CMSC 420: Bloom Filters

Justin Wyss-Gallifent

December 2, 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 Probability of a False Positive ............................ 5
7 Tuning and Rebuilding ..................................... 5
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 \( h_1, \ldots, h_k : K \to \mathbb{Z}_m \).

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:

$$h_1(x) = x \mod 20$$
$$h_2(x) = 3x \mod 20$$
$$h_3(x) = 7x \mod 20$$

The bit array starts as:

$$B = 00000000000000000000$$

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:

$$B = 01010001000000000000$$

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:

$$B = 01011001000000000000$$

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:

$$B = 01011001010010000000$$

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 $\Theta(1)$.  

3
4 Search

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

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

We return “yes” if so.

**Note 4.1.1.** Notice that it’s entirely possible that for a key \( x \) that the bits \( B[h_1(x)], \ldots, B[h_k(x)] \) 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 \( B[h_1(x)], \ldots, B[h_k(x)] \) equals 0. Thus there are no false negatives.

**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.

\[\blacksquare\]

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 Probability of a False Positive

In Bloom’s original paper he includes a calculation which reports that the probability of a false positive is:

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

This calculation has shown up all over the place since then and is taken as true, but is actually false. The calculation has an assumption It turns out that the proof of this calculation (which Bloom wrote and which exists all over the place) is false because of independence assumptions which are incorrect.

In reality the probability of a false positive is:

\[ p = \frac{1}{m^{k(n+1)}} \sum_{i=1}^{m} i^k \binom{m}{i} \left\{ \frac{kn}{i} \right\} \]

Here the expression \( \left\{ \frac{kn}{i} \right\} \) is the Stirling number of the second kind.

**Note 6.0.1.** The expression \( \left\{ \frac{a}{b} \right\} \) represents the number of ways to partition a set of \( a \) distinct objects into \( b \) subsets and is calculated via:

\[ \left\{ \frac{a}{b} \right\} = \frac{1}{b!} \sum_{i=0}^{b} (-1)^i \binom{b}{i} (b - i)^a \]

The formula Bloom gave (and others continue to give) actually slightly underestimates the probability. Interestingly people have noticed this discrepancy in real-world implementations but have generally believed the calculation and attributed the discrepancy to implementation imperfections.

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.