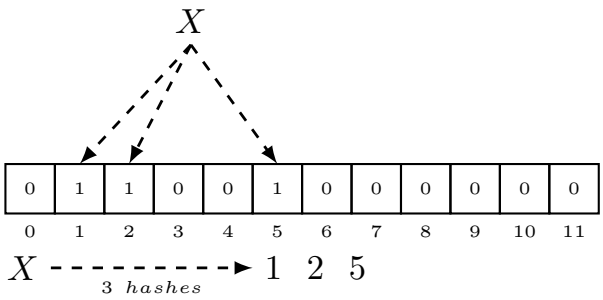

Figure 1: Inserting an element into a bloom filter.

Things are a bit different when we want to check if an element *is* present in the set. This is the kind of query that can result in a false positive.

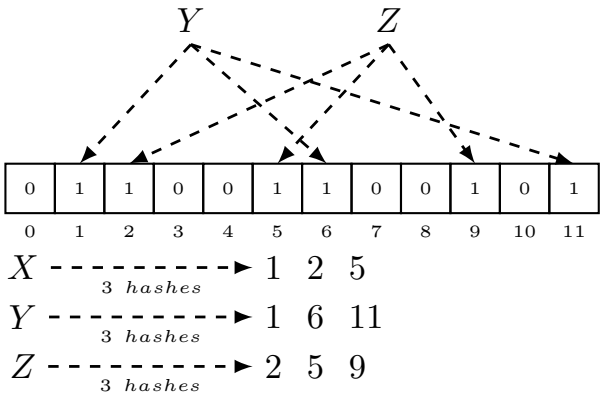

Figure 2: Inserting multiple elements into a bloom filter. "X" was never inserted, yet it appears as if it has, i.e. a false positive has occurred.

To better understand why this is the case, let's look at figure 2. We have element Y and Z and three hash functions to map values to the bloom filter. Let's assume that the resulting hashes for Y are $\{11, 6, 1\}$ and those for Z are $\{5, 2, 9\}$. We then switch all the bits of the given indices to one. If someone wants to verify if Y or Z are truly present in the set, one would get a positive result, this is because all the indices for Y and Z are one. There is however a problem with this. If there is another element, let's call it X, which happens to have $\{1, 5, 2\}$ as its hashes, one will get a false positive. The bits at $\{1, 5, 2\}$ are already one, but we never inserted X to the bloom filter. As a result there is no way for us to know if X really was inserted into

the bloom filter or not.

Fortunately one can lower the chances for false positives by tweaking the parameters of the bloom filter: m (the bloom filter size), n (the number of elements inserted into the bloom filter), and k (the number of hash functions used for the bloom filter). By knowing these values, we can compute the false positive rate of a bloom filter as:

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

Let's unpack this formula: $1 - \frac{1}{m}$ shows the probability that a given bit in the bloom filter is not switched to one by a hash function. $\left(1 - \frac{1}{m}\right)^k$ shows the probability that a given bit is still zero, given that k hashes can potentially point to it. And $\left(1 - \frac{1}{m}\right)^{kn}$ is the probability that a particular bit is still zero, after n elements were inserted. When we check for the presence of an element however, we do not look only at the value of one index, but at k. This is why our final formula for the false positive rate is $\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$, which gives the probability that k indices will be 1 after inserting n elements. Thus by choosing a proper m and k for a given n, one can design bloom filters with predetermined false positive rates.

Another important fact to consider, is that every hash function used has to be independent and its mappings uniformly distributed, otherwise we end up with non-random mappings in the bloom filter. The larger the bloom filter gets, the more hash functions we have to use. This can become a problem, as hash functions are computationally quite expensive and can slow down bloom filters. Because of this, people very often use faster non cryptographic hash functions, such as the Murmur non-cryptographic hash function to populate bloom filters. Distributed Bloom Filters on the other hand do store element hashes, but do not use hash functions for every element when populating bloom filters. This makes distributed bloom filters significantly faster to compute.

1.2. Set reconciliation via bloom filters

Set reconciliation is a problem in computer science where two nodes A and B, each holding a set of elements S_A and S_B, try to find the set difference $D_{A-B} = S_A - S_B$ and $D_{B-A} = S_B - S_A$ with as minimal traffic as possible. In other words, each node needs to figure out which elements of the other

node it does not have, without having to send, or ask for the whole set of elements. Especially in peer-to-peer networks, if a node already has a subset of the elements from another node, it is desirable to ask somehow only for the missing elements.

Nodes in a distributed system often have to synchronize states with one-another. Bloom filters are a great way for achieving such set reconciliation, as they require minimal space for transmission. Instead of sending a whole set to another node for set reconciliation, one can send a very small bloom filter. The node that receives the bloom filter can then create a bloom filter with its own elements, and reduce its own bloom filter with the one it received from its peer, as shown in figure 3. If any of the new values are -1, or in the negative range in general (in the case of counting bloom filters), then the node knows that the other node has some elements that it does not posses. It can then ask for the element(s) that point to the given indices. If any of the values on the other hand are in the positive range, the node knows that its peer is missing the elements whose hashes point at the indices of the positive values.

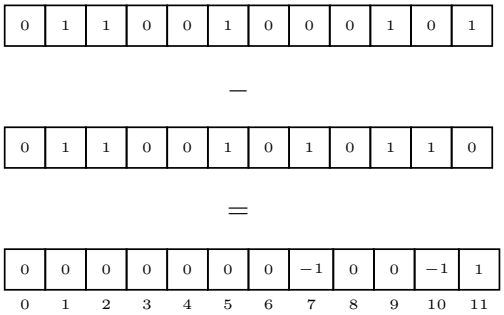

Figure 3: Reducing a bloom filter from another bloom filter

There is of course a catch with this approach - as mentioned earlier, there can be false positives. To guarantee a very low false positive rate, one has to build a significantly larger bloom filter. So we can either sacrifice bloom filter size, which results in set inconsistencies across the network (which is not ideal for most tasks), or we use large bloom filters, which will lead to very few inconsistencies (which is still not desirable for some tasks), but increased bandwidth.

This problem lead to the development of novel ideas such as the invertible bloom lookup tables (IBLT) [6], or using counting bloom filters (CBF) for set