The Bloom Filter:

A probabilistic approach to “maybe”.

### Summary

Everyone likes an easy yes or no answer, sometimes, that’s just not possible. The Bloom Filter, created by Burton Howard Bloom in 1970 (2), is a probabilistic data structure. It can be used to easily determine if a given piece of data has not already been seen by the filter, but it cannot as easily determine if the filter has already seen the data in question. This sounds counterintuitive, but it does provide at least one half of a definitive answer set, and that answer can be used very effectively in applications of data deduplication, network optimization, and other large scale data search tasks.

#### History

The Bloom filter is a member of a group of data structures termed “probabilistic data structures”. These data structures function on probabilities, and provide data based upon the underlying mathematics of their implementation. In the case of the Bloom filter, it was conceived in 1970 by Burton Howard Bloom to allow the examination of large data sets that could not fit into system memory, even when organized into hash tables. Bloom suggested the use of a bit array and multiple hashes of a single piece of data to reduce memory use, but allow for reasonable probability that a given piece of data is not in a set. This approach allowed for far less data access to disk and sped up the search for data across the entire set (2).

#### Algorithm

The ideas behind the Bloom filter are simple, but offer some complexities when pushed to include more functionality. The basic Bloom filter discussed here is a bit array, a bitset, initially set to zeroes, of length n. N is typically an overestimation of the actual number of elements required to allow for a good distribution across the array. The bits of the array are set based upon a hashing function, k, output for a given segment of data. The segment is hashed with either multiple functions or with incremented seeds to the same hashing function, and each of those values then sets the bit array to “1” for the corresponding index.

When new data is brought in, it is hashed and the output of the hash is checked against each hash function, or seed value of the bit array. If all bits are present, the data segment “may” be present in the current data set. If any of the bits are set to “0”, the data segment is definitively NOT in the output data set at this time. A visualization of the this process looks like:

Our initialized, zeroed out, bit array:

0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Sending data into our hashing function - hash(value, seed):

hash(x,1) = 3

hash(x,2) = 6

hash(x,3) = 0

From this output, set the bits in the array:

0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |

Now, to make use of these values, attempt to determine if another element is present by hashing against the same seed values for our hashing function:

hash(y,1) = 2

hash(y,2) = 3

hash(y,3) = 6

Examining the bit array reveals that the bits are indexes 6 and 3 are set, but index 2 is not. Definitively, since the index at 2 is not set, we can say that this value is not present. In the case where bits in the array are set for each run of the hash, we cannot definitively say that the element is present. The reason for this is that the values may collide. To avoid this as much as possible in this simple implementation, the value must be hashed a greater number of times. The more hashes that are computed, the more indexes that will be checked, allowing for greater probability of the correctness of the answer as to whether or not the element has been seen previously. To determine the optimal number of hash functions (1):

k = number of hashes

m = number of bits, or length, of the bit array filter

n = the estimated number of elements expected

e = 2.71828

p = percentage probability of a false positive

k = m / n \* log(2)

3.32 = 10 / 2 \* 0.693147

There is also a way to calculate the most efficient length of a filter for a given number of inputs and a desired probability of false positives (1):

m = -n\*log(p) / (log(2)2)

10 = -2 \* log(.1) / (log(2)2)

The calculation to determine the false positive rate of a given filter implementation is (1):

p = (1-e-kn/m)k

0.14 = (1-e-3\*2/10)3

The probability of a false positive with the above implementation is 14%. That’s a bit high, but given the size of the filter and input set in the example, this is to be expected.

The intended point here is that the filter is tunable by values of m, or the length of the filter. The larger the filter, the larger the capacity for values to be hashed, and the decrease in the probability of a collision, allowing for greater confidence in the positive case. The negative case will always be consistent and true. If one index is not set, the value is NOT present.

#### Hashes, the Keys to the Kingdom

At the core of the Bloom filter’s efficiency is the hashing function used to determine the key values for insertion and comparison. The hashing function in question should be fast, flexible and as collision free as possible. The implementation accompanying this paper uses MurmurHash, a function written by Austin Appleby in 2008 (4). MurmurHash is targeted for use in creating keys for applications like the Bloom filter. The difference between a hashing function like MurmurHash and a secure hashing function like SHA-1 is the intent in the implementation. MurmurHash intends to very quickly create a very low collision value for a given input. The focus is speed and individuation of the result. For SHA-1, security is the intent and the byproduct of that intent is loss of speed. The secure hashing algorithm will use very large primes to compute a hash that is irreversible without the correct key value to interpret the result. The MurmurHash intends to produce a result that is quick, but not necessarily cryptographically useful. It may not be reversible, and there is no key pair in use, since the output of the function is only intended for comparison.

MurmurHash is somewhat old, and has been superseded by work done at Google on FarmHash and, more recently, CityHash. This work was done to counter the finding that Murmurhash is susceptible to Distributed Denial of Service Hash attacks (4). In this case, if data is crafted with knowledge of the hashing algorithm, the results can be constructed such that all hashed values occur in order, and reduce any hash table using MurmurHash to a simple array. Lookup times will increase from O(1) to O(n), effectively. Yet MurmurHash is still used in large scale applications such as Cassandra, Hadoop and Elasticsearch (4).

#### Application and Implementation

Bloom filters are used in an array of applications, from data reduction to determination of presence in a large data set. Some examples include data deduplication in storage. In this instance the filters are used against hashed fingerprints of data segments to determine if a given segment already exists in storage, and therefore does not need to be written again, simply referenced. In the case of network optimization, the case is similar, where two endpoints will exchange a hash of a packet, instead of the packet itself. If the hash exists in the receiving endpoint, it is not sent, only generated on the receiver and pushed up the stack. Bloom filters are also used on large data sets such as bioinformatics data. Since these datasets can become quite large, the ability to quickly and reasonably determine if a given value exists in the data set is valuable in research. Since some of the data is quite large, a hash of the data is also far more efficient in searching the body of the work rather than searching against the entirety of the data set. The Bloom filter in this case can allow for narrowing of search areas within a large data set. This is also the case in some uses pertaining to “big data” applications, such as Hadoop and other large data store and search technologies.

This implementation used a method, outlined by Mitzenmacher and Kirsch in their paper, “Less Hashing, Same Performance” (3). This method allows for only two hashes against the target element, but multiple, individuated outputs given the incremental addition in the formula used. Interestingly, it is (3):

int[] hashes = new int[numberOfHashesRequired];

long hash1 = hash(element, 0);

long hash2 = hash(element, hash1);

for (int i = 0; i < numberOfHashesRequired; i++) {

hashes[i] = Math.abs((hash1 + 1 \* hash2) % lengthOfFilter);

}

This formula allows for only two hashing runs against the element, but computes reasonable output values given the incrementing of the value of i and the multiplication of the hash values modulo the length of the filter for what would typically be successive hashes. The expense of hashing is saved, and the simple math operations on the already created hashes affords a quicker method to set bit fields in the filter for fast lookup and decisions making.

#### Pitfalls

The Bloom filter and its associated hashing algorithms are efficient and surprisingly simplistic structures that produce incredible results against a large set of data. Yet, there are downsides to all of this efficiency and space savings. The largest one that becomes apparent with a few moments examination of the structure and algorithm is the removal of values from the filter. Since multiple values may share a given key in the filter, resetting the bit for a given hash value may invalidate more than one element’s presence in the filter. There are ways around this issue, but they are beyond the reach of this paper.

Along with the removal of elements, there are the issues related to hashing, one of which is spelled out in the issue related to MurmurHash. If not selected carefully, the hashing algorithm can cause the filter to become inefficient and cause best case lookup performance equivalent to an iteration of an array.

#### Conclusions

Bloom filters are an interesting, significant data structure that enables the abbreviation of operations on large scale data. The application of this technology in so many varying fields is a testament to the flexibility and usefulness of computer science research and the application of theory to practice. Without Bloom filters, web crawling, data science, data mining, genetic research, drug research and numerous other fields would be far less productive and agile in their innovations. Bloom filters are not the only data structure used to increase efficiency and efficacy of these efforts, but they are one of the most well known and well researched.

The ease of designing and tuning a Bloom filter is also an appealing aspect of the structure and the math to do so is reasonably simple. This allows for easy use of the structure and reasonable design decisions to be made quickly and effectively. The Bloom filter has not seen the last of its application in computing, it stands as a model of usefulness and efficiency.

## Bibliography

1. Cao P. Bloom filters - the math. University of Wisconsin. http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html. Accessed December 10, 2016.
2. Srivastav P. Bloom filters for dummies. http://prakhar.me/articles/bloom-filters-for-dummies/. Accessed December 10, 2016.
3. Mitzenmacher M. Less Hashing, Same Performance: Building a Better Bloom Filter. Harvard University. https://www.eecs.harvard.edu/~michaelm/postscripts/rsa2008.pdf. Accessed December 10, 2016.
4. Boßlet M. Breaking murmur: Hash-flooding DoS reloaded. Breaking Murmur: Hash-flooding DoS Reloaded. https://emboss.github.io/blog/2012/12/14/breaking-murmur-hash-flooding-dos-reloaded/. Accessed December 12, 2016.