RedisBloom

A Bloom filter is a probabilistic data structure which provides an efficient way to verify that an entry is certainly not in a set. This makes it especially ideal when trying to search for items on expensive-to-access resources (such as over a network or disk): If I have a large on-disk database and I want to know if the key foo exists in it, I can query the Bloom filter first, which tells me with a certainty whether it potentially exists (and then the disk lookup can continue) or whether it does not exist, and in this case I can forego the expensive disk lookup and simply send a negative reply up the stack.

While it’s possible to use other data structures (such as a hash table) to perform this, Bloom filters are also especially useful in that they occupy very little space per element, typically counted in the number of bits (not bytes!). There exists a percentage of false positives (which is controllable), but for an initial test of whether a key exists in a set, they provide excellent speed and most importantly excellent space efficiency.

Bloom filters are used in a wide variety of applications such as ad serving - making sure a user doesn’t see an ad too often; likewise in content recommendation systems - ensuring recommendations don’t appear too often, in databases - quickly checking if an entry exists in a table before accessing it on disk, and so on.

How Bloom filters work

Most of the literature on Bloom filter uses highly symbolic and/or mathematical descriptions to describe it. If you’re mathematically challenged like yours truly, you might find my explanation more useful.

A Bloom filter is an array of many bits. When an element is ‘added’ to a bloom filter, the element is hashed. Then bit[hashval % nbits] is set to 1. This looks fairly similar to how buckets in a hash table are mapped. To check if an item is present or not, the hash is computed and the filter sees if the corresponding bit is set or not.

Of course, this is subject to collisions. If a collision happens, the filter returns a false positive - indicating that the entry is indeed found (note that a bloom filter never returns a false negative, that is, claim that something does not exist when it fact it is present).

In order to reduce the risk of collisions, an entry may use more than one bit: the entry is hashed bits_per_element (bpe) times with a different seed for each iteration resulting in a different hash value, and for each hash value, the corresponding hash % nbits bit is set. To check if an entry exists, the candidate key is also hashed bpe times, and if any corresponding bit is unset, then it can be determined with certainty that the item does not exist.

The actual value of bpe is determined at the time the filter is created. Generally the more bits per element, the lower the likelihood of false positives.

In the example above, all three bits would need to be set in order for the filter to return a positive result.

Another value affecting the accuracy of a Bloom filter is its fill ratio, or how many bits in the filter are actually set. If a filter has a
vast majority of bits set, the likelihood of any specific lookup returning false is decreased, and thus the possibility of the filter returning false positives is increased.

**Scalable Bloom filters**

Typically Bloom filters must be created with a foreknowledge of how many entries they contain. The \( b \) number needs to be fixed, and likewise, the width of the bit array is also fixed. Unlike hash tables, Bloom filters cannot be “rebalanced” because there is no way to know which entries are part of the filter (the filter can only determine whether a given entry is not present, but does not actually store the entries which are present).

In order to allow Bloom filters to ‘scale’ and be able to accommodate more elements than they’ve been designed to, they may be stacked. Once a single Bloom filter reaches capacity, a new one is created atop it. Typically the new filter has greater capacity than the previous one in order to reduce the likelihood of needing to stack yet another filter.

In a stackable (scalable) Bloom filter, checking for membership now involves inspecting each layer for presence. Adding new items now involves checking that it does not exist beforehand, and adding it to the current filter. Hashes still only need to be computed once, however.

When creating a Bloom filter - even a scalable one, it’s important to have a good idea of how many items it is expected to contain. A filter whose initial layer can only contain a small number of elements will degrade performance significantly because it will take more layers to reach a larger capacity.

**More info**

- [RedisBloom commands](#)
- [RedisBloom source](#)

**Updated:** April 26, 2022