

# Minimizing Page Faults on Bloom Filters

Adam Zawierucha (adz2)

Rice University  
zawie@rice.edu

## Abstract

Bloom filters [1] are used ubiquitously due to their speed and memory efficiency in theory and in practice. However, the standard implementation of sufficiently large bloom filters suffers from page faults. In this paper we propose a bloom filter implementation that guarantees one page access per operation. This minimizes page faults, thereby drastically improving efficiency. We will show theoretically and empirically that our hierarchical implementation is expected to be faster than the standard implementation without altering the false positive rate.

**Keywords:** probabilistic data structures, bloom filters, memory hierarchy, implementation, page fault analysis

## 1 Introduction

Bloom filters are extremely space efficient probabilistic data structures developed to implement set operations developed by Bloom in 1970 [1]. When an element is queried for membership, a “false” result means the element has definitely not been added, but a “true” only means the element *may* have been inserted. The rate at which the data structure incorrectly reports that an element has been inserted is called the *False Positive Rate (FPR)*. This probabilistic feature allows Bloom filters to be extraordinarily space efficient, only needing a constant number of bits per stored element (irrespective of the elements size).

Due to Bloom filter’s space efficiency they have been adopted in many domains from network security [2] to bioinformatics [4]. Any improvement in performance (without negatively altering the false positive rate) could lead to reduce latency in network calls or speed-ups in sequencing DNA. This motivates our proposed solution which exploits computer systems’ memory hierarchy to provide better improvement without increasing the false positive rate of bloom filters.

## 2 Implementations

### 2.1 Standard Implementation

Before we discuss the proposed solution, let us highlight the weakness of the standard implementation. The standard implementation allocates a bit vector of size  $m$  bits. Typically, this is set to be  $\times 10$  the expected number of elements it will hold.

*Standard Bloom Filter*

Bit Vector: 

--	--	--	--	--	--	--	--	--	--	--

 ...

Per the standard implementation, to insert an element we hash the element  $k$  times and set the corresponding bit in the vector. To query, we simply read instead of set the bit. Notice, that if  $m$  is sufficiently large, it will span accross multiple pages of memory. Thus, when we read and write to the underlying bit vector, we may page fault for every bit set, slowing down our insertions or queries.

### 2.2 Proposed Hierarchical Implementation

Our proposal is to allocate  $w$  bit vectors of size  $P$  bits, where  $P$  is computer system’s page size in bits and  $w$  is an integer such that  $m = wP$ . Notice, our proposed implementation uses the same amount of memory, but splits the bit vector into page size chunks.

Bit Vector 1: 

--	--	--	--

 Bit Vector 2: 

--	--	--	--

 ... Bit Vector  $w$ : 

--	--	--	--

To insert, hash the element  $l$  times mod  $w$  and insert the element per the standard bloom filter operations to the corresponding bit vector. Each bit vector (bloom filter) has  $k$  hash functions associated with it. Here  $l$  is a pre-determined parameter of the datastructure; we will discuss what the optimal setting is in a following section. To query, we simply query the corresponding bit vector instead of inserting. In essence, our proposal to create a bloom filter of bloom filters, hence the name.

Notice, for any given insertion or query, we have to perform  $l$  more hash operations than the standard implementation. This is not a concern if we select sufficiently cheap hash functions. More importantly, since each bit vector is on its own page of memory, we expect to page fault at most  $l$  times. Thus, if we minimize  $l$  without sacrificing effectiveness, we will reduce the expected number of page faults and increase the data structures efficiency.

### 3 Theoretical Work

In this section we will justify why the hierarchical implementation will outperform the standard implementation in theory while preserving effectiveness. First, we will **calculate expected number of page faults** per implementation. Second, we will **find the theoretical false positive rate**, which will guide us in our parameter selection for the implementation.

#### 3.1 Expected Page Faults

Page faults occur when the operating system must fetch memory from a source higher in the memory hierarchy to be used by the process. Whenever a page fault occurs, the program must be halted unnecessarily to resolve the page fault, which could require relatively slow I/O operations such as checking the TLB or loading the page from memory or disk. This leads to slower performance.

It is next to impossible to know when accessing a page will cause a page fault as this is highly dependant on the operating system. In general though, the more memory you are using, the higher the likelihood a pagefault will occur. In this analysis we will assume more page accesses is correlated with a higher likelihood of page faulting. We can formally compute the expected number of different pages that will be accessed (unlike whether or not it will page fault).

Note, when discussing page faults in this section, we will only discuss the page faults caused due to accessing the underlying bit vector of the bloom filter. Naturally, page faults can occur while running the underlying code of the bloom filter or running the hash functions, but this should be rare and would realistically only cause one page fault.

First, we will discuss the expected number of page faults for the standard implementation.

Let  $A$  be a random variable representing the number of pages accessed. Let  $P$  be the number of bits in a page and let  $m$  be the number of bits in the underlying bitvector. Suppose there are  $k$  hash functions. We now define the indicator variable  $A_i$  which is 1 if bit  $i$  is set, 0 otherwise.

$$A = A_1 + A_2 + \dots + A_{m/P}$$

Thus, the expected number of pages accessed can be found by computing the expected count that any page is accessed by the linearity of expectation:

$$E(A) = \sum_{i=1}^{m/P} E(A_i) = \frac{m}{P} \cdot E(A_i) \text{ for arbitrary } i$$

The probability that  $A_i$  is accessed at least once is the inverse of it being never accessed. Assuming that our hash function is uniform, we expect it to pick any particular page with probability  $\frac{1}{m/P} = \frac{P}{m}$ . Thus, the probability  $A_i$  is accessed at least once is:

$$E(A_i) = 1 - \left(1 - \frac{P}{m}\right)^k$$

Ergo, we have a closed form equation for the expected number of page faults for a given operation:

$$\text{Expected distinct pages accessed (Standard)} = \frac{m}{P} \left(1 - \left(1 - \frac{P}{m}\right)^k\right)$$

We will use this formula to compute the expected number of page faults for two reasonable cases. First, suppose you wanted a bloom filter to store 32,768 elements with an underlying bit vector of size  $\times 10$  that. Note, most computer systems have a page size of 4096 bytes, so this works out to require exactly 10 pages of memory. Additionally, suppose we pick the optimal bloom filter parameter and set  $k = 7$ . Under this scenario, we anticipate:

$$\text{Expected distinct pages accessed (Standard)} = 10(1 - 0.9^7) \approx 5.2$$

If we wanted to store 327,680 elements under a similar setting, then the number of page faults would be:

$$\text{Expected distinct pages accessed (Standard)} = 100(1 - 0.99^7) \approx 6.8$$

As we can see, even using a relatively small bloom filter, we anticipate almost every bit to be located in an entirely different page. This means we can page fault multiple times during a single operation!

Since our operation limits bit setting and reading to a single page per insertion or query, we need to access exactly one page!

$$\text{Distinct page accessed (Hierarchical)} = 1$$

Thus, our proposed hierarchical solution limits the number of page faults to at most one! Therefore, we anticipate much better performance.

## 3.2 False Positive Rate

### 3.2.1 Standard False Positive Rate

As we have covered in class, the false positive rate of a standard bloom filter is as follows.

$$(1 - (1 - \frac{1}{m})^{nk})^k \approx (1 - e)^{kn/m} \quad (1)$$

Setting  $k = 7$  will minimize the false positive rate, as discussed in class.

### 3.2.2 Hierarchical False Positive Rate

Suppose we have a allocated  $m$  bits in total chunked into bit vectors of size  $P$  and that we have to inserted  $n$  elements into our bloom filter.

We now want to compute the false positive rate of the hierarchical implementation. This can be computed by supposing the bloom filter has been filled with  $n$  elements and computing the probability that a querying a false key would result in a true response.

Assuming our hash function is uniform, any particular sub-bloom filter is anticipated to have  $\frac{nl}{m/P} = \frac{nlP}{m}$  elements in it. There are  $m/P$  bloom filters and we choose  $l$  of them to check. As before  $k$  is the number of hash functions a particular bloom filter uses. Thus, the chance any one bloom filter is set is:

$$(1 - (1 - \frac{1}{P})^{lknP/m})^k$$

For our hierarchical implementation to return true, all  $l$  bloom filters selected must return a positive result. Thus, the false positive rate is:

$$((1 - \frac{1}{P})^{lknP/m})^{kl}$$

We can use a well known identity for  $e^{-1}$  to approximate this false positive rate:

$$\approx (1 - e^{-lkn/m})^{lk}$$

Interestingly, this removes the dependence on the page size. More importantly, this formula becomes isomorphic to the false positive rate for the standard implementation. If we let  $k' := lk$  we see that our false positive rate for the hierarchical implementation is:

$$\approx (1 - e^{-k'/m})^{k'}$$

Therefore, as discussed for the standard implementation, the optimal selection for  $k'$  is 7. Therefore, either  $l = 1$  and  $k = 7$  or  $l = 7$  and  $k = 1$ . Since our goal is reduce the number of pages accessed, the former is a more sensible choice.

Therefore, the best parameter selection for our hierarchical implementation would be to select 1 bloom filter the size of a bloom filter and set 7 bits in each one.

$$l = 1 ; k = 7$$

## 4 Experiments

We will now empirically validate that the hierarchical implementation is more time efficient than the standard implementation without sacrificing accuracy. Two experiments will be conducted. First, we will **measure elapsed time as insertions scale** of each implementation. We anticipate that the hierarchical implementation will take less time than the standard implementation for any sufficiently large value of insertions. Second, we will **measure false positives as we scale bits allocated per element** of each implementation. We anticipate that the hierarchical and standard implementation will be approximately same as the theoretical false positive rate discussed before.

For both of these sections, we pseudorandomly generate keys to both insert and query. Discussion on exactly how these keys are generated is outlined in the appendix.

### 4.1 Comparing Time Efficiency

For this experiment, we seek to validate that the hierarchical implementation performs better than the standard implementation as the number of insertions grow. *Hypothesis:* The hierarchical bloom filter will take less time than the standard implementation to insert  $n$  keys for any  $n$ .

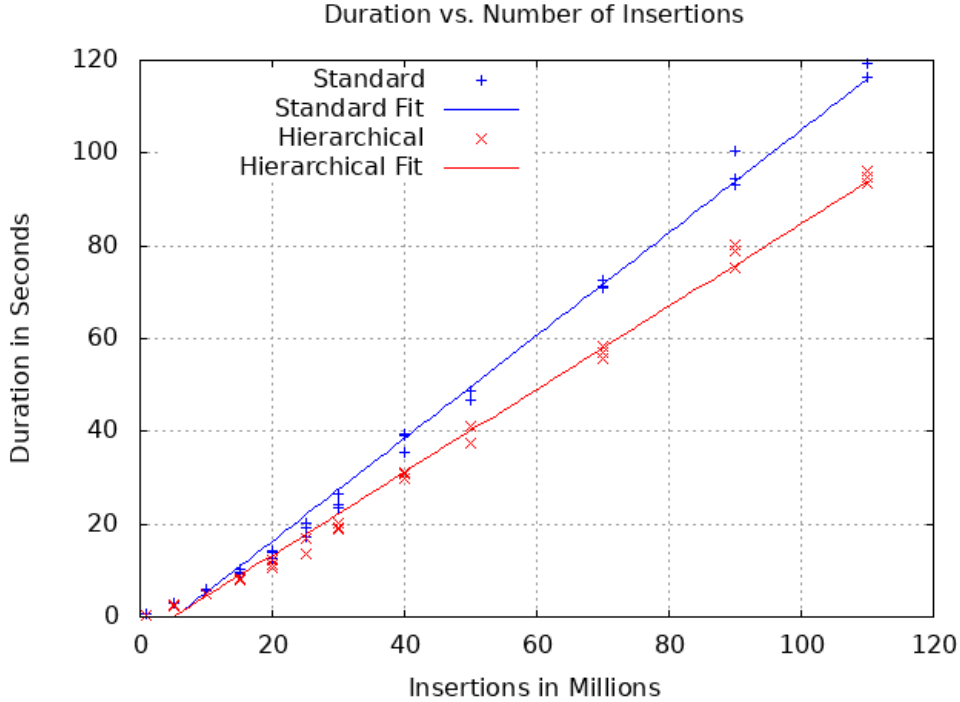
#### 4.1.1 Experimental Settings

The experiment will run as follows. For each implementation run the following procedure:

1. Generate  $n$  random keys.
2. Generate a bloom filter of both variants of size  $10n$ . Use the optimal theoretical configuration for each bloom filter (i.e,  $k = 7$ ,  $l = 1$ ).
3. Time how long it takes to insert all  $n$  keys into each of the bloom filters. Report this number.
4. Repeat for various sizes of  $n$ .

Repeat this entire process 3 times.

#### 4.1.2 Results



The slope of the line of best fits that are plotted are as follows where  $n$  is millions of insertions:

- The best fit line for the standard implementation is:  $t = (1.1078 \pm 0.01446) \cdot n - (5.78611 \pm 0.7407)$

- The best fit line for the hierarchical implementation is:  $t = (0.893221 \pm 0.01166) \cdot n - (4.52759 \pm 0.597)$

We will disregard the constant as we care about how these data structures perform as input volume scales. We can compute the efficiency difference by dividing the slopes:

$$\text{Hierarchical Implementation Slope} / \text{Standard Implementation Slope} = \text{Efficiency Difference}$$

$$(0.893221 \pm 0.01166 \text{ seconds/operation}) / (1.1078 \pm 0.01446 \text{ seconds/operation}) = 0.806301679 \approx 80\%$$

In other words, our implementation takes 80% less time per operation to complete. Thus, for any time frame, if the standard implementation performs one operation, the hierarchical implementation is expected to perform  $1/80\% = 1.25$  operations. Thus, our experiment shows that the hierarchical implementation performs 25% more operations per second than the standard implementation!

## 4.2 Comparing False Positive Rate

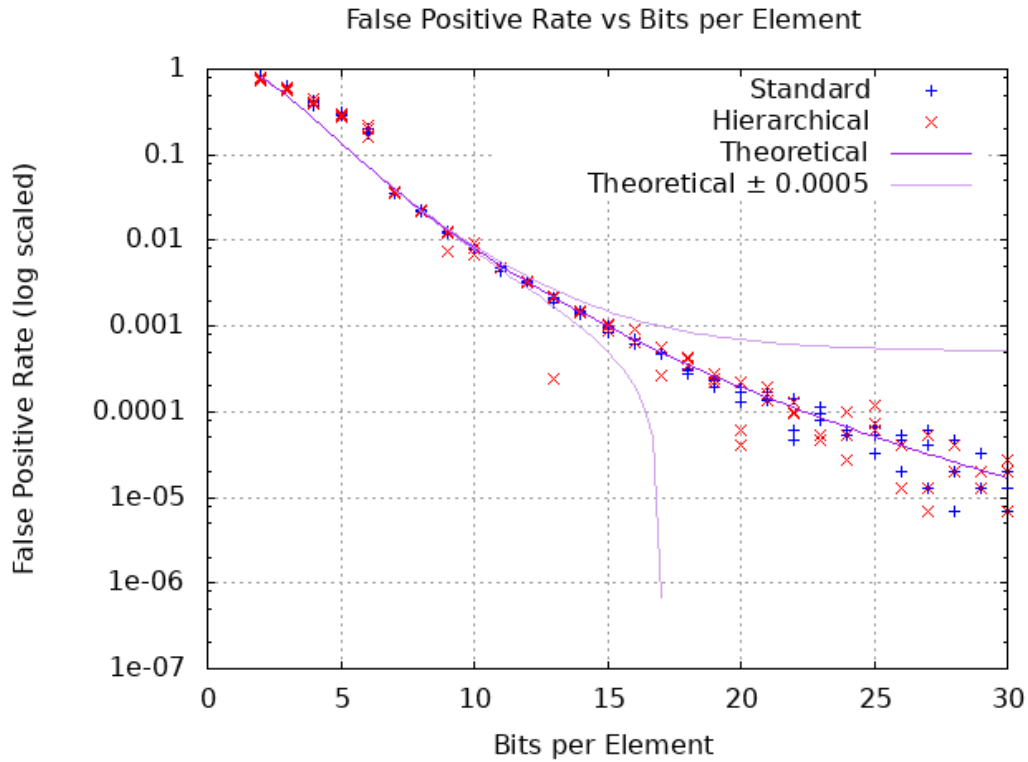
For this experiment, we seek to validate that the hierarchical implementation does not have a worse false positive rate than the standard implementation. *Hypothesis:* We expect them to have approximately the same false positive rate as the theoretical expectation discussed earlier.

### 4.2.1 Experimental Settings

The experiment will run as follows.

1. Generate 150,000 random “insertion” keys (of length 16).
2. Generate 150,000 random “false” keys to query distinct from the insertion keys (of length 15).
3. For each implementation run the following procedure:
  - (a) Let  $BPE$  be the bits per element (e.g  $BPE = 1$  or  $BPE = 10$ ).
  - (b) Generate a bloom filter of size  $150,000 \cdot BPE$ .
  - (c) Insert all the “insertion” keys and query all the “false” keys and measure how many of them the bloom filter return. Report this number.
  - (d) Repeat for various values of  $BPE$ .
4. Repeat this procedure again 3 times with different insertion and false keys.

### 4.2.2 Results



As we can see, both the standard and hierarchical implementation closely match the theoretical expectation. Both implementations do worse than theoretically expected if bloom filters are overpacked, but after Bits per Elements is greater than 6, both implementations are very close to theoretical expectation ( $\pm 0.0005$ ) or better.

### 4.3 Conclusion

Our experiments have supported both of our theoretically justified hypotheses. We have demonstrated that the hierarchical implementation is more efficient than the standard implementation; it can perform 25% more operations per second! Additionally, we have verified our that our implementation is just as effective as the standard implementation.

## 5 Literature Survey

Felix Putze et al. developed bloom filters with better cache efficiency and requiring less hash bits than the standard bloom filter [3].

## 6 Conclusion

## References

- [1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.
- [2] Shahabeddin Geravand and Mahmood Ahmadi. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks*, 57(18):4047–4064, 2013. ISSN 1389-1286. doi:<https://doi.org/10.1016/j.comnet.2013.09.003>. URL <https://www.sciencedirect.com/science/article/pii/S1389128613003083>.
- [3] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash- and space-efficient bloom filters. In Camil Demetrescu, editor, *Experimental Algorithms*, pages 108–121, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. isbn:978-3-540-72845-0. URL [https://link.springer.com/content/pdf/10.1007/978-3-540-72845-0\\_9.pdf](https://link.springer.com/content/pdf/10.1007/978-3-540-72845-0_9.pdf).
- [4] Allander T Andersson B Arvestad L Lundeberg J. Stranneheim H, Käller M.