# Bloom Filters – A Tutorial, Analysis, and Survey

**James Blustein**
**Amal El-Maazawi**

Technical Report CS-2002-10

Dec 10, 2002

Faculty of Computer Science
6050 University Ave., Halifax, Nova Scotia, B3H 1W5, Canada

# Bloom Filters — A Tutorial, Analysis, and Survey

Authors: James Blustein* and Amal El-Maazawi
Faculty of Computer Science
Dalhousie University
6050 University Avenue
Halifax, NS
B3H 1W5
Canada
Contact: Telephone: +1(902)494-6104
Facsimile: +1(902)492-1517
E-mail: ⟨jamie@cs.dal.ca⟩

**Abstract**

Bloom filters use superimposed hash transforms to provide a probabilistic membership test. The only types of errors are false positives (non-members being reported as members). Non-members are typically detected quickly (requiring only two probes in the optimal case).

This article surveys modern applications of this technique (e.g., in spell checking and Web caching software) and provides a detailed analysis of their performance, in theory and practice. The article concludes with practical advice about implementing this useful and intriguing technique.

**Keywords:** Bloom filter, hashing, performance analysis, network cache

*Corresponding author

# Contents

**1 Introduction** **1**
1.1 Definition . . . 1
1.2 What Is An Error . . . 3

**2 Novel Uses** **3**
2.1 Rule-based systems . . . 4
2.2 Spell Checkers . . . 4
2.3 Estimating Join Sizes . . . 5
2.4 Differential Files . . . 5
2.5 Network Applications . . . 6
2.6 Attenuated Bloom filters . . . 8
2.7 Text Analysis . . . 9

**3 Implementation** **11**
3.1 Hashing . . . 11
3.2 Basic Implementation . . . 11
3.2.1 Operations on Cells . . . 12
3.2.2 Operations on Bloom filters . . . 13
3.3 Compressed Bloom filters . . . 13

**4 Analysis** **14**
4.1 Time Complexity . . . 14
4.2 Relationship Between Parameters . . . 14
4.2.1 The General Case . . . 15
The Governing Equation . . . 15
Error Rate . . . 16
Rejection Time . . . 16
Growing Sets . . . 17
4.2.2 The Optimal Case . . . 17
Error Rate . . . 18
Rejection Time . . . 18
Governing Equation . . . 19
4.3 Performance issues . . . 19
4.4 Variation on Standard Bloom Filters . . . 20

**5 Summary** **21**
5.1 Optimal Filters . . . 21
5.2 Trade-offs in Filter Performance . . . 21

**References** **23**

**A Miscellaneous Methods** **27**

**B Derivation of Rejection Time Inequality** **28**

# 1 Introduction

The Bloom filter a way of using hash transforms to determine set membership [1]. Bloom filters find application wherever fast set membership tests on large data sets are required. Such applications include spell checking, differential file updating, distributed network caches, and textual analysis. It is a probabilistic method with a set error rate. Errors can only occur on the side of inclusion — a true member will never be reported as not belonging to a set, but some non-members may be reported as members.

We describe Bloom filters, elucidate some of their properties, and present a survey of their uses. The survey in this paper concentrated on research findings published between 1996 and 2002 and abstracted or cited in the following periodical indexes: 1) INSPEC[1], 2) NEC Research Institute ResearchIndex[2], and 3) ACM Digital Library[3].

The rest of the article is organized as follows: Section 2 outlines novel uses of Bloom filters. Section 3 describes the implementation of Bloom filters and the operations conducted. Section 4 is the analysis of Bloom filters and we present a summary in Section 5.

## 1.1 Definition

Bloom filters use hash transforms to compute a vector (the filter) that is representative of the data set. Membership is tested by comparing the results of hashing on the potential members to the vector. In its simplest form the vector is composed of $N$ elements, each a bit. An element is set if and only if some hash transform hashes to that location for some key. Figure 2 shows such a filter with $m = 4$ hash transforms and $N = 8$ bits.

[1] ⟨URL:http://www.iee.org/Publish/INSPEC/⟩
[2] ⟨URL:http://citeseer.nj.nec.com/⟩
[3] ⟨URL:http://portal.acm.org/⟩

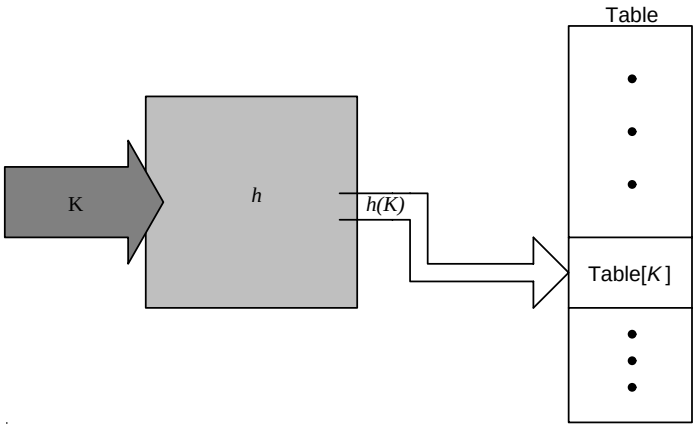



Figure 1: A typical hash transform in action

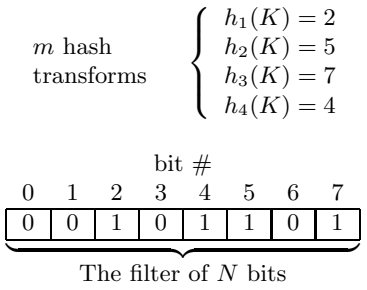



Figure 2: A simple Bloom filter

Bloom filters can be combined with other methods, such as signatures [1, 2]. Figure 3 depicts a case in which the filter contains references to information related to records rather than only the records. In that case the hash transforms will hash to $N/(b+1)$ cells, where $b$ is the size of the signature and the extra bit is used to flag cells containing signatures [1]. The analysis of such a filter is equivalent to that for the simple case, which this paper discusses.

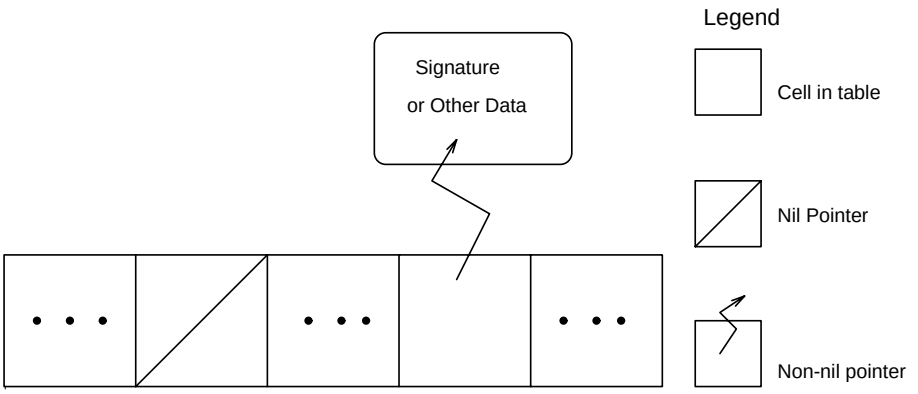



Figure 3: A Bloom filter with signature information

### 1.2 What Is An Error

Errors can occur when two or more transforms map to the same element. The membership test for a key $K$ works by checking the elements that would have been updated if the key had been inserted into the vector. If all the appropriate flag bits have been set by hashes then $k$ will be reported as a member of the set. If the elements have been updated by hashes on other keys — and not $K$ — then the membership test will incorrectly report $K$ as a member. For example, if the set *Vegetables* contains *potato* and *cabbage* but not *tomato*, then the Bloom filter illustrated in Figure 4 would incorrectly identify *tomato* as a *Vegetable*. Such an error could occur because all the bits that would be set if *tomato* were hashed on would already be set in the filter.

## 2 Novel Uses

This section reviews some of the most interesting applications of Bloom filters. It is perhaps surprising that what is essentially a set-membership test is of use in so many important applications.

A portion of the filter

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ··· | P | C | P | P | C | — | C | ··· |
| ··· | — | T | — | T | T | — | — | ··· |

Results of hashing *tomato*

| Key | Meaning |
|---|---|
| P | updated by *potato* |
| C | updated by *cabbage* |
| T | would be set by *tomato* |
| — | unset |

Figure 4: *tomato* is erroneously identified as a member

## 2.1 Rule-based systems

Burton H. Bloom originally proposed filter hashing as part of a program to automatically hyphenate words. He wanted to separate words that could be hyphenated by the application of simple rules from the minority that required extensive analysis. He proposed using his filter method to separate the 10% of difficult words from the rest [1].

## 2.2 Spell Checkers

Bloom filters have been successfully applied in spell checking programs such as `cspell` [3–5]. They are employed to determine if candidate words are members of the set of words in a dictionary. In the case of `cspell`, suggested corrections are generated by making all single substitutions in rejected words and then checking if the results are members of the set [3]. Bloom filters perform very well in such cases [3]. The filter size was chosen to be large enough to allow the inclusion of additional words added by the user.

## 2.3 Estimating Join Sizes

Mullin [6, 7] suggested using Bloom filters to estimate the size of joins in databases. This is of particular advantage in the case of distributed databases where communications costs are to be kept to a minimum. He presented a method by which filters that are too large to fit in memory can be used [7]. The method is essentially to use a representative sample of a filter for testing and ignore all hashes outside the range of the sample. Since hash transforms are pseudo-random, any significantly large portion of a filter can act as a sample.

## 2.4 Differential Files

A major area of interest in the application of Bloom filters has been their use in differential file access [4, 5, 8]. A differential file is essentially a separate file which contains records that are modified in the main file [4]. Differential files are used as caches in large databases: when a change is made to a record in the main database the differential file is updated; when all the changes have been made to the database then the differential file is used to update the database. When the differential file is much smaller than the database, changes to it can be made without the overhead needed to search the main file. Of course, it would be best to keep the entire set of records in memory at once, but this is not feasible for large data sets and so the probabilistic approach offered by Bloom filters is used. Bloom filters in core memory are used to predict if a record will be found in the differential file.

### Benefits and Drawbacks

A common assumption in such analyses is that if a record (e.g., a store's credit card account) has been updated recently then it is likely to be updated again soon [5]. If this assumption does not hold, then Bloom filters may not be a

suitable for this application. Another important consideration is how often the differential file should reset. The differential file must not be allowed to grow too large to fit in memory or much of the advantage is lost [5].

The benefits of using this technique includes improvements in performance, greater database reliability and reduced backup and recovery costs. However the technique is effective only within certain parameters. Implementors must decide on trade-offs between on the one hand the size of the filter and the number of hash transforms to use, and on the other hand the filter error rate. Those values will depend in turn on the given transaction volume, the number of keys to be accessed and updated, and the characteristics of the key set. We provide an analysis of those trade-offs in Section 4.

## 2.5 Network Applications

Bloom filters have recently found many applications in networks [9]. Networks rely on some form of routing to transfer messages between hosts. Routers are special-purpose network devices that must operate with high efficiency in real-time. Data can be delayed or lost when routers are overwhelmed with traffic. An interesting solution to the problem of enforcing fairness in routing is by Feng et al.'s Stochastic Fair Blue (SFB) algorithm [10]. When routers implementing SFB get near their capacity they begin to drop packets from the various hosts that are connecting to them. The routers use Bloom filters and labels to probabilistically determine which hosts continue to send more than their share of traffic even when some of their data are dropped by the router. Hosts which continue to operate in this non-cooperative fashion have more of their packets dropped. But traffic from hosts which reduce their demands on the router are not dropped. Bloom filters are used as a space- and time-efficient method to keep track of which hosts are sending too much traffic.

An important aspect of network applications is that they are not independent. The cost of transactions that rely on network traffic is generally higher than computations made in a single host. To minimize the time needed to compute and deliver correct results, distributed applications are designed to avoid using the network as much as possible and to send messages to the right host when they must rely on the network. This is a similar concept to the differential file paradigm we discussed in Section 2.4.

We give an example that is characteristic of the class. Bloom filters are used in caching proxy servers on the World Wide Web (WWW). The WWW can be viewed as a distributed system for delivering documents which are sent by servers to clients only upon request. Caching improves performance when clients obtain copies of files from neighbouring servers instead of from the originating server (which may be several slow network links away). Proxy servers intercept requests from clients and either fulfill the requests themselves or re-issue them to servers [11].

If the proxy can obtain a copy of the document from a cache (either its own or that of a nearby co-operating proxy) then the document is retrieved from the cache and a *cache hit* is registered. The *hit rate* of a cache the a measure of a cache's effectiveness. Proxies are typically deployed as hierarchies (which mimic logical network architectures) or as a series of co-operating proxies (without regard to network architecture) [11]. The performance of a Web cache scheme depends on the size of its client community; the bigger the user community, the higher the probability that a cached document will soon be requested again.

Bloom filters are used in Web caches to efficiently determine the existence of an object in a cache [10] and they can be used for Web cache sharing too. Web caches are shared to reduce message traffic. Caching proxies may be implemented so as not to transfer the exact content of their caches (i.e., lists of

URLs) but instead to broadcast much smaller Bloom filters that represent the contents of the cache. If a proxy wants to determine if another proxy has a page in its cache, it checks the appropriate Bloom filter.

Bloom filters are also used in cache digests. A *cache digest* is essentially a lossy compression of all cache keys with a lookup capability. Digests are made available via HTTP (the main network protocol of the WWW), and a cache downloads its neighbors digest at startup. By checking a neighbor's digest, a cache can determine with certainty if a neighbouring cache does not hold a given object. Their use in cache digest allows caches to efficiently inform each other about their contents without any per-request delays. The main goal is to reduce 'cache directory' size while keeping the number of collisions low. Bloom filters are an efficient way of serving those purposes. The small chance of a false positive is greatly outweighed by the significant reduction in network traffic achieved by using the succinct Bloom filter instead of sending the full list of cache contents [12].

## 2.6 Attenuated Bloom filters

The caches we have seen so far do not replicate data. Where cached data might be replicated, attenuated Bloom filters (ABFs) may be useful. Rhea and Kubiatowicz developed such 'a lossy distributed index' technique using Bloom filters for networks with nodes that communicate network topology with each other [13, p. 1248].

ABFs are composed of arrays of Bloom filters. Each node for the network stores an ABF for each outgoing link. An ABF is an array of $n$ Bloom filters which together represent the contents of the data cached at neighbouring nodes that can be reached within $n$ network links.

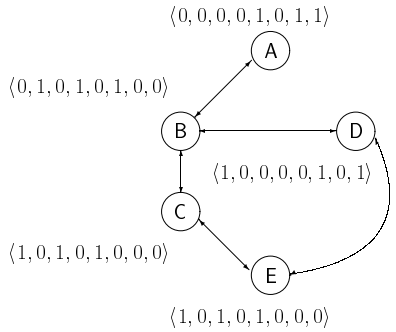
An example will make the application clearer. Consider the outgoing link

from node A to node B in the network depicted in Figure 5. The $n^{\text{th}}$ Bloom filter in the ABF from A to B is the union of all the Bloom filters in all nodes on a path of length $n$ beginning with B.

A basic Bloom filter could represent the probability that a specific datum is available from a node on a path beginning with B. Attenuated Bloom filters (ABFs) provide that information and also information about how many links away that datum is presumed to be. ABFs can be used to speed up searches in peer-to-peer networks since the searches resemble depth first traversals of the network as guided by the probabilistic information in the ABFs. Such searches are biased in favour of nodes that are most likely to contain the data and are closest to the current root node. The search algorithms apply penalties to Bloom filters that are along longer paths (because there is a greater cost associated with traversing more network links, and the longer the path the greater the possibility of searching many nodes). The exact penalties applied and details of the search algorithms are experimental. The methods are being used in conjunction with OceanStore [14], an extremely large global persistent data store.

## 2.7 Text Analysis

Bernstein [15, 16] produced an interactive program that uses Bloom filters to find related passages in a monograph. It works by constructing a Bloom filter of all the words in each passage of a monograph and then computing the normalized dot product of all pairs of them. The result of every dot product is a similarity measure — the higher the value the more likely the passages are to have related content. This can be a valuable tool for scholars, if as Bernstein claims it often finds connections that would otherwise go unnoticed [15]. Mylonas and Bernstein [17] have adapted it to work with Latin as well as English texts. They claim that this tool




(a) A network with Bloom filters in each node

| bit # | | | | | | | | nodes |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | B |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | C, D |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | E |

(b) The attenuated Bloom filter for A → B for the network in Figure 5(a)

Figure 5: Attenuated Bloom Filters

> . . . provide[s] impressionistic information that can open the text in new and valuable directions under the reader's guidance. . . . [It is] inexact and error-prone, seeking to provide intriguing suggestions for the scholar's consideration rather than objective data. [17, p. 182]

Bloom filters are used in many areas including updating databases, estimating the size of database joins, aiding scholarly research, and in spell checking programs.

# 3 Implementation

Having now shown some of what Bloom filters can do and where they are used we will examine how they operate.

## 3.1 Hashing

Hashing transforms are typically pseudo-random mathematical transforms used to compute addresses for lookup [18, 19]. Figure 1 shows the use of the hash transform $h$, to find an item with a key $K$, stored at address $h(K)$. The time complexity of searches by hashing can be as low as $O(1)$ or as high as $O(N)$, for a hash table with $N$ elements. The worst-case behaviour occurs when two or more distinct keys $K_i \neq K_j$ collide, i.e., $h(K_i) = h(K_j)$, and the entire table must be searched to find the correct entry [18, pp. 507 – 508]. Bloom filters are a fast method in which the hash transforms always have constant time complexity — there is no attempt at collision resolution. Knuth [18] described Bloom filters as a type of superimposed coding because all of the hash transforms map to the same table.

## 3.2 Basic Implementation

Bloom filters have three operations: A membership test (Procedure 1), Initialization (Procedure 5), and Update (Procedure 6). Procedures 2 – 6 are listed in Appendix A. INITIALIZE clears all the elements in the vector. INSERT computes the values of the $m$ hash transforms for a key and updates the appropriate elements. In the simplest case the update sets the element's flag bit. It requires time proportional to the number of hash transforms. In more complicated cases, additional information would also be placed in the element. The time complexities are summarized in Table 1.

In the example shown in Figure 2, hash transform $h_1$ updates the value of

| Procedure | Parameters | Time complexity |
|---|---|---|
| INITIALIZE | Table of $N$ cells | $O(N)$ |
| SET | Cell in Table | $O(1)$ |
| CLEAR | Cell in Table | $O(1)$ |
| ISSET | Cell in Table | $O(1)$ |
| INSERT | Table, Key, and $m$ hash transforms | $O(m)$ |
| ISMEMBER | Table, Key, and $m$ hash transforms | $O(m)$ |

Table 1: Time Complexities of Filter Steps

element 2 for key $K$ and transform $h_2$ updates the value of element 5 using the same key. ISMEMBER computes the same hash values as INSERT but instead of updating the elements it checks if they have been set. By definition, only members have their keys inserted into the vector. If any of the hash transforms, $h_i(K)$, compute a vector element that has not been set, then the key $K$ could not have been inserted into the vector and therefore cannot be a member of the set. Note that the worst time complexity for ISMEMBER occurs for members (and non-members that are erroneously reported as members). As we show in the sections named 'Rejection Time' below, the complexity can be considerably less for non-members.

### 3.2.1 Operations on Cells

For the purposes of the analysis, we are presenting only the essentials of Bloom filters — the algorithms are for single bit elements. The analysis of filters with more complicated cells is essentially the same as for the simple case [1].

Blustein has shown how to efficiently implement these operations using C [20]. ISMEMBER is presented immediately below. The other algorithms are presented in Appendix A. Their essential characteristics are presented in Table 1.

### 3.2.2 Operations on Bloom filters

**Procedure 1 (IsMember)**

IsMember($Table$,$Key$)→ **Boolean**

*1.* $i \leftarrow 0$

*2.* **repeat**

*3.* $i \leftarrow i + 1$

▷ $h_i$ is the $i^{\text{th}}$ hash transform, where $1 < i \leq m$

*4.* **until** $((i = m) \vee \neg(\text{IsSet}(Table[h_i(Key)])))$

*5.* **if** $i = m$ **then**

*6.* **return**(IsSet($Table[h_i(Key)]$))

*7.* **else**

*8.* **return**(**False**)

**end.**

## 3.3 Compressed Bloom filters

Space efficiency is particularly important for applications, such as distributed caches, that send Bloom filters as messages over networks. Mitzenmacher [21] proposed a method based on information entropy measures for compressing such filters to improve transmission rates at the cost of more computing time. Interestingly he found that 'the number of hash functions that minimizes the false positive [error] rate without compression in fact maximizes the false positive [error] rate with compression.' [21, p. 146] The method has not yet been implemented or tested.

# 4 Analysis

We now analyse the performance of Bloom filters. First we show the worst-case times for the algorithms, then we determine the various trade-offs that are necessary in any practical implementation.

## 4.1 Time Complexity

INITIALIZE The naïve implementation requires $O(N)$ time, however if $N$ is the size of a native data type then it can be done in constant time.

INSERT Insertion requires the computation of $m$ hash transforms, each of which requires $O(1)$ time. (Since collision detection is not necessary all the hash transforms have to do is compute values.) INSERT therefore takes $O(m)$ time per key or $O(mk)$ for all $k$ keys.

Note that the filter, or substantial portions of it, can be computed in advance. In the case of a spell checking program for instance, a filter of the standard dictionary can be built prior to running the program. If the program allows a user to add words to the dictionary on-the-fly, then keys based on those words would need to be inserted at run-time.

ISMEMBER The loop in Procedure 1 may require the computation of as many as $m$ transforms. Below we prove that, in the optimal case, on average only two transforms will be required to reject any non-member. In the worst case, when the key is a member of the set, the time complexity is $O(m)$.

## 4.2 Relationship Between Parameters

The behaviour of Bloom filters is determined by four parameters:

$N$ The number of elements (or cells) in the filter

$m$ The number of hash transforms to be used

$k$ The number of set members

$f$ The fraction of elements (or cells) that are set in the filter

Here we derive equations that describe the relationship between these factors for the general and optimal cases. The general case is applicable to growing and static sets but the optimal case occurs when the error rate is minimal. As we show in Section 4.2.2 optimal performance is predicted for only static sets in which half of the elements have been updated.

The *governing equation* provides a way to predict the amount of space a particular filter will require. The expected fraction of false positive results given the parameter values is the *error rate*. The *rejection time* is the expected number of hashes that will be required to determine that a key is not a member of the set. These analytic results are summarized in Table 2 (on page 19).

### 4.2.1 The General Case

Since hash transforms are pseudo-random, the probability of a particular filter element being addressed by a hash transform is $1/N$. Therefore the probability of a particular element not being updated is $1 - 1/N$. If we assume that the keys are randomly distributed then the probability of a particular element not being updated after after all $k$ keys have been hashed is $(1 - 1/N)^k$.

**The Governing Equation** The probability that a particular element not being updated by any of the $m$ transforms, after all $k$ keys have been entered is $P_{unset}$.

$$P_{unset} = (1 - 1/N)^{mk} \tag{1}$$

Equation 1 is based on the assumption that the hashes are equally likely to set any bits.

The probability that an element is set is $P_{set}$.

$$P_{set} = 1 - P_{unset}$$

**Error Rate** The analytic probability that an element is hashed to by all $m$ hash transforms is $P_{allset}$.

$$\begin{aligned} P_{allset} &= (P_{set})^m \\ &= \left(1 - (1 - 1/N)^{mk}\right)^m \end{aligned} \tag{2}$$

Both of these computations are based on the standard assumption that the hash transforms are independent.

**Rejection Time** If $f$ is the fraction of the bits that are set in a Bloom filter then a single hash has a probability $1 - f$ of not rejecting a non-member. Assuming that the hash transforms are independent, the $h^{\text{th}}$ hash also has a probability $1 - f$ of not rejecting the non-member. In general then the probability of $h$ hashes being required to reject a non-member is $\sum_{h=1}^{m} h \times f^{h-1} \times (1 - f)$. We can simplify the sum by recognizing that $\sum_{h=1}^{m} h \times f^{h-1}$ as the integral of the sum of the (finite) geometric series with $a_0 = 1$ [22]. A detailed derivation is in Appendix B. If $m$ is infinite then the sum converges when $|f| < 1$. Clearly $0 \leq f < 1$, since a Bloom filter with all of its bits set cannot be used to detect non-members and $f$ will be zero only if no keys have been hashed. Thus Equation 3 represents the relationship between the predicted number of hashes needed to reject a non-member and the number of elements set in the filter.

$$\sum_{h=1}^{m} h \times f^{h-1} \times (1-f) \quad \leq \quad \frac{1}{1-f} \tag{3}$$

Note that although non-members can be rejected with fewer than $m$ hashes, member keys will require all of the hashes to verify their status.

**Growing Sets** In applications where the membership set is allowed to grow, e.g., in spell checkers with user dictionaries, the number of keys should be the total number of keys expected. For example, if a spell checker is constructed with an initial dictionary of 30 000 words and it is predicted that another 5000 will be added as the applications runs then the value of $k$ should be 35 000.

### 4.2.2 The Optimal Case

Analysis of the optimal case is based on the standard assumption of parallel hash functions each of which covers half of the $N$ elements in the table. The optimal case is the one where the error rate is minimized for a given filter size $N$, i.e. $\frac{dP_{allset}}{dm} = 0$.

It follows immediately that [23]:

$$\begin{aligned}
\frac{d\,P_{allset}}{dm} &= \frac{d}{dm} e^{m \ln(1-(1-1/N)^{mk})} \\
&= ((\ln(1-(1-1/N)^{mk}) + \frac{m}{1-(1-1/N)^{mk}} \\
&\quad \times (-\frac{d}{dm}(1-1/N)^{mk}))) \times e^{m \ln(1-(1-1/N)^{mk})}
\end{aligned}$$

where

$$\begin{aligned}
\frac{d}{dm}(1-1/N)^{mk} &= \frac{d}{dm} e^{mk \ln(1-1/N)} \\
&= k \ln(1-1/N) \times e^{mk \ln(1-1/N)}
\end{aligned}$$

so that

$$\begin{aligned}\frac{d\,P_{allset}}{dm} &= \left( \begin{array}{l} \ln(1-(1-1/N)^{mk}) \quad - \\ \frac{mk\ln(1-1/N)\times(1-1/N)^{mk}}{1-(1-1/N)^{mk}} \end{array} \right) \times (1-(1-1/N)^{mk})^m \\ &= 0\end{aligned}$$

By dividing by Equation 2, which cannot be zero (unless no keys have been hashed), and moving the coefficient we obtain

$$\begin{aligned}&\ln(1-(1-1/N)^{mk}) \\ = \quad &\frac{(\ln(1-1/N)^{mk})\times((1-1/N)^{mk})}{1-(1-1/N)^{mk}}\end{aligned}$$

By substituting $x$ for $(1-1/N)^{mk}$ and multiplying both sides by $(1-x)$ we obtain $(1-x)\ln(1-x) = x\ln x$, which is the same as $(1-x)^{(1-x)} = x^x$. Therefore $x = 1-x$, so the error rate will be minimal when $(1-1/N)^{mk} = 1/2$.

Therefore optimal filters have half their flag bits set; i.e., the set is composed of $N/2$ elements. In an optimal filter $P_{set} = P_{unset} = 1/2$.

**Error Rate** Combining this with Equation 2 we find that the error rate, for optimal Bloom filters, is $P_{opt}$.

$$P_{opt} = \left(\frac{1}{2}\right)^m \tag{4}$$

**Rejection Time** In the optimal case a good hash transform will exclude half the elements of the vector for any key. One hash will eliminate half the elements; all subsequent hashes will eliminate half the remaining elements. If one hash eliminates half of the elements then two hashes will eliminate $\frac{1}{2} + \frac{1}{4}$ elements, etc. Since the pattern is a geometric series which converges to 2 as the number of hashes grows, on average only 2 hashes are needed to reject candidates [3]. This prediction is consistent with the general case shown in Equation 3.

**Governing Equation** The general governing equation (Equation 1, above) is $P_{unset} = (1 - 1/N)^{mk}$. In the optimal case this becomes $1/2 = (1 - 1/N)^{mk}$. From this is trivial to derive Equation 5.

$$-\ln 2 = mk \ln (1 - 1/N) \tag{5}$$

The Taylor expansion of $\ln(1 + x)$ is

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} \cdots \tag{6}$$

In the case of Equation 5, $x = -1/N$. For $N \gg 1$, $x^2 \approx 0$ and $\ln(1 - 1/N) \approx -1/N$. It follows that, in the optimal case, described by Equation 7, the relationship holds.

$$\begin{aligned} N &\approx mk/\ln 2 \\ &\approx mk/0.69 \end{aligned} \tag{7}$$

The analytic results are summarized in Table 2.

| | General Case | Optimal Case* |
|---|---|---|
| Governing equation | $P_{unset} = (1 - 1/N)^{mk}$ | $P_{unset} = 1/2$<br>$N \approx mk/\ln 2$ |
| False positive rate | $\left(1 - (1 - 1/N)^{mk}\right)^m$ | $2^{-m}$ |
| Rejection time | $\leq 1/(1 - f)$ | 2 |

Table 2: Summary of Analytic Results

*The optimal case holds for static sets with $f = 1/2$.

## 4.3 Performance issues

Of course, the choice of hash transforms will have a major impact on the performance of the Bloom filter. To be useful Bloom filters require hashing transforms

that will not hash to the same set of addresses. Gremillion [8] and Mullin [5] found that, when applied to differential files, the error rate was much higher than the analysis predicted.

To remedy that deficit Ramakrishna [4, 24] developed so-called perfect hash transforms. *Perfect hash transforms* are a class of hash function that completely avoid collisions for the specific key set for which they were generated.

His tests were simulations of Bloom filters on differential files of user IDs, the Unix™ word list (`/usr/dict/words`), and library call numbers. Simulations studies of Bloom filters are accepted in the literature as an accurate method for determining test performance [5]. Ramakrishna reported that all the results were similar and gave details for the file of user IDs. The results were all within a standard deviation of what the governing equation predicts. Czech et al. [25] have since devised a fast algorithm generate minimal perfect hash transforms.

Perfect hash transforms can be used only when the entire membership set is known a priori. They are therefore not suitable for applications that use growing sets. For instance, perfect hash transforms are suitable for use with differential files of bank accounts because all of the possible account numbers are known in advance. However a spell checker that allowed users to include arbitrary words could not expect optimal results from a Bloom filter built with perfect hash transforms for the words in its list of correct spellings.

## 4.4 Variation on Standard Bloom Filters

If the membership set is known in advance then better performance than with a standard Bloom filter can be achieved using related techniques. For instance we can establish an arbitrarily low false positive rate by using perfect hash functions and limiting the size of the shared hash table. The important step is to order the elements by their discriminating power and eliminate those with

the lowest power until the desired tradeoff between the size of the table and the predicted false positive rate is reached or exceeded.

# 5 Summary

This paper has described Bloom filters, some of their applications, and provided an analysis of the general and optimal performance cases. Bloom filters used purely for probabilistic membership tests accurately identify non-members.

## 5.1 Optimal Filters

In the optimal case, non-members are detected within two hashes. The optimal case occurs only to sets in which all of the possible keys are known in advance and in which half of the elements are set. In practice the optimal case requires the use of perfect hash transforms (as described in Section 4.3).

## 5.2 Trade-offs in Filter Performance

The error rate can be decreased by increasing the number of hash transforms and the space allocated to store the table. The analytic performance of Bloom filters for growing and static sets is given in Equation 3. Formulae that can be used to tune filters with respect to error rate, filter size, number of keys, and hash function are summarized in Table 2.

Bloom filters should be considered for programs where an imperfect set membership test could be helpfully applied to a large data set of unknown composition. Such programs include spell checkers and those that use data stored in differential files. The great advantage of Bloom filters over the use of single hash transforms is their speed and set error rate. Although the method can be applied to sets of any size, small sets are better dealt with by trees and heaps

that can determine for certain if a key belongs to a set. Other methods are generally more accurate for sets whose composition is known in advance but they require more space than Bloom Filters.

**Acknowledgments** This work was improved by Jim Mullin's comments on an earlier draft and Ray Spiteri's help with the derivation of Equation 3. Conversations with Jason Rouse were most helpful.

# References

[1] Burton H. Bloom. Space/time trade-offs in hashing coding with allowable errors. *Communications of the ACM*, 13(7):422 – 426, July 1970. URL ⟨`http://doi.acm.org/10.1145/362686.362692`⟩.

[2] Christos Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):49 – 74, March 1985. URL ⟨`http://doi.acm.org/10.1145/4078.4080`⟩.

[3] James K. Mullin and Daniel J. Margoliash. A tale of three spelling checkers. *Software — Practice and Experience*, 20(6):625 – 630, June 1990.

[4] M. V. Ramakrishna. Practical performance of Bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237 – 1239, October 1989. URL ⟨`http://doi.acm.org/10.1145/67933.67941`⟩.

[5] James K. Mullin. A second look at Bloom filters. *Communications of the ACM*, 26(8):570 – 571, August 1983. URL ⟨`http://doi.acm.org/10.1145/358161.358167`⟩.

[6] James K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16(5):558 – 560, May 1990. URL ⟨`http://ieeexplore.ieee.org/iel1/32/1900/00052778.pdf`⟩.

[7] James K. Mullin. Estimating the size of a relational join. *Information Systems*, 18(3):189 – 196, 1993. ISSN 0306-4379.

[8] L. L. Gremilion. Designing a Bloom filter for differential file access. *Communications of the ACM*, 25(9):600 – 604, September 1982. URL ⟨`http://doi.acm.org/10.1145/358628.358632`⟩.

[9] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. URL ⟨`http://www.eecs.harvard.edu/~michaelm/`

NEWWORK/postscripts/BloomFilterSurvey.pdf⟩. Submitted to Annual Allerton Conference on Communication, Control, and Computing, 2002.

[10] Wu-Chang Feng, Dilip D. Kandlur, Debanjan Saha, and Kang G. Shin. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *INFOCOM 2001: Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1520 – 1529. IEEE, 2001. URL ⟨http://ieeexplore.ieee.org/iel5/7321/19795/00916648.pdf⟩.

[11] Jia Wang. A survey of web caching schemes for the internet. *ACM SIGCOMM Computer Communication Review*, 29(5):36 – 39, October 1999. ISSN 0146-4833. URL ⟨http://doi.acm.org/10.1145/505696.505701⟩.

[12] Alex Rousskov and Duane Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22 – 23):2155 – 2168, April 1998. URL ⟨http://www.sciencedirect.com/science/article/B6TYT-3VY4SS7-B/1/868712711e204b138cea744eaf0d39a8⟩.

[13] Sean C. Rhea and John Kubiatowicz. Probabilistic location and routing. In *INFOCOM 2002: Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1248 – 1257. IEEE, 23 – 27 June 2002. URL ⟨http://ieeexplore.ieee.org/iel5/7943/21923/01019375.pdf⟩.

[14] Kris Hildrum. The OceanStore project: Project overview. [Webpage], 2002. URL ⟨http://oceanstore.cs.berkeley.edu/info/overview.html⟩. Last modified on 07/08/2002.

[15] Mark Bernstein. An apprentice that discovers hypertext links. In N. Streitz, A. Rizk, and J. André, editors, *Hypertext: Concepts, Systems and Appli-*

*cations*, The Cambridge Series on Electronic Publishing, pages 212 – 223. INRIA, France, Cambridge University Press, 1990.

[16] Mark Bernstein, Jay David Bolter, Michael Joyce, and Elli Mylonas. Architectures for volatile hypertext. In *Hypertext '91 Proceedings*, pages 243 – 260, San Antonio, Texas, December 1991. URL ⟨`http://doi.acm.org/10.1145/122974.122999`⟩.

[17] Elli Mylonas and Mark Bernstein. A literary apprentice. In *The 19th International Conference of the Association for Literary and Linguistic Computing; and the 12th International Conference on Computers and Humanities*, pages 181 – 186, Christ Church, Oxford, April 1992.

[18] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, 1973.

[19] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics A Foundation for Computer Science*, chapter 8.5 Hashing, pages 397 – 412. Addison-Wesley Publishing Company, 1989. ISBN 0-201-14236-8.

[20] James Blustein. Implementing bit vectors in C. *Dr. Dobb's Journal*, 20 (233), August 1995. Updated code is available from URL ⟨`http://www.csd.uwo.ca/~jamie/BitVectors/`⟩.

[21] Michael Mitzenmacher. Compressed Bloom filters. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, pages 144 – 150, Newport, RI, USA, 2001. ACM SIGACT, ACM SIGOPS. ISBN 1-58113-383-9. URL ⟨`http://portal.acm.org/citation.cfm?id=384004`⟩.

[22] Thomas Chan, Dan Margoliash, and Howard Shidlowsky. A word legality

module using a Bloom filter and suffix simulation. Unpublished work obtained from James K. Mullin (Dept. of Computer Science, Univ. of Western Ontario, London, Ontario N6A 5B7), 1983.

[23] Daniel J. Margoliash. CSpell — A Bloom filter-based spelling correction program. Master's thesis, University of Western Ontario, 1987.

[24] M. V. Ramakrishna. Perfect hashing for external files. Technical Report CS-86-25, University of Waterloo Computer Science Department, June 1986.

[25] Zbigniew J. Czech, George Havas, and Bohda S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257 – 264, October 1992. URL ⟨`http://www.sciencedirect.com/science/article/B6V0F-45FKW7V-68/1/95400691dc2f8738310f31bc965ca964`⟩.

# A Miscellaneous Methods

**Procedure 2 (Set)**

SET(*cell*)

▷ essentially a logical **or**

*cell* ← 1

**end.**

**Procedure 3 (Clear)**

CLEAR(*cell*)

*cell* ← 0

**end.**

**Procedure 4 (IsSet)**

ISSET(*cell*)→**Boolean**

**return**(*cell* = 1)

**end.**

**Procedure 5 (Initialize)**

INITIALIZE($Table$)

1. **for** $i \leftarrow 1 \ldots i = N$ **do**
2. CLEAR($Table[i]$)
3. **endfor**

**end.**

**Procedure 6 (Insert)**

INSERT($Table$,$Key$)

*1.* **for** $i \leftarrow 1 \ldots i = m$ **do**

*2.* ▷ $h_i$ is the $i^{\text{th}}$ hash transform

*3.* SET($Table[h_i(Key)]$)

*4.* **endfor**

**end.**

# B Derivation of Rejection Time Inequality

$$
\begin{aligned}
\sum_{h=1}^{m} h \times f^{h-1} \times (1-f) &= (1-f)\sum_{h=1}^{m} h \times f^{h-1} \\
&\leq (1-f)\sum_{h=1}^{\infty} h \times f^{h-1} \\
&= (1-f)\sum_{h=1}^{\infty} \frac{d}{df} f^h \\
&= (1-f)\frac{d}{df}\sum_{h=1}^{\infty} f^h, \text{ where } |f| < 1 \\
&= (1-f)\frac{d}{df}\frac{1}{1-f}, \text{ where } |f| < 1 \\
&= (1-f)\frac{1}{(1-f)^2}, \text{ where } |f| < 1 \\
&= \frac{1}{1-f}, \text{ where } |f| < 1
\end{aligned}
$$

Note that $\sum_{h=1}^{\infty} f^h$ is the sum of a geometric series.