

---

## Boom Filters



**Boom Filters** are probabilistic data structures for processing continuous, unbounded streams. This includes **Stable Bloom Filters**, **Scalable Bloom Filters**, **Counting Bloom Filters**, **Inverse Bloom Filters**, **Cuckoo Filters**, several variants of **traditional Bloom filters**, **HyperLogLog**, **Count-Min Sketch**, and **MinHash**.

Classic Bloom filters generally require a priori knowledge of the data set in order to allocate an appropriately sized bit array. This works well for offline processing, but online processing typically involves unbounded data streams. With enough data, a traditional Bloom filter “fills up”, after which it has a false-positive probability of 1.

Boom Filters are useful for situations where the size of the data set isn’t known ahead of time. For example, a Stable Bloom Filter can be used to deduplicate events from an unbounded event stream with a specified upper bound on false positives and minimal false negatives. Alternatively, an Inverse Bloom Filter is ideal for deduplicating a stream where duplicate events are relatively close together. This results in no false positives and, depending on how close together duplicates are, a small probability of false negatives. Scalable Bloom Filters place a tight upper bound on false positives while avoiding false negatives but require allocating memory proportional to the size of the data set. Counting Bloom Filters and Cuckoo Filters are useful for cases which require adding and removing elements to and from a set.

For large or unbounded data sets, calculating the exact cardinality is impractical. HyperLogLog uses a fraction of the memory while providing an accurate approximation. Similarly, Count-Min Sketch provides an efficient way to estimate event frequency for data streams, while Top-K tracks the top-k most frequent elements.

MinHash is a probabilistic algorithm to approximate the similarity between two sets. This can be used to cluster or compare documents by splitting the corpus into a bag of words.

### Installation

```
1 $ go get github.com/tylertreat/BoomFilters
```

### Stable Bloom Filter

This is an implementation of Stable Bloom Filters as described by Deng and Rafiei in Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters.

---

A Stable Bloom Filter (SBF) continuously evicts stale information so that it has room for more recent elements. Like traditional Bloom filters, an SBF has a non-zero probability of false positives, which is controlled by several parameters. Unlike the classic Bloom filter, an SBF has a tight upper bound on the rate of false positives while introducing a non-zero rate of false negatives. The false-positive rate of a classic Bloom filter eventually reaches 1, after which all queries result in a false positive. The stable-point property of an SBF means the false-positive rate asymptotically approaches a configurable fixed constant. A classic Bloom filter is actually a special case of SBF where the eviction rate is zero and the cell size is one, so this provides support for them as well (in addition to bitset-based Bloom filters).

Stable Bloom Filters are useful for cases where the size of the data set isn't known a priori and memory is bounded. For example, an SBF can be used to deduplicate events from an unbounded event stream with a specified upper bound on false positives and minimal false negatives.

## Usage

```
1 package main
2
3 import (
4     "fmt"
5     "github.com/tylertreat/BoomFilters"
6 )
7
8 func main() {
9     sbf := boom.NewDefaultStableBloomFilter(10000, 0.01)
10    fmt.Println("stable point", sbf.StablePoint())
11
12    sbf.Add([]byte{'a'})
13    if sbf.Test([]byte{'a'}) {
14        fmt.Println("contains a")
15    }
16
17    if !sbf.TestAndAdd([]byte{'b'}) {
18        fmt.Println("doesn't contain b")
19    }
20
21    if sbf.Test([]byte{'b'}) {
22        fmt.Println("now it contains b!")
23    }
24
25    // Restore to initial state.
26    sbf.Reset()
27 }
```

---

## Scalable Bloom Filter

This is an implementation of a Scalable Bloom Filter as described by Almeida, Baquero, Pregoica, and Hutchison in Scalable Bloom Filters.

A Scalable Bloom Filter (SBF) dynamically adapts to the size of the data set while enforcing a tight upper bound on the rate of false positives and a false-negative probability of zero. This works by adding Bloom filters with geometrically decreasing false-positive rates as filters become full. A tightening ratio,  $r$ , controls the filter growth. The compounded probability over the whole series converges to a target value, even accounting for an infinite series.

Scalable Bloom Filters are useful for cases where the size of the data set isn't known a priori and memory constraints aren't of particular concern. For situations where memory is bounded, consider using Inverse or Stable Bloom Filters.

The core parts of this implementation were originally written by Jian Zhen as discussed in Benchmarking Bloom Filters and Hash Functions in Go.

## Usage

```
1 package main
2
3 import (
4     "fmt"
5     "github.com/tylertreat/BoomFilters"
6 )
7
8 func main() {
9     sbf := boom.NewDefaultScalableBloomFilter(0.01)
10
11     sbf.Add([]byte(`a`))
12     if sbf.Test([]byte(`a`)) {
13         fmt.Println("contains a")
14     }
15
16     if !sbf.TestAndAdd([]byte(`b`)) {
17         fmt.Println("doesn't contain b")
18     }
19
20     if sbf.Test([]byte(`b`)) {
21         fmt.Println("now it contains b!")
22     }
23
24     // Restore to initial state.
25     sbf.Reset()
26 }
```

---

## Inverse Bloom Filter

An Inverse Bloom Filter, or “the opposite of a Bloom filter”, is a concurrent, probabilistic data structure used to test whether an item has been observed or not. This implementation, originally described and written by Jeff Hodges, replaces the use of MD5 hashing with a non-cryptographic FNV-1 function.

The Inverse Bloom Filter may report a false negative but can never report a false positive. That is, it may report that an item has not been seen when it actually has, but it will never report an item as seen which it hasn’t come across. This behaves in a similar manner to a fixed-size hashmap which does not handle conflicts.

This structure is particularly well-suited to streams in which duplicates are relatively close together. It uses a CAS-style approach, which makes it thread-safe.

## Usage

```
1 package main
2
3 import (
4     "fmt"
5     "github.com/tylertreat/BoomFilters"
6 )
7
8 func main() {
9     ibf := boom.NewInverseBloomFilter(10000)
10
11     ibf.Add([]byte(`a`))
12     if ibf.Test([]byte(`a`)) {
13         fmt.Println("contains a")
14     }
15
16     if !ibf.TestAndAdd([]byte(`b`)) {
17         fmt.Println("doesn't contain b")
18     }
19
20     if ibf.Test([]byte(`b`)) {
21         fmt.Println("now it contains b!")
22     }
23 }
```

## Counting Bloom Filter

This is an implementation of a Counting Bloom Filter as described by Fan, Cao, Almeida, and Broder in Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol.

---

A Counting Bloom Filter (CBF) provides a way to remove elements by using an array of n-bit buckets. When an element is added, the respective buckets are incremented. To remove an element, the respective buckets are decremented. A query checks that each of the respective buckets are non-zero. Because CBFs allow elements to be removed, they introduce a non-zero probability of false negatives in addition to the possibility of false positives.

Counting Bloom Filters are useful for cases where elements are both added and removed from the data set. Since they use n-bit buckets, CBFs use roughly n-times more memory than traditional Bloom filters.

See Deletable Bloom Filter for an alternative which avoids false negatives.

## Usage

```
1 package main
2
3 import (
4     "fmt"
5     "github.com/tylertreat/BoomFilters"
6 )
7
8 func main() {
9     bf := boom.NewDefaultCountingBloomFilter(1000, 0.01)
10
11     bf.Add([]byte(`a`))
12     if bf.Test([]byte(`a`)) {
13         fmt.Println("contains a")
14     }
15
16     if !bf.TestAndAdd([]byte(`b`)) {
17         fmt.Println("doesn't contain b")
18     }
19
20     if bf.TestAndRemove([]byte(`b`)) {
21         fmt.Println("removed b")
22     }
23
24     // Restore to initial state.
25     bf.Reset()
26 }
```

## Cuckoo Filter

This is an implementation of a Cuckoo Filter as described by Andersen, Kaminsky, and Mitzenmacher in Cuckoo Filter: Practically Better Than Bloom. The Cuckoo Filter is similar to the Counting Bloom Fil-

---

ter in that it supports adding and removing elements, but it does so in a way that doesn't significantly degrade space and performance.

It works by using a cuckoo hashing scheme for inserting items. Instead of storing the elements themselves, it stores their fingerprints which also allows for item removal without false negatives (if you don't attempt to remove an item not contained in the filter).

For applications that store many items and target moderately low false-positive rates, cuckoo filters have lower space overhead than space-optimized Bloom filters.

## Usage

```
1 package main
2
3 import (
4     "fmt"
5     "github.com/tylertreat/BoomFilters"
6 )
7
8 func main() {
9     cf := boom.NewCuckooFilter(1000, 0.01)
10
11     cf.Add([]byte(`a`))
12     if cf.Test([]byte(`a`)) {
13         fmt.Println("contains a")
14     }
15
16     if contains, _ := cf.TestAndAdd([]byte(`b`)); !contains {
17         fmt.Println("doesn't contain b")
18     }
19
20     if cf.TestAndRemove([]byte(`b`)) {
21         fmt.Println("removed b")
22     }
23
24     // Restore to initial state.
25     cf.Reset()
26 }
```

## Classic Bloom Filter

A classic Bloom filter is a special case of a Stable Bloom Filter whose eviction rate is zero and cell size is one. We call this special case an Unstable Bloom Filter. Because cells require more memory overhead, this package also provides two bitset-based Bloom filter variations. The first variation is the traditional

---

implementation consisting of a single bit array. The second implementation is a partitioned approach which uniformly distributes the probability of false positives across all elements.

Bloom filters have a limited capacity, depending on the configured size. Once all bits are set, the probability of a false positive is 1. However, traditional Bloom filters cannot return a false negative.

A Bloom filter is ideal for cases where the data set is known a priori because the false-positive rate can be configured by the size and number of hash functions.

## Usage

```
1 package main
2
3 import (
4     "fmt"
5     "github.com/tylertreat/BoomFilters"
6 )
7
8 func main() {
9     // We could also use boom.NewUnstableBloomFilter or boom.
10    // NewPartitionedBloomFilter.
11    bf := boom.NewBloomFilter(1000, 0.01)
12
13    bf.Add([]byte{'a'})
14    if bf.Test([]byte{'a'}) {
15        fmt.Println("contains a")
16    }
17
18    if !bf.TestAndAdd([]byte{'b'}) {
19        fmt.Println("doesn't contain b")
20    }
21
22    if bf.Test([]byte{'b'}) {
23        fmt.Println("now it contains b!")
24    }
25
26    // Restore to initial state.
27    bf.Reset()
28 }
```

## Count-Min Sketch

This is an implementation of a Count-Min Sketch as described by Cormode and Muthukrishnan in An Improved Data Stream Summary: The Count-Min Sketch and its Applications.

A Count-Min Sketch (CMS) is a probabilistic data structure which approximates the frequency of events

---

in a data stream. Unlike a hash map, a CMS uses sub-linear space at the expense of a configurable error factor. Similar to Counting Bloom filters, items are hashed to a series of buckets, which increment a counter. The frequency of an item is estimated by taking the minimum of each of the item's respective counter values.

Count-Min Sketches are useful for counting the frequency of events in massive data sets or unbounded streams online. In these situations, storing the entire data set or allocating counters for every event in memory is impractical. It may be possible for offline processing, but real-time processing requires fast, space-efficient solutions like the CMS. For approximating set cardinality, refer to the HyperLogLog.

## Usage

```
1 package main
2
3 import (
4     "fmt"
5     "github.com/tylertreat/BoomFilters"
6 )
7
8 func main() {
9     cms := boom.NewCountMinSketch(0.001, 0.99)
10
11     cms.Add([]byte(`alice`)).Add([]byte(`bob`)).Add([]byte(`bob`)).Add(
12         []byte(`frank`))
13     fmt.Println("frequency of alice", cms.Count([]byte(`alice`)))
14     fmt.Println("frequency of bob", cms.Count([]byte(`bob`)))
15     fmt.Println("frequency of frank", cms.Count([]byte(`frank`)))
16
17     // Serialization example
18     buf := new(bytes.Buffer)
19     n, err := cms.WriteDataTo(buf)
20     if err != nil {
21         fmt.Println(err, n)
22     }
23
24     // Restore to initial state.
25     cms.Reset()
26
27     newCMS := boom.NewCountMinSketch(0.001, 0.99)
28     n, err = newCMS.ReadDataFrom(buf)
29     if err != nil {
30         fmt.Println(err, n)
31     }
32
33     fmt.Println("frequency of frank", newCMS.Count([]byte(`frank`)))
34 }
```



---

```
35  
36 }
```

## Top-K

Top-K uses a Count-Min Sketch and min-heap to track the top-k most frequent elements in a stream.

## Usage

```
1 package main  
2  
3 import (  
4     "fmt"  
5     "github.com/tylertreat/BoomFilters"  
6 )  
7  
8 func main() {  
9     topk := boom.NewTopK(0.001, 0.99, 5)  
10  
11     topk.Add([]byte(`bob`)).Add([]byte(`bob`)).Add([]byte(`bob`))  
12     topk.Add([]byte(`tyler`)).Add([]byte(`tyler`)).Add([]byte(`tyler`))  
13     topk.Add([]byte(`fred`))  
14     topk.Add([]byte(`alice`)).Add([]byte(`alice`)).Add([]byte(`alice`))  
15     topk.Add([]byte(`james`))  
16     topk.Add([]byte(`fred`))  
17     topk.Add([]byte(`sara`)).Add([]byte(`sara`))  
18     topk.Add([]byte(`bill`))  
19  
20     for i, element := range topk.Elements() {  
21         fmt.Println(i, string(element.Data), element.Freq)  
22     }  
23  
24     // Restore to initial state.  
25     topk.Reset()  
26 }
```

## HyperLogLog

This is an implementation of HyperLogLog as described by Flajolet, Fusy, Gandouet, and Meunier in [HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm](#).

HyperLogLog is a probabilistic algorithm which approximates the number of distinct elements in a multiset. It works by hashing values and calculating the maximum number of leading zeros in the

---

binary representation of each hash. If the maximum number of leading zeros is  $n$ , the estimated number of distinct elements in the set is  $2^n$ . To minimize variance, the multiset is split into a configurable number of registers, the maximum number of leading zeros is calculated in the numbers in each register, and a harmonic mean is used to combine the estimates.

For large or unbounded data sets, calculating the exact cardinality is impractical. HyperLogLog uses a fraction of the memory while providing an accurate approximation.

This implementation was originally written by Eric Lesh. Some small changes and additions have been made, including a way to construct a HyperLogLog optimized for a particular relative accuracy and adding FNV hashing. For counting element frequency, refer to the Count-Min Sketch.

## Usage

```
1 package main
2
3 import (
4     "fmt"
5     "github.com/tylertreat/BoomFilters"
6 )
7
8 func main() {
9     hll, err := boom.NewDefaultHyperLogLog(0.1)
10    if err != nil {
11        panic(err)
12    }
13
14    hll.Add([]byte(`alice`)).Add([]byte(`bob`)).Add([]byte(`bob`)).Add(
15        ([]byte(`frank`)))
16    fmt.Println("count", hll.Count())
17
18    // Serialization example
19    buf := new(bytes.Buffer)
20    _, err = hll.WriteDataTo(buf)
21    if err != nil {
22        fmt.Println(err)
23    }
24
25    // Restore to initial state.
26    hll.Reset()
27
28    newHll, err := boom.NewDefaultHyperLogLog(0.1)
29    if err != nil {
30        fmt.Println(err)
31    }
32
33    _, err = newHll.ReadDataFrom(buf)
34    if err != nil {
```

---

```
34     fmt.Println(err)
35 }
36     fmt.Println("count", newHll.Count())
37
38 }
```

## MinHash

This is a variation of the technique for estimating similarity between two sets as presented by Broder in On the resemblance and containment of documents.

MinHash is a probabilistic algorithm which can be used to cluster or compare documents by splitting the corpus into a bag of words. MinHash returns the approximated similarity ratio of the two bags. The similarity is less accurate for very small bags of words.

## Usage

```
1 package main
2
3 import (
4     "fmt"
5     "github.com/tylertreat/BoomFilters"
6 )
7
8 func main() {
9     bag1 := []string{"bill", "alice", "frank", "bob", "sara", "tyler",
10                     "james"}
11     bag2 := []string{"bill", "alice", "frank", "bob", "sara"}
12     fmt.Println("similarity", boom.MinHash(bag1, bag2))
13 }
```

## References

- Approximately Detecting Duplicates for Streaming Data using Stable Bloom Filters
- Scalable Bloom Filters
- The Opposite of a Bloom Filter
- Benchmarking Bloom Filters and Hash Functions in Go
- Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol
- An Improved Data Stream Summary: The Count-Min Sketch and its Applications
- HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm
- Package hyperloglog

- 
- On the resemblance and containment of documents
  - Cuckoo Filter: Practically Better Than Bloom