**bloom filter**

**MINI PROJECT REPORT**

**21CSC201J– DATA STRUCTURES AND ALGORITHMS**

**(2021 Regulation)**

**II Year/ III Semester**

**Academic Year: 2022 -2023**

By

**A. JENNIE PRIYANKA (RA2112702010001)**

**N. NARENDRAN (RA2112702010005)**



**FACULTY OF ENGINEERING AND TECHNOLOGY**

**SCHOOL OF COMPUTING**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**Kattankulathur, Kancheepuram**

**NOVEMBER 2022**

**BONAFIDE**

This is to certify that the project report titled “**BLOOM FILTER**” is the bonafide work  of  **JENNIE PRIYANKA (RA2112702010001) NARENDRA (RA2112702010005)** who undertook the task of completing the project within the allotted time. Certified further, that to the best of my knowledge the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

Dr. K.KALAISELVI DR. ANNAPURANI PANAIYAPPAN .K

GUIDE HEAD OF THE DEPARTMENT

Assistant Professor, Networking Networking and Communications

and Communications

**INTERNAL EXAMINER EXTERNAL EXAMINER**

**Signature of the Course Faculty**

Dr. K. Kalaiselvi

**Assistant Professor**

Department of NWC,

SRM Institute of Science and Technology

**ABSTRACT**

A Bloom filter is a **space-efficient probabilistic** data structure that is used to test whether an element is a member of a set. For example, checking availability of username is set membership problem, where the set is the list of all registered username. The price we pay for efficiency is that it is probabilistic in nature that means, there might be some False Positive results. **False positive means**, it might tell that given username is already taken but actually it’s not. **Interesting Properties of Bloom Filters**

Unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements.Adding an element never fails. However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result .Bloom filters never generate **false negative** result, i.e., telling you that a username doesn’t exist when it actually exists.

Deleting elements from filter is not possible because, if we delete a single element by clearing bits at indices generated by k hash functions, it might cause deletion of few other elements. Example – if we delete “geeks” (in given example below) by clearing bit at 1, 4 and 7, we might end up deleting “nerd” also Because bit at index 4 becomes 0 and bloom filter claims that “nerd” is not present.

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **1** | **Introduction** |  |
| **2** | **System Design** |  |
| **3** | **Implementation** |  |
| **4** | **Conclusion** |  |

**CHAPTER 1**

**1.1 INTRODUCTION :**

A “bloom filter” is a probabilistic data structure that is used to test whether an item is a member of a set. A bloom filter that has been populated with a set of items is able to give one of two responses when asked if an item is a member of the set:

1. The item is **definitely not** in the set.
2. The item is **possibly** in the set.

When a bloom filter is populated with a set of items, it does not store copies of the items themselves (more on this later). It is instead able to remember which items have been added using an incredibly space-efficient data structure (which is usually orders of magnitude smaller than the original dataset). This comes with the tradeoff of being “probabilistic”. It can say with certainty that an item is **not** in the set, but it can’t say with certainty whether an item **is** in the set. With careful planning, however, the level of uncertainty can be bounded.

**1.2 PROBLEM STATEMENT :**

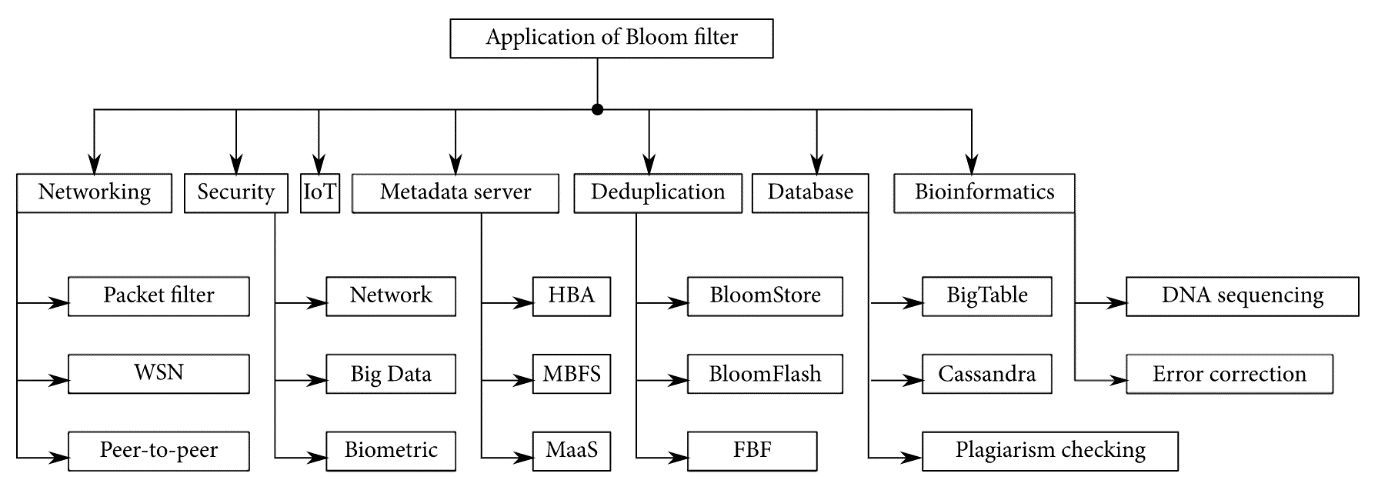
Wherever a list or set is used, and space is at a premium, consider using a Bloom filter if the effect of false positives can be mitigated.

**1.3 PROBLEM DEFFINITION :**

Multiple elements can set the same bit in the filter. Therefore, elements once added can't be removed from the filter. Neither can we obtain the list of elements added. Standard Bloom filter is not suited for applications that need these features

**1.4 EXISTING SYSTEM :**

Which are some real-world applications of the Bloom filter?



**Some applications of Bloom filter. Source: Patgiri et al. 2019, fig. 5.**

For network-related applications, Bloom filter is used to collaborate in peer-to-peer networks, resource routing, packet routing, and measurements. Content Delivery Networks (CDNs) use Bloom filters to avoid caching files seen only once.

For applications that use databases, Bloom filter enables efficient searches, privacy preservation, content synchronization, and duplicate detection.

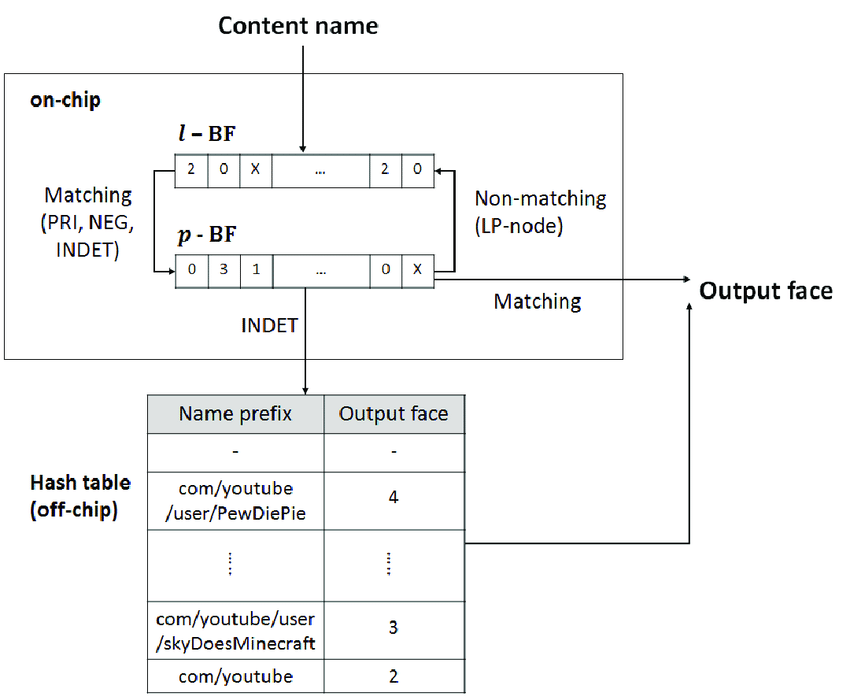
Medium uses Bloom filter to deduplicate recommendations. In a streaming application processing 17 million events per day per partition, Bloom filter was used to deduplicate events at scale. The filter needed 108Gb across 1024 partitions. Reads and writes were 20x and 3x faster respectively.

Chrome browser uses the filter to represent a set of malicious URLs. When user requests a URL, if the filter indicates a probable match, the request is sent to a server to check if the URL is indeed safe.

In a web application, Bloom filters could be used to keep count of visitors coming from a specific city accessing a webpage

**CHAPTER 2**

**2.1 ARCHIETUTRE DIAGRAM :**

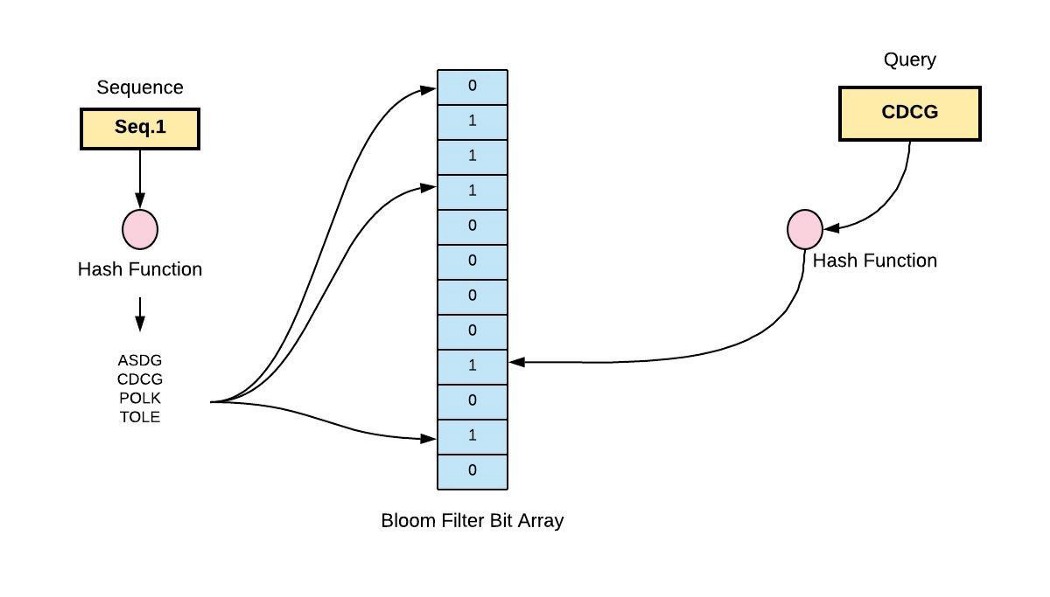


**2.2 USE CASE :**

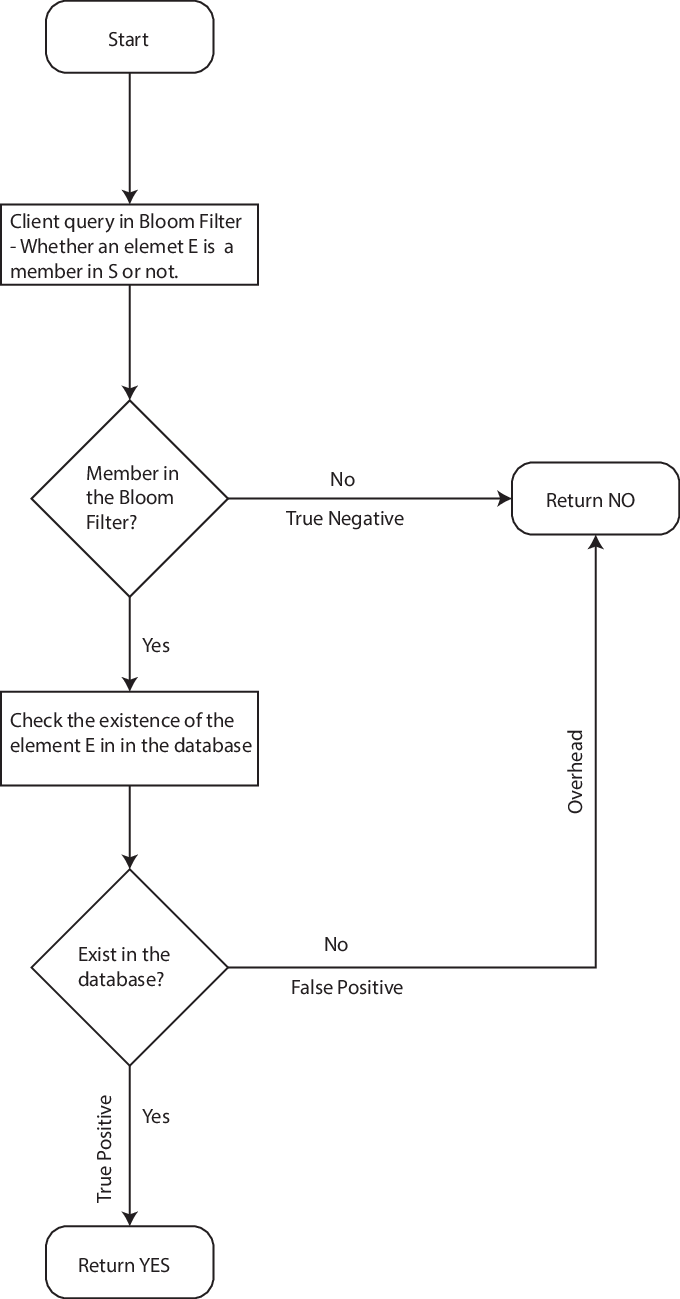
Bloom filters are used to quickly check if an item is present in a large dataset. This is usually done to avoid an expensive operation involving manually searching through the entire dataset.

Here a few concrete examples of when a bloom filter might be helpful:

* Checking if a username is already in use during signup.
* Avoiding recommending content to a user that they have lready seen.
* Bypassing expensive network or disk lookups for non-existent keys.
* Verifying whether an email address is in an unsubscription list



**2.3 FLOW DIAGRAM:**



**CHAPTER 3**

**IMPLEMENTATION :**

How do I select the parameters of the Bloom filter?

There are essentially three parameters: filter size mm, number of hash functions kk and number of elements to be stored nn.

For lower false positives, use higher values of mm and kk with a tradeoff. Higher mm means more memory consumption. Higher kk means more computation. Since higher kk sets more bits, there’s an optimal value koptkopt beyond which false positives start to increase.

When adding elements beyond the design limit, false positives go up. To achieve a fixed false positive probability, we should linearly scale mm as nn increases. In fact, the bit-to-element ratio is bounded as m/n⩾1/ln(2)m/n⩾1/ln(2).

In practice, we estimate nn in advance for our application. We decide on the false positive probability we’re willing to tolerate. We select mm and calculate kk, or select kk and calculate mm. A [handy calculator is available online](https://hur.st/bloomfilter/).

If parameters are not chosen correctly, an attacker can corrupt the filter with well-chosen inputs.

Which hash functions are suitable for use in a Bloom filter?

Hash functions should ideally be independent, meaning that for the same input they set different bits of the filter. Otherwise, hash collisions become more common, leading to higher false positives. Functions should also distribute the input space uniformly.

Hash functions should be fast. The use of cryptographic hash functions is possible but not really required. Among the non-cryptographic hash functions suited for Bloom filter are MurmurHash, Fowler–Noll–Vo (FNV) and Jenkins. One developer showed how replacing MD5 with MurmurHash brought performance improvements.

It’s possible to generate kk hash functions from just two hash functions. One researcher pointed out that if the filter size is 32 bits, on a 64-bit machine a single hash function can give two 32-bit hash values, thus simulating two hash functions.

**3.2 ALGORITHM :**

Step 1:

Search master index for record address.

Step 2 :

Access record from this master file address.

Step 3:

If this is an update, then update master index, master file, and transaction log.

(a) No differential file

Step 1:

Search master index for record address.

Step 2 :

Access record from either the master file or the differential file, depending on the address obtained in Step 1.

Step 3:

If this is an update, then update master index, differential file, and transaction log.

(b) Differential file in use

Step 1:

Search differential index for record address. If the search is unsuccessful. then search the master index.

Step 2 :

Access record from either the master file or the differential file, depending on the address obtained in Step 1.

Step 3 :

If this is an update, then update differential index, differential file, and transaction log.

(c) Differential index and file in use

Step 1:

Query the Bloom filter. If the answer is "maybe," then search differential index for record address. If the answer is "no' or if the differential index search is unsuccessful, then search the master index.

Step 2:

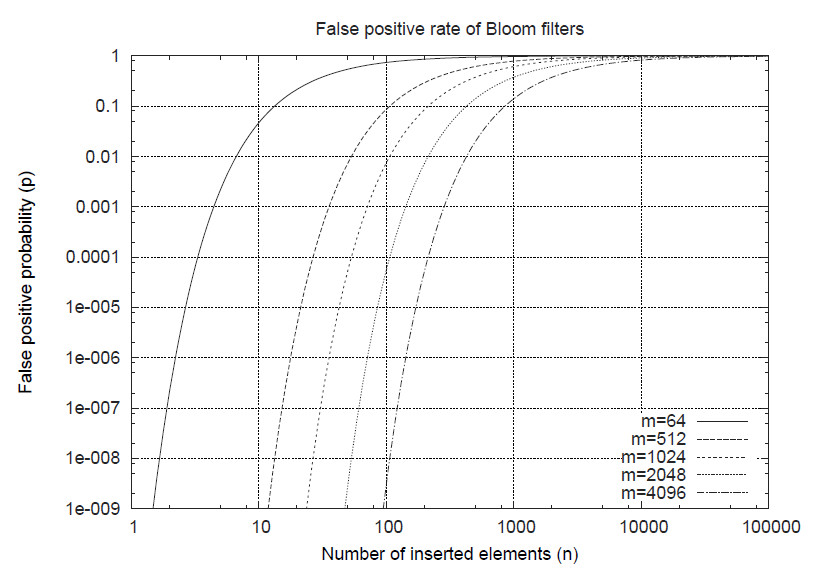
Access record from either the master file or the differential file, depending on the address obtained in Step 1.

Step 3:

If this is an update, then update Bloom filter, differential index, differential file, and transaction log.

(d) Differential index and file and Bloom filter in use

**3.3 PERFORMANCE :**



**False positive rate (FPR) of the Bloom filter. Source: Tarkoma et al. 2012, fig. 3.**

A Bloom filter with one hash function is equivalent to ordinary hashing. Bloom filter is a generalization of hashing. It allows for more interesting design tradeoffs.

The complexity to add an element or to query the filter is fixed at O(k)O(k). In other words, it's independent of how many items have been added to the filter.

Bloom filter simplifies some operations. Suppose we have two filters to represents sets S1S1 and S2S2. The union of the these two sets is simply an OR operation of the two filters. Intersection of the two sets can also be approximated with the filters. Another trick is to half the size of the filter with an OR operation of the first and second halves of the filter.

**3.4 SAMPLE CODE :**

**C++**

#include <bits/stdc++.h>

#define ll long long

**using** **namespace** std;

// hash 1

**int** h1(string s, **int** arrSize)

{

    ll **int** hash = 0;

**for** (**int** i = 0; i < s.size(); i++)

    {

        hash = (hash + ((**int**)s[i]));

        hash = hash % arrSize;

    }

**return** hash;

}

// hash 2

**int** h2(string s, **int** arrSize)

{

    ll **int** hash = 1;

**for** (**int** i = 0; i < s.size(); i++)

    {

        hash = hash + **pow**(19, i) \* s[i];

        hash = hash % arrSize;

    }

**return** hash % arrSize;

}

// hash 3

**int** h3(string s, **int** arrSize)

{

    ll **int** hash = 7;

**for** (**int** i = 0; i < s.size(); i++)

    {

        hash = (hash \* 31 + s[i]) % arrSize;

    }

**return** hash % arrSize;

}

// hash 4

**int** h4(string s, **int** arrSize)

{

    ll **int** hash = 3;

**int** p = 7;

**for** (**int** i = 0; i < s.size(); i++) {

        hash += hash \* 7 + s[0] \* **pow**(p, i);

        hash = hash % arrSize;

    }

**return** hash;

}

// lookup operation

**bool** lookup(**bool**\* bitarray, **int** arrSize, string s)

{

**int** a = h1(s, arrSize);

**int** b = h2(s, arrSize);

**int** c = h3(s, arrSize);

**int** d = h4(s, arrSize);

**if** (bitarray[a] && bitarray[b] && bitarray

        && bitarray[d])

**return** **true**;

**else**

**return** **false**;

}

// insert operation

**void** insert(**bool**\* bitarray, **int** arrSize, string s)

{

    // check if the element in already present or not

**if** (lookup(bitarray, arrSize, s))

        cout << s << " is Probably already present" << endl;

**else**

    {

**int** a = h1(s, arrSize);

**int** b = h2(s, arrSize);

**int** c = h3(s, arrSize);

**int** d = h4(s, arrSize);

        bitarray[a] = **true**;

        bitarray[b] = **true**;

        bitarray = **true**;

        bitarray[d] = **true**;

        cout << s << " inserted" << endl;

    }

}

// Driver Code

**int** main()

{

**bool** bitarray[100] = { **false** };

**int** arrSize = 100;

    string sarray[33]

        = { "abound",   "abounds",       "abundance",

            "abundant", "accessible",    "bloom",

            "blossom",  "bolster",       "bonny",

            "bonus",    "bonuses",       "coherent",

            "cohesive", "colorful",      "comely",

            "comfort",  "gems",          "generosity",

            "generous", "generously",    "genial",

            "bluff",    "cheater",       "hate",

            "war",      "humanity",      "racism",

            "hurt",     "nuke",          "gloomy",

            "facebook", "geeksforgeeks", "twitter" };

**for** (**int** i = 0; i < 33; i++) {

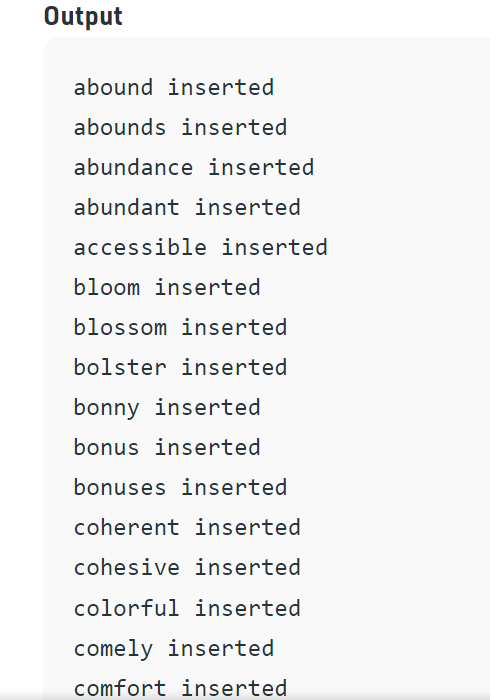
        insert(bitarray, arrSize, sarray[i]);

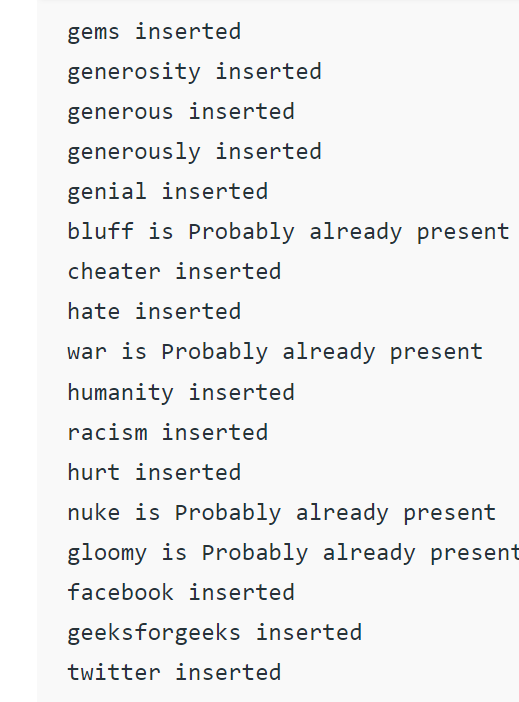
    }

**return** 0;

}

**3.4 OUTPUT :**

****

****

**CHAPTER 4**

**CONCLUSION :**

Bloom filter has inspired dozens of variants. One survey paper from 2019 mentioned more than 60 variants, their capabilities, performance and application areas. A variant might aim to reduce false positives, improve performance or provide additional features.

**Counting Bloom filter** allows us to implement a multiset so that multiple instances of the same element can be stored. Some of these support deletions as well but at the cost of false negatives. Bitwise Bloom filter and spectral Bloom filter improve on the counting Bloom filter.

When elements are added to a filter beyond it's designed capacity, false positives increase. **Scalable Bloom filter** allows us to grow the filter size as more items are added.

When filters have to be exchanged over a network (such as in web caching applications), **Compressed Bloom filter** saves on network bandwidth. By choosing the hash functions and thereby changing the way bits are distributed in the filter, better compression is achieved.

**REFERENCE :**

1. [Akyildiz, Bugra. 2016. "A Gentle Introduction to Bloom Filter." KDNuggets, August. Accessed 2020-05-11.](https://www.kdnuggets.com/2016/08/gentle-introduction-bloom-filter.html)
2. [Almeida, Paulo Sérgio, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. "Scalable Bloom Filters." Information Processing Letters, Elsevier, vol. 101, no. 6, pp. 255-261, March 31. Accessed 2020-05-11.](http://gsd.di.uminho.pt/members/cbm/ps/dbloom.pdf)
3. [Batson, Alan. 1965. "The organization of symbol tables." Communications of the ACM, vol. 8, no. 2, pp. 111-112, February. Accessed 2020-05-12.](https://dl.acm.org/doi/10.1145/363744.363776)
4. [Bloom, Burton H. 1970. "Space/Time Trade-offs in Hash Coding with Allowable Errors." Communications of the ACM, vol. 13, no. 7, pp. 422-426, July. Accessed 2020-05-11.](https://dl.acm.org/doi/pdf/10.1145/362686.362692)
5. [Bose, Prosenjit, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. 2008. "On the false-positive rate of Bloom filters." Information Processing Letters, Elsevier, vol. 108, no. 4, pp. 210-213, October 31. doi:10.1016/j.ipl.2008.05.018. Accessed 2020-05-11.](http://people.scs.carleton.ca/~kranakis/Papers/TR-07-07.pdf)
6. [Broder, Andrei, and Michael Mitzenmacher. 2003. "Network Applications of Bloom Filters: A Survey." Internet Mathematics, vol. 1, no. 4, pp. 485-509, November. Accessed 2020-05-11.](https://www.researchgate.net/publication/220465609_Survey_Network_Applications_of_Bloom_Filters_A_Survey)
7. [Davies, Jason. 2020. "Bloom Filters." Accessed 2020-05-11.](https://www.jasondavies.com/bloomfilter/)
8. [Fan, Li, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. "Summary cache: a scalable wide-area web cache sharing protocol." IEEE/ACM Transactions on Networking, vol. 8, no. 3, pp. 281-293, June. Accessed 2020-05-11.](http://pages.cs.wisc.edu/~jussara/papers/00ton.pdf)
9. [Fitzgerald, William. 2011. "Producing n hash functions by hashing only once." Will.Whim blog, September 3. Accessed 2020-05-11.](http://willwhim.wpengine.com/2011/09/03/producing-n-hash-functions-by-hashing-only-once/)
10. [GeeksforGeeks. 2017. "Bloom Filters – Introduction and Python Implementation." GeeksforGeeks, April 17. Updated 2018-02-08. Accessed 2020-05-11.](https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/)
11. [Gerbet, Thomas, Amrit Kumar, and Cédric Lauradoux. 2014. "The Power of Evil Choices in Bloom Filters." Research Report RR-8627, INRIA Grenoble, hal-01082158v2, November. Accessed 2020-05-11.](https://hal.inria.fr/hal-01082158/document)
12. [Hurst, Thomas. 2018. "Bloom Filter Calculator." October 15. Accessed 2020-05-11.](https://hur.st/bloomfilter/)
13. [Katsov, Ilya. 2012. "Probabilistic Data Structures for Web Analytics and Data Mining." Highly Scalable Blog, May 1. Accessed 2020-05-11.](https://highlyscalable.wordpress.com/2012/05/01/probabilistic-structures-web-analytics-data-mining/)
14. [Kirsch, Adam, and Michael Mitzenmacher. 2006. "Less Hashing, Same Performance: Building a Better Bloom Filter." In Y. Azar and T. Erlebach (Eds.), Proc. of European Symposium on Algorithms (ESA), Springer-Verlag Berlin, Heidelberg, LNCS 4168, pp. 456–467, September 11-13. Accessed 2020-05-11.](https://www.researchgate.net/publication/220770131_Less_Hashing_Same_Performance_Building_a_Better_Bloom_Filter)
15. [Luo, Lailong, Deke Guo, Richard T.B. Ma, Ori Rottenstreich, and Xueshan Luo. 2019. "Optimizing Bloom Filter: Challenges, Solutions, and Comparisons." arXiv, v2, January 7. Accessed 2020-05-11.](https://arxiv.org/abs/1804.04777)
16. [Martí, Vicent. 2012. "Some performance tweaks." bitly/dablooms, on GitHub, August 5. Accessed 2020-05-11.](https://github.com/bitly/dablooms/pull/19)
17. [Nath, Kousik. 2018. "Bloom Filter: A simple but interesting data structure." Data Driven Investor, on Medium, September 23. Accessed 2020-05-11.](https://medium.com/datadriveninvestor/bloom-filter-a-simple-but-interesting-data-structure-37fd53b11606)
18. [Nilsson, Stefan. 2018. "Your basic int: a most powerful data type." Yourbasic.org, February 21. Updated 2019-05-14. Accessed 2020-05-11.](https://yourbasic.org/algorithms/your-basic-int/)
19. [Patgiri, Ripon, Sabuzima Nayak, and Samir Kumar Borgohain. 2019. "Hunting the Pertinency of Bloom Filter in Computer Networking and Beyond: A Survey." Journal of Computer Networks and Communications, Hindawi, February 5. doi:10.1155/2019/2712417. Accessed 2020-05-11.](https://www.hindawi.com/journals/jcnc/2019/2712417/)
20. [Pellow, David, Darya Filippova, and Carl Kingsford. 2017. "Improving Bloom Filter Performance on Sequence Data Using k-mer Bloom Filters." Journal of Computational Biology, 24(6): 547–557, June. Accessed 2020-05-11.](https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5467106/)
21. [Rae, Jack W, Sergey Bartunov, and Timothy P Lillicrap. 2019. "Meta-Learning Neural Bloom Filters." Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97. Accessed 2020-05-11.](http://proceedings.mlr.press/v97/rae19a/rae19a.pdf)
22. [Singh, Gurminder. 2019. "Deduplication at Scale." Amplitude Blog, on Medium, May 24. Accessed 2020-05-11.](https://amplitude.engineering/dedupe-events-at-scale-f9e416e46ca9)
23. [Tarkoma, Sasu, Christian Esteve Rothenberg, and Eemil Lagerspetz. 2012. "Theory and Practice of Bloom Filters for Distributed Systems." IEEE Communications Surveys & Tutorials, vol. 14, no. 1, pp. 131-155, March. Accessed 2020-05-11.](https://www.researchgate.net/publication/224230450_Theory_and_Practice_of_Bloom_Filters_for_Distributed_Systems)
24. [Vallentin, Matthias. 2011. "A Garden Variety of Bloom Filters." Blog, June 14. Updated 2013-07-17. Accessed 2020-05-11.](http://matthias.vallentin.net/blog/2011/06/a-garden-variety-of-bloom-filters/)
25. [Wikipedia. 2020a. "Bloom filter." Wikipedia, April 29. Accessed 2020-05-11.](https://en.wikipedia.org/wiki/Bloom_filter)
26. [Wikipedia. 2020b. "Hash function." Wikipedia, May 6. Accessed 2020-05-12](https://en.wikipedia.org/wiki/Hash_function)