Bloom filters are a variant on hash tables. They are called Bloom filters as they were developed by Burton Bloom in 1970. They are more space efficient than a hash table with the tradeoff there will be some false positive lookups (can be mitigated)

Supported Operations

Fast lookups and inserts.

Comparison to Hash Tables

Pro: More space efficient. This is because a Bloom filter only stores whether or not an object has been seen before, not the object or a reference to the object (pointer).

Cons:
1. Can't store an associated object. This is the tradeoff made for space efficiency.
2. No deletions are supported. Rather than perform a deletion which can lead to a false negative, a new Bloom filter would be initialized and constructed.
3. Small false positive probability - The lookup says the item exists or has been inserted even though it has not been.

Applications

Useful for instances where space is at a premium or false positives will not adversely affect the program.

Original use: Spell checker. In the 1970s memory was at a premium and the English language is large

Canonical: Maintaining a list of forbidden passwords. Very quick to tell a user of an insufficiently secure password and the damage from a false positive is extremely minimal; the user just has to create another password.

Modern: Network routers. Not a whole lot of storage space and a router has to process a lot of information quickly: lookup of spurious IP addresses, maintain statistics to identify a denial of service attack, keep track of the contents of a cache to prevent slow disk lookups, etc.

Implementation Details

1. Array of n bits, bit = $\emptyset$ or 1 (False or true, respectively)
2. Hash functions

Array Structure

The array consists of n bits.

$$n = \begin{pmatrix}\text{\# of bits chosen to} \\ \text{represent an object}\end{pmatrix} \begin{pmatrix}\text{\# unique elements} \\ \text{in dataset } S\end{pmatrix} \leftarrow \text{cardinality of } S,\ |S|$$

We can also adjust the size of the array to work within memory constraints.

$$\begin{matrix}\text{\# of bits needed to} \\ \text{represent an object}\end{matrix} = n|S|$$

While we can tune the size of the filter by varying n you get fewer false positives with decently sized n, for example, $n = 8$ or higher.

Hash Functions

In a bloom filter there must be k hash functions $(h, \ldots, h_k)$ with k being a small constant. The optimal value for k can be found with:

$$k = \left(\frac{\text{\# of bits in the array}}{\text{\# of inserted elements}}\right) \ln 2$$

Insert

To insert an element x into the Bloom filter A:

For $i = 1, \ldots, k$
Set $A[h_i(x)] = 1$ (Regardless if the value is $\emptyset$ or 1 in A)

Lookup

To lookup an element x in Bloom filter A:

Return True if $A[h_i(x)] = 1$ for every $i = 1, 2, \ldots, k$

IMPLEMENTATION DETAILS (con't)

ERROR RATE, ε

The third design factor in the design of a Bloom filter is knowing your acceptable error rate (rate of false positives). If there is a required constraint on the error rate it can have an effect on the size of the filter.

$$\varepsilon \approx \left(1 - e^{-ko/n}\right)^k$$

k = number of hash functions
o = number of objects inserted into the filter
n = number of bits in (size of) the filter

As such, knowing the acceptable upper bound on ε will help drive the parameters needed to correctly implement the Bloom filter.

$$n = \frac{-o \ln \varepsilon}{(\ln 2)^2}$$

$$k = \frac{n}{o} \ln 2$$