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 insert or query. This minimizes page faults, thereby drasticaly improving efficiency. We will show theoretically and empirically that our hierarchical implementation is expected to be faster than the standard implementation.
Keywords: probabilistic data structures, bloom filters, memory hierarchy, implementation, page fault analysis

1 Introduction

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. The figure below represents the bit vector of length m.

Standard Bloom Filter

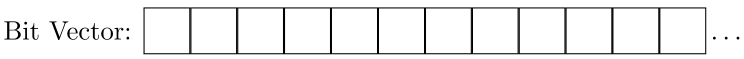


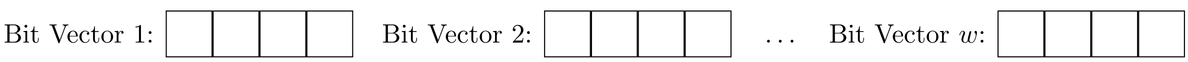
Figure 1

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.

Hierarchical Bloom Filter



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 dicuss 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, ince each bit vector is on it's 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 theoeretical 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 unneccesarily 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 likelyhood a pagefault will occur. In this analysis we will assume more page accesses is coorelated with a higher liklihood of page faulting. We can formly compute the expected number of different pages that will be asccessed (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. Natuarelly, 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 discussed 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 indictor variable A_i which is 1 if bit i is set, 0 otherwise.

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

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 accesed 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 probablity A_i is accessed at least once is:

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

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

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

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 anticpate:

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

If we wanted to store $327,680$ elements under a similar setting, then the number of pages 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 anticpate 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 anticpate much better performance.

3.2 False Positive Rate

3.2.1 Standard False Positive Rate

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

3.2.2 Hierarchical False Positive Rate

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 bloom filter is anticpated to have $\frac{nl}{m/P} = \frac{nlP}{m}$ elements in it. There are m/P bloom filters and for any element we choose l filters. 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.

$$l = 1\ ;\ k = 7$$

3.2.3 Hierarchical Implementation

Suppose we have a allocated m bits in total chunked into w bit vectors of size P bits:

$$m = wP$$

Moreober, suppose we anticipate to insert n elements are we allocate

Assuming our hash function's output is uniform, we can conclude that for

$$f_p = (1 - (1 - \frac{1}{m}^{nk}))^k \tag{2}$$

4 Experiments

We will now emperically validate that the hierarchical implementation is more time efficient than the standard implementation without sacrificying accuracy. Two experiments will be conducted. First, we will **measure elapsed time as insertions scale** of each implementation. We anticpate 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 hierarchial and standard implementation will be approximately same as the theoerical 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.

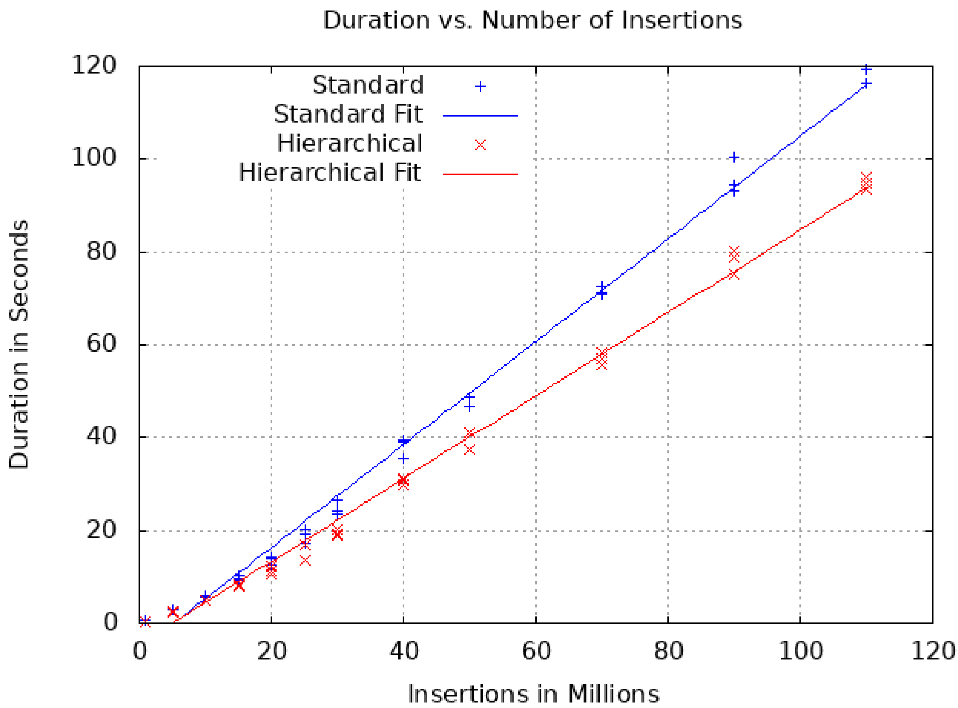
4.1.1 Experimental Settings

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

1. Generate n random keys.
2. Generate a bloom filter of both varianets of size $10n$. Use the optimal theoeretical configuraiton 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 whre 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:

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

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

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

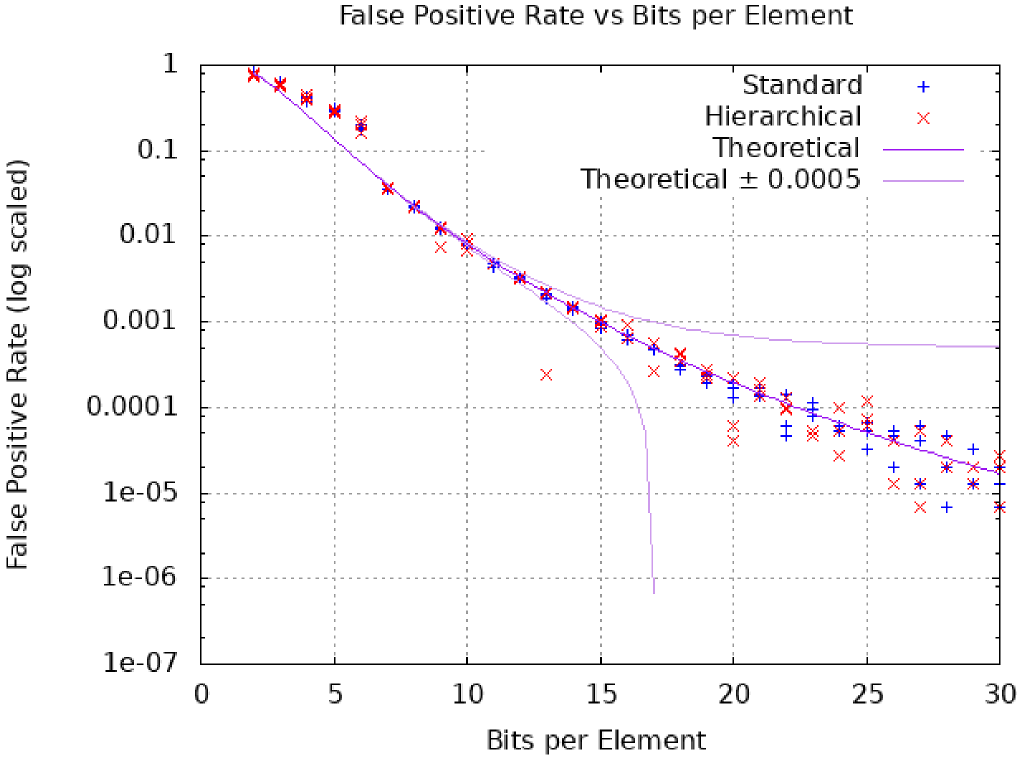
For this experiment, we seek to validate that the hierarchical implementation does not have a worse false positive rate than the standard implementation. *Hypothesis:* We expect them to have approximately the same false positive rate as the theoeretical expectation discussed earlier.

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 implementaiton 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 theoeretical expectation. Both implementations do worse than theoeretically 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 hypothesises. 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 Application

6 Literature Survey

In my literature survey, I discovered a similar approach except making bloom filters hierarchial. Tim Kaler's proposed cache-efficient bloom filters [3] makes sub-bloom filters of size cache-size; thus, their idea is very similar to mine except they do it on a smaller unit of memory.

Evgeni Krimer and Mattan Erez used a power-of-two choice principle within blocked-bloom filters to decrease the false positive rate. Instead of simply selecting one block to write into, they choose multiple. [2].

7 Conclusion

References

[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

[2] Mattan Erez Evgeni Krimer. The power of 1+alpha for memory efficient bloom filters. *Electrical and Computer Engineering Department University of Texas at Austin*. URL `http://lph.ece.utexas.edu/users/krimer/pub/im11_bf.pdf`.

[3] Time Kayler. Cache efficient bloom filters for shared memory machines. *MIT Computer Science and Artificial Intelligence Laboratory*. URL `http://tfk.mit.edu/pdf/bloom.pdf`.