed \( m \)? Is it possible to choose the number of hash functions \( k \) to minimize the probability of a false positive?

As long as we’re happy with approximations it can be shown via some rather messy calculus that the above formula is minimized when:

\[ k = \frac{m}{n} \ln 2 \]

Of course this is probably not an integer so we would choose either the floor or the ceiling.

Moreover for this value of \( k \) we get:

\[ p \approx \left( 1 - e^{-kn/m} \right)^k = \left( 1 - e^{-\ln 2} \right)^{\frac{m}{n}} \ln 2 = \left( \frac{1}{2} \right)^{\ln 2} \approx 0.002464755 \]

**Example 6.2.** Suppose we are storing \( n = 40 \) values in a Bloom filter and our bit array has \( m = 500 \) bits. We would obtain the minimum probability of false positives by using \( k \) hash functions where:

\[ k = \frac{m}{n} \ln 2 = \frac{500}{40} \ln 2 \approx 8.66 \]

So we might decide to use \( k = 8 \) or 9 hash functions. If we do so then we expect the probability of false positives to be:

\[ p \approx \left( \frac{1}{2} \right)^{500/40} \approx 0.002464755 \]

This is about 0.25% - really quite small!

**Note 6.3.1.** Note that our formula \( k = (m/n) \ln 2 \) could be solved for either \( m \) or \( n \). However typically \( n \) is not something we have control over in the real world and typically \( k \) is chosen after \( m \) because selection of hash functions depends upon knowing how large the bit array is. That is, we typically don’t say “we should use \( k \) hash functions, what should \( m \) be?” but we certainly could.
7 Tuning and Rebuilding

As mentioned earlier, typically when a Bloom filter is going to be implemented we should do our best to figure out an upper bound on the number of keys we’ll store in it (maximum $n$ value) and what our acceptable false positive rate $p$ is and then we choose $m$ and $k$ to make this happen.

Of course it may happen that we underestimate how large $n$ can get and then when it gets too large, $p$ gets above our acceptable level.

One solution would be to rebuild the Bloom filter using a larger upper bound for $n$, but this is expensive since it would require choosing a new $m$ and a new $k$, and therefore new hash functions, and re-inserting everything. As a consequence this should be a tactic of last resort. We should not, for example, treat rebuilding as part of the functioning of the structure.