Bloom Filter : A Probabilistic Data Structure

Vivek Kumar Singh Follow
May 27 · 7 min read ★

Bloom filter is a *probabilistic data structure* invented by *Burton Howard Bloom* in 1970. It allows for membership check in **constant space** and **time**. Bloom filter trades *exactness* for *efficiency* and has a large number of applications in software engineering.

Some of the properties of bloom filters are

- It allows for membership lookups in *constant space & time*. Bloom filter can very quickly answer YES/NO questions, like "*is this item in the set?*".
- Very **infrequently** it will give a *false positive* answer, implies it will say YES if the answer is NO (*Probably in the set*).
- It will **never** give *false negative* answer implies it will never say NO if the answer is YES (*Definitely not in the set*).
- Basic bloom filter supports two operations *test* and *add*.

- It has *constant time* complexity for both adding items and asking whether a key is present or not.
- You can't *remove* an item from a bloom filter.
- It also requires *very less space* compared to the number of items you need to store and check.

Bloom filters uses hash functions and before discussing bloom filter in detail, let's have a look at hashing and hash functions.

Hashing and Hash Functions

A hash is like a *fingerprint* of your data. A hash function takes input data which can be of *any length* and gives back output as an identifier of *smaller, fixed length*. The identifier can be used to index or compare or identify data.

Most hash functions are *one-way* operations, which means you can get an identifier from the data, not vice versa. *Two-way* hash functions also exist, but they are not particularly useful.

Important properties of hash functions are

- Same input must always have the same output. It is one of the most important features.
- The output values should be *uniformly distributed*. It means each possible output value should be equally likely.
- The output should be *distributed randomly*. Similar input should not give similar output.
- Each input should give a *unique* output to minimize the number of collisions.
- It should be *fast*.

Different hash functions are designed for different tasks. Their properties differ based on the task for which you are using them. Hash functions used with bloom filter should have the following properties.

- It should be fast.
- The hash function should have rare collisions and hash value should be evenly and randomly distributed.

Bloom filter implementation

A basic bloom filter will have two operations **test** and **add.** Base data structure for bloom filter is **bit vector** or **bit-array.** It uses a bit array of size **m** and **k** hash functions. Initially, all the bits in bit vector will be set to 0.

To add an element to the bloom filter, we hash the element k times using hash functions and set bits at indexes of those hash values.

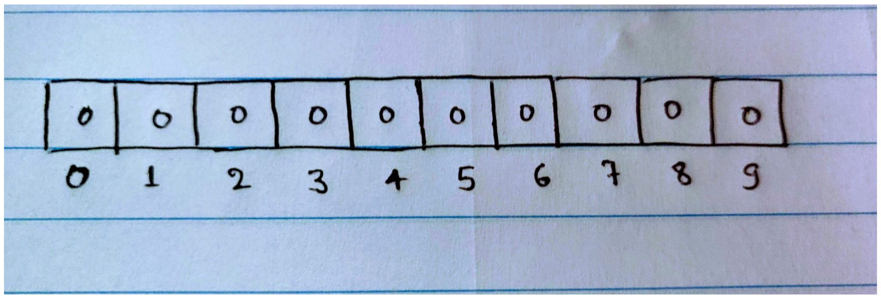
```
def add(item):
  for hash_function in hash_functions:
    index = hash_function(item) % m
    bit_array[index] = True
```

To test for membership, we simply hash the element with hash functions and then check if those indexes are set in the bit vector. If the bit at all those indexes is not set, you know that the element is not in the set. If they are set, it might be because the same element or combination of other elements could have set the same bits. Later is the reason why bloom filter can sometimes give a *false positive* answer.

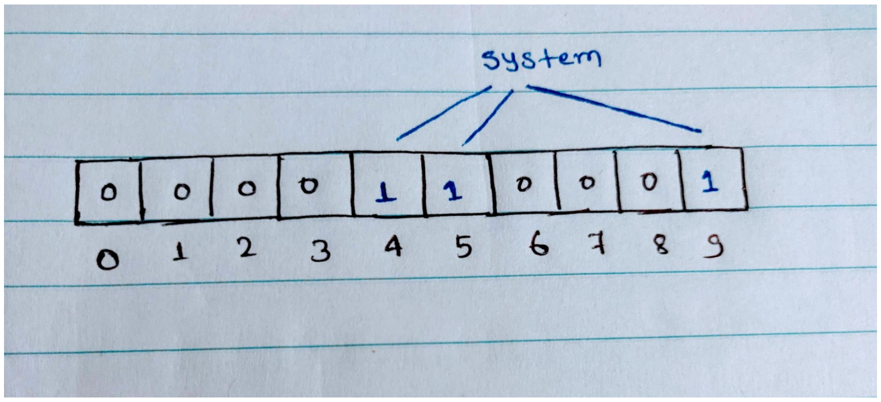
```
def test(item):
  results = []
  for hash_function in hash_functions:
    index = hash_function(item) % m
    if bit_array[index]:
      results.append(True)
    else:
```

For illustration, we will use a bit array of size 10 and three hash functions.

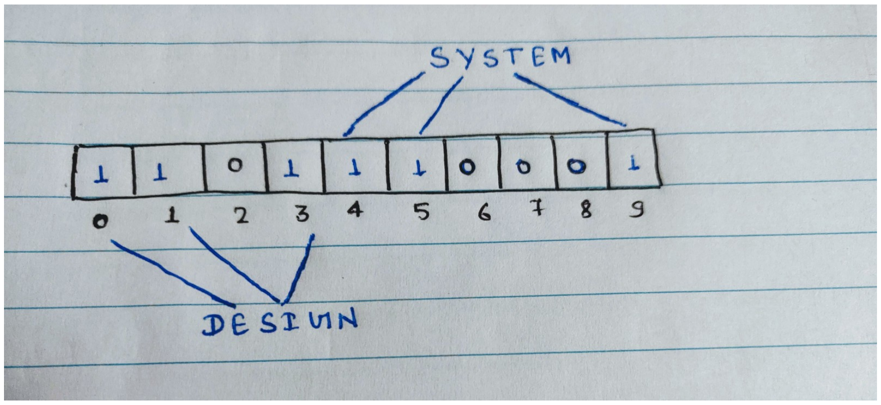
Initially, all the index in the bit vector will be zero.



Let's add SYSTEM to the bloom filter. Hash values for three hash functions are 4, 5 and 9. We set the bits at those indexes to 1.



Let's add DESIGN to the bloom filter. Hash values for three hash functions are 0, 1 and 3. We set values at these indexes to 1.

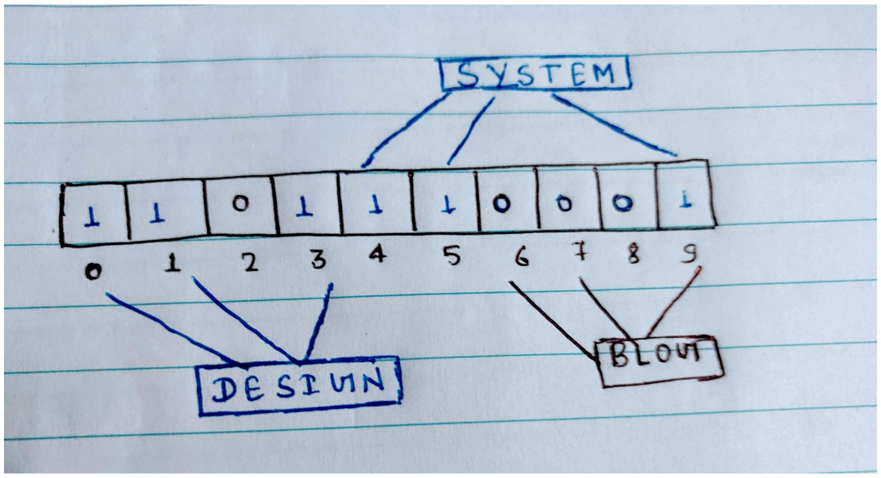


Now let's check whether bloom filter has SYSTEM. SYSTEM will be hashed to 4, 5, and 9. All three bits are set in the bloom filter so we can say that SYSTEM exists.

False Negative is Impossible (Definitely not in the set)

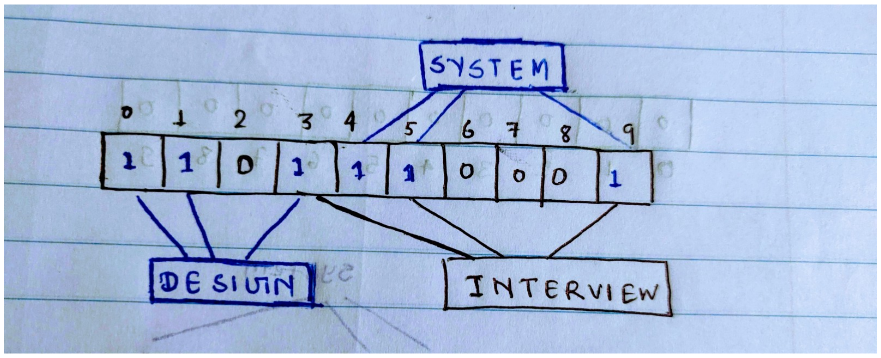
Let’s check membership for BLOG which hashes to 6, 7 and 9. In bloom filter, only index 9 is set, 6 and 7 are not. This means the BLOG is not present in the bloom filter. If it was present in the set, index 6 and 7 would have been also set.

So when an element is not present in the set, we are definitely sure that item is not present and there is no scope for false negative.



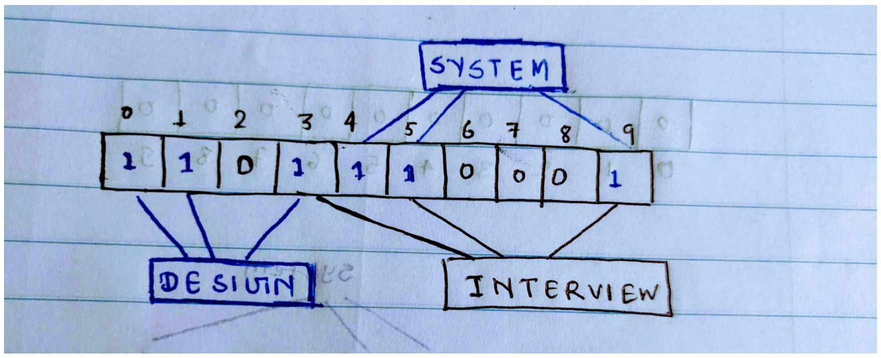
False Positive (Possibly in the set)

Now let’s look at the case of false positive. We want to check membership for INTERVIEW which hashes to 3, 5 and 9. We can see 3, 5 and 9 are set to 1, which means INTERVIEW is present. But wait for a second, we never added INTERVIEW to the bloom filter. This is a false positive. Those Indexes were set by SYSTEM and DESIGN. More occupied cells in the bit-vector, higher the chance of false positive.



Remove an Item

Suppose we have already added INTERVIEW to the bloom filter.



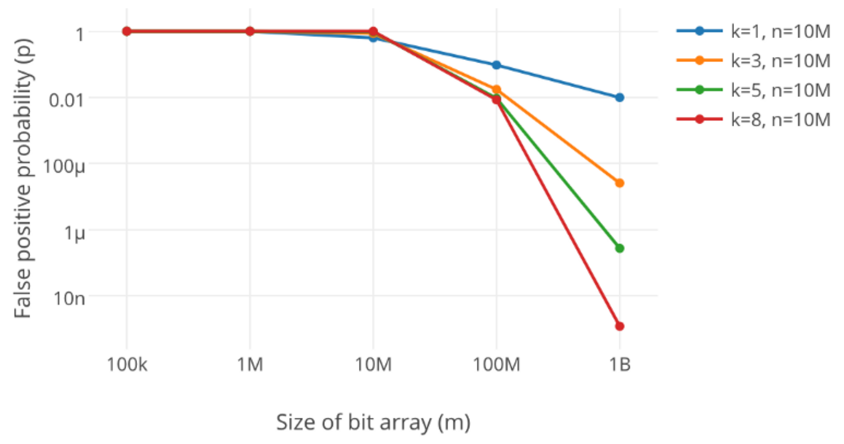
Now we want to remove SYSTEM from the bloom filter. To do that we need to unset indexes 4, 5 and 9. If we do so, INTERVIEW will also be removed from the bloom filter because 5 and 9 were also set by INTERVIEW. It is almost *impossible* to remove an element from a bloom filter without removing other elements.

Bloom filter size and number of hash function

Bit array size and number of hash functions plays an important role in the false positive rate for the bloom filter. If the size of the bit array is too small, all bits will be set to 1 more quickly. It will start giving a false positive result for every input after that. Larger the bit array lower the false positive result.

If we have too few hash functions there is more chance for a false positive result. If we have too many hash functions, bloom filter will become slow because it will have to hash every input that many times. If there are k number of hash functions, **test** operation will be **o(k)**.

Suppose, **n** is the number of elements
m is the size of the bit array
k is the number of hash functions
p is the false positive rate



Source: https://www.semantics3.com/blog/use-the-bloom-filter-luke-b59fd0839fc4/

As we can see from the above graph that with increasing value of the number of hash functions **k,** will reduce false positive rate. But a large value of **k** can also make our bloom filter slow. In this case (n = 10 M and p < 0.01), the optimal value for m = 100M and k = 5 because it minimizes for space and time complexity. For more information on this calculation, please refer to this blog by Abhishek Bhat.

Applications

Bloom filter has a large number of applications in software engineering.

- Medium uses bloom filter in medium blog recommendation to check whether a user has already read this post before or not. Read more about it on this amazing blog.
- Google Bigtable, Apache HBase and Apache Cassandra, and Postgresql use bloom filters to *reduce the disk lookups* for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation.
- Bloom filter is used by Content Delivery Network provider Akamai to avoid caching web objects which are requested by users just once (also known as "**one-hit-wonders**"). Using a

Bloom filter to detect the second request for a web object and caching that object only on its second request prevents one-hit wonders from entering the disk cache, significantly reducing disk workload and increasing disk cache hit rates.

- The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned if that too returned a positive result)

Some of the existing implementations of bloom filters are

- Bloomd is a high-performance C server which is used to expose bloom filters and operations over them to networked clients. It nicely fits into the micro-services architecture.
- PyreBloom is python based implementation of bloom filter based on Redis
- The Cuckoo filter is another variation of the bloom filter. Cuckoo filters support adding and removing items dynamically while achieving even higher performance than Bloom filters. There is a nice blog by Farhan on the difference between Cuckoo and Bloom filters.