Special Issue - 2020
International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
NCAIT - 2020 Conference Proceedings

Bloom Filter

A Data Structure for Quick Searching

Pranav Byali, Md Zaid S Bevinahalli, Vaishnavi Chavan
Dept. of Information Science and Engineering
KLE Institute of Technology, Hubli

***Abstract*- A Bloom filter is a data structure designed to tell us whether an element is present in a set. The base data structure of a Bloom filter is a Bit Vector (an array of flag variables). This data structure is used when there are searches to be made in large datasets. The results are definite negatives or probabilistic positives. In datasets containing records in numbers very large where search algorithms like linear search and binary search will not be suitable in many case. Bloom filter fits more appropriately in such cases. Lists like the ones of all google ids are difficult in cases where one id is to be checked for membership. In such cases elements in the set will be in billions in number. Searches will take unacceptable amounts of time to complete with linear search or binary search, or any other conventional search algorithms The objective in this seminar to explain how searches can made without actually checking to see if the element is present in the set, i.e. without comparing the element with every element in the set. The objective is also to explain how use of bloom filter is a better way of searching or checking membership.**

I. INTRODUCTION

A Bloom Filter is a data structure designed to tell if an element is definitely not present in a set. It gives probabilistic positive results or definite negative results. Bloom filter either tells us that an element is definitely not present in a set or may be present in the set. Bloom filter gives results rapidly and memory efficiently

II. THE DATA STRUCTURE OF BLOOM FILTER

The base data structure of a Bloom filter is a Bit Vector. A vector in programming is just an array and a bit is what is called a flag. Flags take one of two possible values zero or one.

Hash Functions

A hash function is like an f(x) of math. It passes a given value through some operation(s) and gives a value, the hash of that value. To add an element to the Bloom filter, we simply hash it a few times and set the bits in the bit vector at the index of those hashes to 1.

False Positive probability

The probability of a false positive result for the search of an element not in the set, or the false positive probability, can be calculated in a simple way, given our assumption that hash functions are completely random. After all the elements of set S are hashed into the Bloom filter, the probability of a specific bit being 0 is

$$p = (1-1/m)^{kn} \approx e^{(-kn/m.)}$$

If we plot the false positive probability against the size of the bit array or the bloom filter for a fixed number of elements that are to be inserted, we can see that the rate of false positives decreases as bit array size increases. From the graph for $p < 0.01$, we would need a bit array of size 100x the number of values (n) if we used only one hash function. A key idea behind a bloom filter is that if we use k hash functions, the space requirement (m) can be reduced in a considerable amount. Whenever a new value is to be added to the filter, it is hashed and the bits at the values gotten by the hash functions are set to 1. Similarly, when we need to check if the given value exists, we hash it and check if every corresponding bit gotten by hash functions is set to 1. If one bit is 0, it is concluded that the value is not present.

Choosing Hash Functions

Depending on the number of values to be inserted and the maximum acceptable false positive rate, we can get at the optimal number of hash functions to be used. There is a lot of content on how to choose good hash functions.
$p < 0.01$ at (k=1, m=1B), (k=5, m=100M) and (k=8, m=100M). Here, (n=10M, $p < 0.01$) our
ideal choice would be (k=5, m=100M) as it minimizes both the space and time/compute requirements.
A bloom filter's aim, by default, is to minimize the size of it's bit array, and the number of hash functions

Drawbacks

A bloom filter which is filled to it's capacity would cause the false positive possibility to be more.
The false positives increase as the filter is filled to a point where it is not useful for lookups. Choosing a large bit array size to solve this issue would make the bloom filter sparse and also would slow down the lookups. When the number of elements in the filter is maximum, expanding the size of the filter isn't exactly rendering them to be very rigid. The rigidity can be dealt with by adding another bloom filter when (m/n) crosses a certain threshold. Additional elements go directly into the new filter, while lookups would have to go via every single filter in the chain. This leads to multiplicative increases in the overall false positive probability. Chaining bloom filters would still require tighter constraints on false positive rates to maintain the overall false positive rate at the desired level, due to multiplying effect in the chaining.

Bloom Filter Working

The bloom has to do with, as we have seen, two things. One is when an element is added and another when we check if an element is present in a set.

Adding elements to the set

When we want to add Elements to the set, they are hashed and the bits at the positions of the hashes are set to 1.

Suppose we want to add the string “filter” to the set. We get some hash values, say as follows.

h1(“filter”) % 10 = 1

h2(“filter”) % 10 = 4

h3(“filter”) % 10 = 7

The positions 1, 4 and 7 in bloom filter now have their bits set to 1 and the rest are zero.

Checking membership

While checking membership, we refer to the filter. The hash of the element is calculated and at these hashes the bits are checked to be set. For example, if checking for the string “Element”, the hashes are first calculated. Bits at the positions gotten are calculated. If all are set then the string is present, else it is not.

Probability of false positives

Say a hash function treats each array position with the same probability. Consider m as the number of bits in the bloom filter, the probability that any of the bits is not set to 1 by a hash function during the insertion of an element is 1-1/m. If k is the number of hash functions and each has no significant correlation between each other, then the probability that any bit in bloom filter is not set by any of the hash functions is (1+1/m)^k. If we have inserted n elements, the probability that a certain bit is still 0 is the probability that it is 1 is therefore

1-(1-1/m)^kn

For testing membership of an element that is not in the set, each of the positions gotten by the hash functions is 1 with a probability as above. The probability of all bits being 1, which would cause the algorithm to give result that the element is in the set, is often given as

(1-(1-1/m)^kn)^k Which is

approximately, (1-e^-(kn/m))^k

This is not strictly correct as it assumes independence for the probabilities of each bit being set. However, assuming it is a close approximation we have that the probability of false positives decreases as m (the number of bits in the array) increases, and increases as n (the number of inserted elements) increases. An alternative analysis arriving at the same approximation without the assumption of independence is given by Mitzenmacher and Upfal. After all n items have been added to the Bloom filter, let q be the fraction of the m bits that are set to 0. (That is, the number of bits still set to 0 is qm.) Then, when testing membership of an element not in the set, for the array position given by any of the k hash functions, the probability that the bit is found set to 1 is 1-q

. So the probability that all k hash functions find their bit set to 1 is (1-q^k). Further, the expected value of q is the probability that a given array position is left untouched by each of the k hash functions for each of the n items, which is (as above)

E(q)=(1-1/m)^kn

Optimal number of hash functions

We are given m and n, so we choose a k to minimize the false positive rate. Let p = e−kn m

. Thus we have

f = (1 – e^(−kn/m)^k

= (1 − p)^k

= e^(kln(1−p))

So we wish to minimize g = k ln(1 − p) We could use calculus. Less messy, we notice that since ln(e^(−kn/m))= −kn/m

we have

g = k ln(1 − p) = − (m/n)ln(p)ln(1 − p)

and by symmetry, we see that g is minimized when p = ½ Since p = e−kn/m , when p = 1/2 we have k = ln2 ·(m/n) Plugging back into f = (1 − p)^k, we find the minimum false positive rate is (1 2)^k ≈ (.6185)^(m/n) 3.3

Optimal Filter Structure Recall

p = e^(−kn)/m is the probability than any specific bit is still 0. So p = 1/2 corresponds to a half-full Bloom filter array.

III. APPLICATIONS

Bloom filters are useful when trying to determine of element doesn’t belong to a set. Other than this, bloom filter has a few applications in the field security.

Google Ids

When creating a new user id in google, we may see “user name is already taken”. This is because google needs user names to be unique. Now, when the user enters the user name, it has to be checked to see if it is present in the set of existing user names. There are more than 1.7 Billion gmail users. Comparing the new user name with existing usernames using whichever algorithm, is not an efficient way of solving this problem. For this, Google used bloom filters.

Instagram

Instagram, like Google, needs user names to be unique. Just like in creating Google user id, creating new user name in instagram is also quite a task. Instagram had reported in 2018 that it has over 1 billion users. Instagram too, for creating new user names, used bloom filters.

Other applications

The servers of Akamai Technologies, a satisfied delivery provider, use Bloom filters to prevent "one-hit-wonders" from being stored in its disk caches. One-hit-wonders are web objects requested by users just once, something that Akamai found applied to nearly three-quarters of their caching infrastructure. 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.[10] Google Bigtable, Apache HBase, and Apache Cassandra and PostgreSQL[11] use Bloom filters to decrease the disk lookups for fictitious rows or columns. Evading costly disk lookups considerably increases the performance of a database query operation.[12] The Google Chrome web browser used to use a Bloom filter to recognize 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).[13][14] Microsoft Bing (search engine) uses multi-level hierarchical Bloom filters for its search index, BitFunnel. Bloom filters produced a lower cost than the previous Bing index, which was based on inverted files.[15]. The Squid Web Proxy Cache uses Bloom filters for cache digests.[16] Bitcoin uses Bloom filters to speed up wallet synchronization.[17] The Venti archival storage system uses Bloom filters to detect earlier stored data.[18] The SPIN model checker uses Bloom filters to track the reachable state space for large verification problems.[19] The Cascading analytics framework practices Bloom filters to speed up asymmetric joins, where one of the joined data sets is significantly larger than the other (often called Bloom join in the database literature).[20]
The Exim mail transfer agent (MTA) uses Bloom filters in its rate-limit feature.[21] Medium practices Bloom filters to avoid recommending articles a user has earlier read.[22] Ethereum uses Bloom filters for quickly finding logs on the Ethereum block chain.

Bloom Filters other applications

Weak Password Dictionary: Store dictionary has easy passwords as bloom filters, queries when users pick passwords. Can add new entries (e.g. earlier used passwords). What is a false positive in these circumstances? A strong password that strikes to hit, just asks the user for another password. Application: Cache Sharing: Proxies on the same side of the network bottleneck share their caches. Proxies use the Internet Cache Protocol (ICP). Proxy hashes all of the URLs in its cache into Bloom Filter. Proxies regularly exchange Bloom filters, so queries of other caches can be made locally without sending an ICP message.

Bloomier filters

Chazelle et al. (2004) designed an idea of Bloom filters that could link value with each element that had been inserted, implementing an associative array. While using a Bloom filter, these structures assume the potential effects of false positives. In the even of "Bloomier filters", a false positive is determined as returning a result when the key is not in the map. The map will never return the wrong value for a key that is in on the map.

Counting Bloom filters

Counting filters abide by to implement a delete operation on a Bloom filter without refurbishing recreating the filter afresh. In a counting filter, the array positions (buckets) are extended from being a single bit to being a multi-bit counter. Regular Bloom filters can be considered as counting filters with a bucket size of one bit.
The insert operation is used to increment the value of the buckets, and the lookup operation checks that each of the required buckets is non- zero. The delete operation then includes decrementing the value of each of the buckets. The size of counters is usually 3 or 4 bits. Hence counting Bloom filters use 3 to 4 times more space than static Bloom filters.

Layered Bloom filters

A layered Bloom filter has multiple Bloom filter layers. Layered Bloom filters help us to keep the track of how many times an item was added to the Bloom filter by checking how many layers contain the item. With a layered Bloom filter, a check operation will normally return the deepest layer number the item was found in.

IV. CONCLUSION

Bloom Filters and their extensions can be useful tools for many of the applications. When a list or set is being used, and space is a consideration, a Bloom filter can be used. While using a Bloom filter, assume potential effects of false positives. Bloom filters are simple data structures that can be used in practice. They are so useful that any significant reduction in the time required to perform a Bloom filter operation immediately translates to a substantial speedup for many practical applications. There is no room for optimization in Bloom Filters. This report focuses on the introduction to Bloom Filters, and applications of bloom filters in searching over other search algorithms.

V. REFERENCES

[1] Adam Kirsch,* Michael Mitzenmacher, Less Hashing, Same Performance: Building a Better Bloom Filter, In: Harvard School of Engineering and Applied Sciences, Cambridge, Massachusetts 02138; e-mails: {kirsch, michaelm}@eecs.harvard.edu

[2] Jacob Honoroff, An Examination of Bloom Filters and their Applications, March 16, 2006