Bloom Filter Executive Summary

Brian Bippert, Emily Rothrock, Livvy Yurish

Introduction

A Bloom Filter is a probabilistic data structure proposed by Burton Bloom in 1970 that tests whether an element is a member of a set. It is similar to a HashSet, but performs better on large datasets for membership queries because its worst case is not a linear time complexity, but constant based on the time complexity of the chosen hash function. However, this speed comes at the expense of query responses being non-deterministic. An element can only be “possibly in” or “definitely not in” a set. Bloom Filters use less memory than HashSets and other set data structures, with a constant space complexity. Other data structures' space complexity is based on the number of total elements in the set. Deletion is not supported in a normal Bloom Filter, however, later papers have proposed extra data tracking or modified versions of the Bloom Filter to allow for deletions and deterministic “in set” or “not in set” results. Regardless, adding a member or querying for membership has constant time complexity and constant space complexity.

Bloom Filters can be useful when normal hashing methods require too much memory for storage due to the constant space complexity. Because of their tree structure, queries for membership can also be run in parallel on different child nodes, which increases speed even more compared to other similar data structures. Bloom Filters are particularly useful for caching. Traditional caching can use high amounts of memory and still have slow lookups, especially on large datasets. However, since a Bloom Filter can quickly determine whether an element is definitely not in a set, it is a quick way to know when an element has not already been cached. In addition to caching, Google Bigtable, Apache HBase, Apache Cassandra, ScyllaDB and PostgreSQL use Bloom Filters for improved query performance on non-existent rows or columns in a database.

Approach

A basic Bloom Filter has an array of bits, initialized to zero. To insert, an element is hashed by a pre-chosen number of hash functions to get all the indices in the bit array to change from zero to one. This differs from a normal HashSet where only one hash is used to find the insertion index for the array representation. However, unlike a normal HashSet, a basic Bloom Filter does not support deletion, so the same index could represent multiple inserted elements if there are collisions for element hashed indices. This is why the Bloom Filter can only deterministically answer whether an element is not in the set.

Many hash functions are viable in a Bloom Filter, though most implementations use a hash function that is very fast, since a Bloom Filter is only as fast as its hashing function. However, many of these functions are not cryptographically secure because they rely on xor, shifting, or

rotation operations that are reversible. MD5 is a popular choice because it balances both speed and security. In our solution, we used MurmurHash3 because it is readily available in most languages. However, the C++ challenge solution uses a modified version of MurmurHash3 for ease of use.

The original Bloom Filter outlined by Burton Bloom does not support deletions, which is why without modifications it is only possible to definitely say whether an element is not in the set. However later variations have solved the issue while still minimizing storage needs. For our purposes, we can support deletion with either a Counting Bloom Filter or by tracking collisions to determine which indices are safe to reset.

Implementation

To start, there are three key variables that will determine the structure of our Bloom Filter: *k*–the number of hash functions, *m*–the number of bits in the filter, and *n*–the number of elements to be inserted. By default, if *m* is not provided then it is assumed the false positive rate *P* is 0.001, and the number of bits in the array can be calculated with $m = -\frac{nln(P)}{(ln(2))^2}$. *k* can be calculated with $k = \frac{m}{n}ln(2)$. In turn, *P* can be found by $P = (1 - [1 - \frac{1}{m}]^{k*n})^k$, guaranteeing a probability *P* of a false positive given *m, n,* and k. Beyond that, there is also *bits[]*, which is an array representing the bits of the filter, and *seeds[]* which is an array of random integers used for generating independent hash functions from a single parent hash function.

In order to set the hash functions themselves, first integer "seed' values must be calculated at random. In order for the Bloom Filter to function properly, the *k* hash functions must be independent and uniformly random. In python, this can be accomplished with mmh3 by using a single hash function (like MurmurHash3) and modifying its output by using different seed values. Different seed values create separate versions of the hash function, allowing each one to produce a unique and independent index for the same input. This means that when the same element is hashed using each seed, it is likely to map to a different position in the bit array each time, spreading out the indices in *bit* that are set.. This is fundamental in maintaining a low false positive rate and space efficiency.

```
function set_seeds(k):
    self.k = k
    rng = new random generator
    self.seeds = []
    for i in range(k):
        seed = rng.randint(0, 2147483647)
        self.seeds.append(seed)
```

Each implementation handles the hashing slightly differently due to the availability of tools in the language and the difficulty of the task. Some solutions, like the regular C++ implementation,

hash longs, while the python and Java solutions hash Strings. Regardless, the result is a 32-bit hash that is then reduced by modulo *m* to adjust the result to fit within the bounds of the bit array. Finally the corresponding *k* bits in the *bit*[] array are then set to True to represent the item has been added. Below is pseudo code that shows how variable *add* is hashed *k* times, each with a different seed, and how the values in *bit*[] are set.

```
function insert(self, n:str):
    for i in range(self.k):
        s = self.seeds[i]
        s = some hash of s given the seed
        self.bit[s % self.m] = True
```

In order to query membership in the set, the same process as adding an element is used. The input element is converted to bytes and *k* distinct hashes are computed. For every index produced by hashing, if the corresponding bit in the *bits[]* array is examined, and this holds *True* for all *k* hashes, then the element is probably present. If even one of these bits is *False*, the element is definitively not in the set. This basic implementation of a Bloom Filter guarantees no false negatives. However, there may sometimes be a false positive during collisions. Thus a tradeoff must be made between the false positive rate and space in memory. A lower rate results in a larger *m* and *bits[]* array size, but fewer collisions and false positives.

```
function contains(n):
    for i in range(k):
        s = seeds[i]
        s = some hash of n given the seed
        index = s % m
        if not bit[index]:
            return False
    return True
```

To extend the Bloom Filter to handle deletion, one method is implementing a collision array, where each bit in the *bits[]* array comes with a companion bit in a flag array indicating if that bit was set multiple times by different elements. When inserting an element, if a bit is not already set, it is turned to one in the *bits[]* array. However, if the bit is set to one, the corresponding bit in the collision flag array is marked as true, which means that more than one element has contributed to that bit (or a collision). During deletion, a bit is only set to zero for an element if its collision flag is false. If the collision flag is true, the bit is left intact to avoid inadvertently removing an index set by another element, thus preventing false negatives.

Another option for deletion is a Counting Bloom Filter. A Counting Bloom Filter is another extension of the standard Bloom filter that uses an array of counters instead of simple bits set to

0 or 1 to allow for safe deletion of elements. When inserting an element, each index produced by the hash functions has a corresponding index in the array of counters that is incremented. To query for membership, a check is done to see if all the corresponding values in the counters array are greater than zero. If the value at any index in the counters array is zero then the queried element has not been inserted. For deletion, all counter bits for an element are decremented by one, with a counter only considered cleared when it drops to zero. This technique enables the filter to remove elements without accidentally affecting bits that other elements have contributed to it, since a counter bit only goes to zero when there is no element in the set that can hash to that bit. While a Counting Bloom Filter effectively prevents false negatives, it does come at the cost of increased memory usage and computational complexity compared to a regular Bloom filter.

Programming Challenge

Our programming challenge tasks students to build a high-performance packet filter and tracker for a fictional video streaming service called Metflicks. Students are “engineers” that must reject packets from IP addresses on a “DO NOT SERVE” blacklist and detect whether any sequences of packet payloads form known "bad data" patterns, even across multiple packets. If any user sends three such sequences, they are permanently blacklisted. Packets are in the form of a 32-bit source IP address followed by a 32-bit data bitstring. “Engineers” must correctly validate message chains per user in addition to keeping track of user history to determine if a user is a bad actor deserving of the blacklist.

The main purpose of this assignment is to add good actors to a good actors bloom filter, and bad actors to a bad actors bloom filter. However, good actors can move to the bad actors bloom filter. Then, the test case will ask which, if any, a given ip address belongs to. A primary challenge with this structure in general is that different executions will produce different results, and this error is especially pronounced on larger test sizes due to the chance of false positives. For this reason, most test cases that contain a large number of packets or test IP addresses will see a lot of variety in the answers, with the actual correct solution only occasionally being found by the bloom filter. For smaller cases, this is not an issue and the solution should be correct most of the time. Due to the nature of the way this problem is outlined, false positives and false negatives are possible due to collisions within the data bloom filter and the bad IPs bloom filter. This is another challenge that this problem poses, but also makes generating test solutions nearly impossible for any interesting case.

Our goal with this challenge is to teach how to manage large-scale data validation using a Bloom Filter. The main twist is allowing deletion. This is a step beyond the normal Bloom Filter, and according to a research paper by Tarkoma et al. summarizing Bloom Filter research over the years, there are at least eight existing Bloom Filter variations that allow for deletion. Therefore it is up to the student to decide which works best in the context of the problem. In addition, the larger test case contains large amounts of data, which requires the choice of a fast hashing method and careful choices in parsing and data storage in order to not run out of memory.

Key Solution Ideas

Our primary goal was to require a Bloom Filter that minimizes false positives. In order to achieve this, deletion must be supported in the Bloom Filter used in the solution. We accomplished this in our own solution with a collision array in all three languages. When an element is added to the Bloom Filter, each of the *k* hash functions will compute an index which is then set to one in the bit array representing the data structure. However, if an element is hashed to an already-taken index, the corresponding cell in the collision array is marked as True to indicate that bit in the original array *m* is being shared between multiple elements. When an element is deleted, the element is hashed to find its *k* indices in the bit array. For each index *i*, if collision[i] is *False*, then that bit in *m* is not shared with other elements and therefore can be changed to zero. Furthermore, in parsing the data, multiple bloom filters of different sizes are required in order to keep track of blacklisted IPs, bad data packets, and non-blacklisted IPs. The non-blacklisted IPs must be removed and added to the blacklisted IPs if they send 3+ bad packets. Without the use of multiple Bloom Filters, a student would run out of memory in more traditional set data structures for the large test cases.

This is only one potential solution to the deletion problem. Other solutions include a counting Bloom Filter. In a Counting Bloom Filter, each bit in a traditional Bloom Filter is replaced by a small counter, which tracks how many times a hash position has been set, allowing both insertions and deletions to be performed by incrementing or decrementing the counters accordingly. The counters start at zero, and during insertions, they increase as elements are added. During deletions, they decrease, and if a counter reaches zero, the corresponding bit is reset. To minimize the risk of counter overflow for numerous repeats, 4-bit counters suffice for most practical scenarios, with an extremely low probability of overflow. The tradeoff this presents compared to the collision array is that it takes up more space, but is 100% accurate in deleting members. For instance, if two members are deleted that have a collision, the collision array would not clear this bit, while the counting Bloom Filter would reset the bit entirely. However, the entire data structure is non-deterministic so more often than not speed and size are preferred over 100% correctness, which is why this solution was not used.

Another way to handle deletions is by maintaining a separate Bloom Filter that contains the values that are deleted, and prioritizing membership of that filter over the good IP filter. This introduces a greater possibility of false negatives, as collisions here indicate that a value has been removed when it actually has not. Since this structure takes up the same amount of space as the collision array, it was not used as it causes even more error in the result.

Conclusion

A Bloom Filter is a non-deterministic set-membership data structure that can only return “is not a member” or “is probably a member”. This result is a function of probability *P*, which is dependent on the size of the filter in relation to the number of elements added. Bloom Filters are used because of their speed and size, allowing them to be distributed with applications, such as

browsers. They work by utilizing a fast hash function to compute evenly distributed indices for an input, and setting those values to 1, indicating that that bit is set. The contains method uses the same hashing functions, and instead checks if all of the bits are set. This process results in the possibility of a false positive, but this tradeoff is desirable for the extremely small size of the data structure. Some functionality, such as deletions, are not available in a regular Bloom Filter implementation, but extra data can be stored to allow for this operation. However, depending on how it is implemented, false negatives can be introduced or the size of the structure will increase by at least roughly double, taking away from the primary use-cases. Regardless, deletion is an interesting concept to consider, since it is a viable operation in many other set-based structures.

Sources

1) https://www.dca.fee.unicamp.br/~chesteve/pubs/bloom-filter-ieee-survey-preprint.pdf
2) https://systemdesign.one/bloom-filters-explained/#advantages-of-bloomfilter
3) https://en.wikipedia.org/wiki/Bloom_filter (it was shockingly complete)