


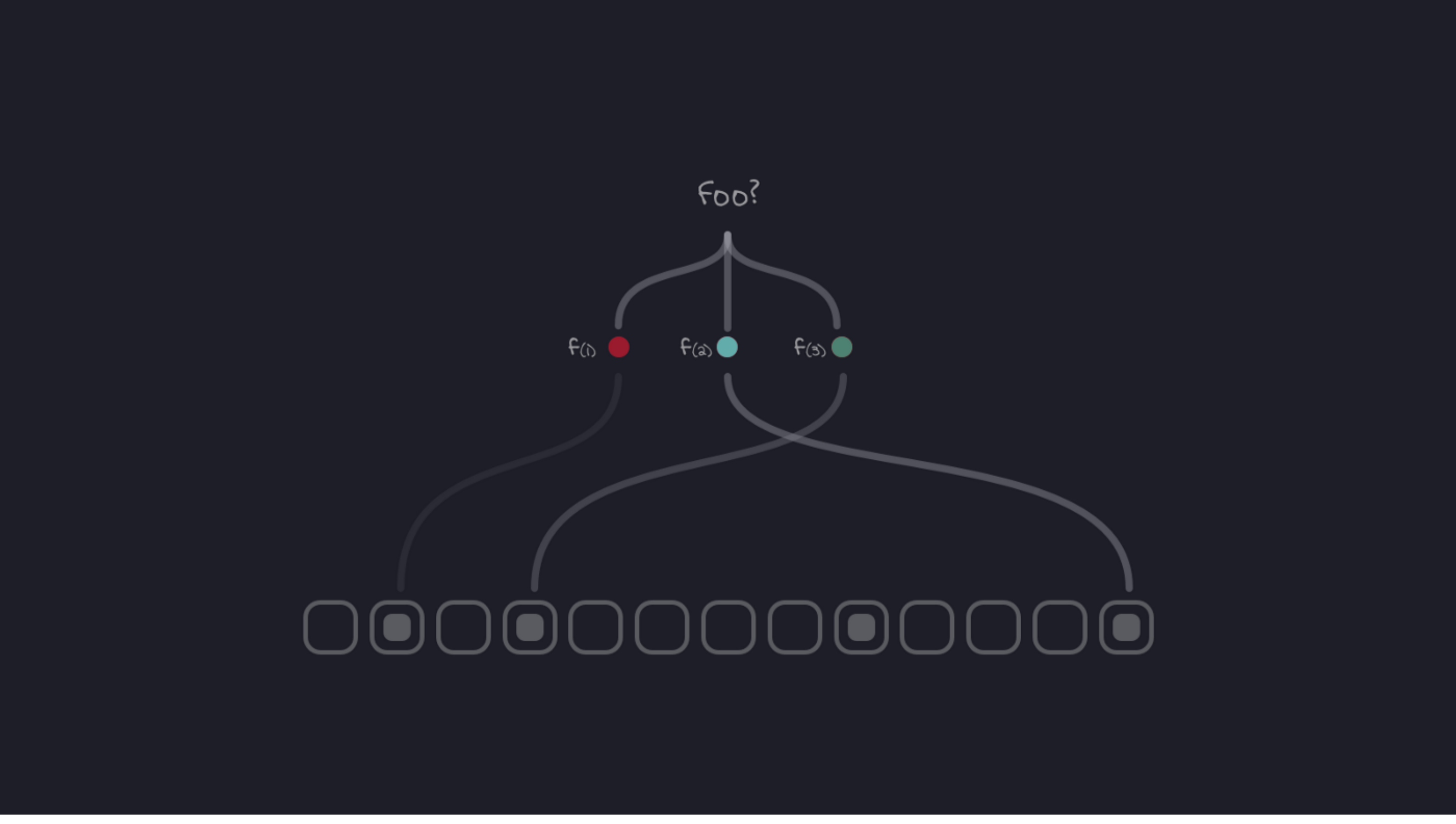
Bloom Filter: A simple but interesting data structure





Kousik Nath

Sep 22, 2018 · 8 min read



In computing, there are many scenarios where we search a little amount of data in large pool of data stored somewhere. As engineers, our job is to optimise that searching, we constantly try to find new data structures, new technologies, new mechanism to make that search a low latent & high throughput operation. Bloom filter is something which helps us to minimise that search operation in certain use cases (read not all use cases). It was invented by [Burton Howard Bloom](#) in 1970.

Consider some use cases which help you understand this concept:

Say you are creating a new email id in Gmail. Gmail has to find out whether the id that you are providing is available or not. Now there are certain ways of doing it. Gmail can scan across all the existing email ids in its data store (thousands of database & cache servers) & find out whether that id already exists. But just imagine, there are billions of email ids stored in Gmail already, is it a feasible approach to scan in thousands of servers for every new email id? Isn't it a better idea to approximately guess the status of the provided id?

Say you are using a cloud based security service which protects you from accessing malicious urls. That service might host a database of millions of malicious urls, might cater to millions of request per minute worldwide. In this scenario, searching a url in their database or cache is huge challenge — what about using some approximation or probability here also to quickly identify if the url is safe or not?

I just gave 2 examples to simplify the concept. Try to find similarity in these examples:

- i) We are trying to find if some item exists in a set of similar items,
- ii) None of these scenarios requires exact answer. Probable answer is fine. If the provided gmail id is not really hosted in Gmail, still Gmail says it's unavailable, that's not a very big problem for the user, user can try with another id and probably after few try, user will be successful to create his / her email id, it's not an endless loop. Similarly in the second example, if by mistake that cloud service identifies some safe site as a malicious site, that's not a very big problem till the time a real malicious site is identified as a safe site to access.

You might say hash / distributed hash based solution can be used to solve this this problem, but hash needs to store the actual item in its internal structure, that will consume huge space. Also for hosting millions to billions of items, you need to allocate a lot of space across thousands of servers. Unless your use-case demands exact result from membership check, using hash is really a space compromising solution which is not actually required in many use-cases.

Bloom filter comes into rescue in this sort of scenario. It is used to check the membership of an item in a set of items. If you ask it whether an item exists in a set, it can provide you any of the 2 answers — ‘May be present’ or ‘No’. So you see clearly that Bloom filter guarantees ‘No’ answer for membership check if the item does not exist in the set, but it's not so sure about the item when it exists in the set — there is a very high chance that the item exists in the set, but not necessarily it exists. That's why Bloom filter is probabilistic in nature & this trade-off makes member searching quite fast also.

So how does it work? Wikipedia explains the [operation](#) very well. In short: you have an array of size m where each cell only contains 0 or 1 in the array. Say $m = 8$ in our example, so the array initially looks like: $[0, 0, 0, 0, 0, 0, 0, 0]$. There are k very efficient hash functions which can produce uniform result for different inputs, say hash functions are — $f_1, f_2, f_3, ..., f_k$. You have n strings to save in the membership set. Now you pass the string one by one through the k hash functions and you take $\text{mod } m$ of each result. Example: string is "Bangalore" . Say the hash functions produce the following results:

```
str = "Bangalore"
f1(str) = 110
f2(str) = 23
f3(str) =54
....
fk(str) = 125
```

Now you take $\text{mod } m$ of each result. In our case $m = 8$. So:

```
110 mod 8 = 4
23 mod 8 = 7
54 mod 8 = 6
.....
125 mod 8 = 5
```

The above mod results are always less than m and they represent positions in the array. The idea is to set all the cells at these positions to 1 if it's not already set. So now our array becomes: $[0, 0, 0, 0, 1, 1, 1, 1]$. We can continue doing this process for all n strings.

Now when I check for the membership status of "Bangalore" in the set, I will pass the string through all these k hash functions, the hash functions will generate the same result, I will take their $\text{mod } m$ result and check if the corresponding positions are set in the array. If all the indices are set, that means there is a chance that the element exists in the set. If any of the positions are not set, then definitely the element does not exist in the set. There might be a situation where almost all of the cells in the array are set to 1, so when we query for an item's existence in the set, we might falsely get 'May be present' result although the item is not there in the array. So if the size is not properly chosen, the tendency towards false positive result might increase. There is some error rate associated with Bloom filter, there is always a chance that you might get some false positive result. Probability of getting false positive is:

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

Courtesy: Wikipedia

You can play around with some values of m , k & n just to identify which values work best to get the least error probability.

Given proper values of m & k according to n , bloom filter will produce less false positives and it will strictly say 'No' when the element does not really exist in the set.

Some real life use of Bloom Filter:

- Cassandra usage:** Apache Cassandra uses SSTable data structure on disk to save rows. So at a millions of scale, there will be thousands of SSTable files on disk. Even tens of thousands of read requests per unit time will cause very expensive disk IO operations to search for concerned row(s) in all the SSTables one by one in some order when data is not found in the in-memory tables of Cassandra. So bloom filters are used to approximately identify if some row / column with the given data or id exists in a SSTable. If the result is 'May Be Present', Cassandra searches in the corresponding SSTable, in case the row is not found, Cassandra continues search operation in other SSTables. So using bloom filter, Cassandra saves a lot of unnecessary SSTable scan hence saving huge disk IO operations cost.
- Content Recommendation System:** Imagine some site recommending you some articles, news, videos etc which you might not have seen earlier. So there are probably thousands of stuffs which can be recommended and you might have seen tens or hundreds of recommendation already. So in order to skip the recommendations that are already served to you, bloom filters are used. Medium uses bloom filter to avoid showing duplicate recommendations.
- One-hit-wonders:** Akamai & Facebook uses bloom filters to avoid caching the items that are very rarely searched or searched only once. Only when they are searched more than once, they will get cached. Several strategies might be designed to avoid such situations.
- Financial Fraud Detection:** If you have a credit card, your credit card company knows about your spending history — the vendors that you have transacted with previously, the category of vendors, the cities where the card was used etc. So when you make a new transaction, in the background some rules can execute which will decide whether the vendor or city or any parameter is already seen or suspicious. Bloom filters can be used to design such strategy.

Advantage of using bloom filter:

- Space efficiency:** Bloom filter does not store the actual items. In this way it's space efficient. It's just an array of integers.
- Saves expensive data scanning across several servers depending on the use case.

There are some caveats to use bloom filter:

- Removal Of Item:** If you remove an item or more specifically clear set bits corresponding to some item from bloom filter, it might erase some other data as well because many set bits might be shared among multiple items. So remember if you use bloom filter, removal is not an option for you. What's inserted in bloom filter, stays in bloom filter.
- Inserted items are not retrievable:** Bloom filters does not track items, it just sets different positions in the array. So inserted items can not be checked out.
- False Positive result:** The size of the bloom filter has to be known beforehand. Otherwise with a small size, the array will saturate and the amount of false positive result will increase. In case of saturation, you can try designing some strategy to reset the bloom filter or use a scalable bloom filter. If you keep adding more & more elements to the bloom filter, the probability of false positive increases.
- Insertion & Search Cost:** Every item goes through k hash functions. So performance during insertion or search operation depends on the efficiency of the chosen hash functions as well. Inserting element & searching element takes $O(k)$ time as you run k hash functions and just set or check k number of indices in the array.

Some Language Implementations:

- [pybloom](#) is a Python implementation that grows automatically once it has reached full capacity.
- Java [Bloom Filter implementation](#) in Guava library.

So if you are fine with some error typically $\text{error} \leq 2\%$, you don't need to rely on exact answer of membership check, also you want to avoid expensive search operation over disk or network or any suitable medium by checking probabilistic-ally beforehand if the element possibly exists, bloom filter is the way to go.

References:

[1] <https://www.kdnuggets.com/2016/08/gentle-introduction-bloom-filter.html>

[2] <https://www.jasondavies.com/bloomfilter/>

[3] <http://www.mahdix.com/blog/2016/08/08/cassandra-internals-bloom-filters/>

[4] <https://sc5.io/posts/what-are-bloom-filters-and-why-are-they-useful/>

[5] <https://github.com/jaybaird/python-bloomfilter/blob/master/pybloom/pybloom.py>

[6] https://nofluffjuststuff.com/blog/billy_newport/2010/01/bloom_filters_for_efficient_fraud_detection_with_ibm_websphere_extreme_scale

[7] https://en.wikipedia.org/wiki/Bloom_filter