A Brief Introduction to Approximate Membership Query Data Structures

Robin Swanson

University of Manitoba
COMP4420 - Advanced Analysis of Algorithms
[umswans5@cc.umanitoba.ca

April 2014

Abstract

In this paper we discuss and evaluate two probabilistic data structures, the bloom filter and the quotient filter which belong the class of approximate membership query (AMQ) data structures. Both data structures promise fast updates and searches and compact space requirements through the use of hashing techniques. However, unlike a traditional hash table, false positives, i.e., a query returns true despite the value not being in the hash table, are possible. Here we compare their merits, use cases, as well as analyze their run time, space requirements, and cache performance. We found that our Bloom Filter had better query performance while using less space. The Quotient Filter, on the other hand, was able to guarantee much better false positive ratios at the cost of more disk space.

Keywords: Bloom, Bender, Hash, Quotient, Filter

1. Introduction

Traditional hashing tables map values to indexes, or buckets, through the use of hashing techniques. In an ideal scenario each value would be hashed to a unique index for storage. However, due to limitations in space and perfect hashing techniques many coping methods have been introduced to account for collisions (when two unique values evaluate to the same index). Many collision resolutions exist including chaining, open addressing, and cuckoo hashing (among many others). Approximate membership query data structures provide only probabilistic resolution of collisions, meaning while false negative queries are not possible there is a probability of returning false positive queries. This probability depends on the size of the hashing area, the number of values inserted, as well as other implementation dependent variables.

At the cost of probabilistic errors, AMQs provide space and time efficient means of querying a set. These techniques are useful for applications in which a large data set would be impossible to store in resident memory. A fast AMQ could be checked, and if present, a more costly storage based lookup could be performed. In particular, applications where the vast majority of queries are presumed to be false can benefit from the reduced storage footprint at minimal costs.

Some uses of AMQs include Google's BigTable which uses bloom filters to reduce disk lookups of non-existant rows and columns[3], Bitcoin to verify payments[4], and by various chemical structure databases to lookup molecular substructures[5].

2. Probabilistic Data Structures

2.1 Bloom Filter

The Bloom Filter was introduced by Burton Howard Bloom in 1970 [1]. Bloom described a variation of the traditional hash table which, for a bit vector of size *m* bits, would evaluate *k* unique hashes for every value inserted and set each corresponding bit to one. Queries, then, would simply be to check all *k* hashes and return true if and only if each corresponding bit is set to one.

Bloom also gave the example application of a hyphenation dictionary of 500,000 words. In this dictionary 90% of hyphenations followed simple rules while the remaining 10% required specific patterns stored on disk. He was able, with the use of his Bloom Filter, to create a hash of only 15% the size of an ideal error-free hash which eliminated 85% of disk queries.

2.1.1 Probability of False Positives

For a bit vector of size m with k hashing functions we know the probability for any hash function to not set a bit to one is:

$$1 - \frac{1}{m} \tag{1}$$

So, for all hashing functions the probability a bit is not set to one is:

$$\left(1 - \frac{1}{m}\right)^k \tag{2}$$

And if we have already inserted n elements, the probability for any bit to still be zero is:

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m} \tag{3}$$

Finally, assuming some fraction of the m bits remain unoccupied, the probability of each of these k elements being set to one for a value currently not in the set (i.e., a false positive) is give by:

$$(1 - e^{-kn/m})^k \tag{4}$$

Which approaches 1 in both the limits of n approaching m and k towards infinity as expected.

2.1.2 Optimal Number of Hash Functions

Given our previous results, we can then optimize k, the number of hash functions, for a given m and n to provide a minimum false positive chance.

Here we let $g = k \cdot ln(1 - e^{-kn/m})$ and $f = e^g$. We can then minimize the false positive probability f by minimizing g with respect to k.

$$\frac{dg}{dk} = ln(1 - e^{-kn/m}) + \frac{kn}{m}\frac{e^{-kn/m}}{1 - e^{-kn/m}} \tag{5}$$

We can then find that this derivative is equal to zero when

$$k = \frac{m}{n} ln2 \tag{6}$$

Which is a global minimum for the function.

2.1.3 Costs

Similar to other hash tables, Bloom filters provide $O(1)$ (constant) insertion and lookup times. However, as the number of hash functions increases, the constant cost of computing all k hashes can become very large.

Deletions, without completely rehashing all values, are not possible due to the fact that any item in the hash table may have common bit indices with any number of other items. Removing any one item may result in false negative lookups for other items. Therefore, deletions require $O(n)$ time to rehash each value.

A Bloom Filter is composed of exactly m bits, giving us $O(m)$ space.

2.1.4 Variations and Extensions

In recent years many variations of the Bloom filter have been proposed to improve some of its weaker points. Some of these include Counting Filters, proposed by Fan et al.[9], provides the ability to delete items from the hash table without rehashing. Bloomier Filters, proposed by Chazelle et al.[10], allowing arbitrary functions to be performed on the hash table as opposed to queries. And Scalable Bloom Filters, proposed by Almeida et al.[11], which allows the size of the Bloom Filter to scale as items are inserted to keep a reasonable false positive rate.

2.2 Quotient Filter

The quotient filter was introduced in 2011 by Bender, et. al [2]. Similar to the Bloom filter, the quotient filter provides space efficient means to provide a probabilistic set query. Given the result of a hash value p bits in length, we break the result into a quotient, q bits in length, and remainder, r bits in length. This is a technique known as *quotienting*, which was first coined by Knuth[6]. To store the values we create 2^q slots, each of which has 3 bits containing the fingerprint and another r bits to hold the remainder.

To insert or query a value we first take its hash and break it into quotient (q) and remainder (r). We check the fingerprint at position q and, if empty, insert r into the remainder at position q. If the fingerprint is not empty it describes what type of value is currently inserted at that position. The value that is currently occupying slot q could either be another value that hashed to the same value of q (all of these values together form a *run*) or another value that was shifted higher in the bit vector by a value before it. The initial run, as well as any number of subsequent runs form a *cluster* which can be terminated by either a new cluster or an empty slot. The fingerprint at each slot describes the various combinations of run and cluster positions, e.g., an empty slot is denoted by 000, 100 denotes a run in its canonical slot, 111 is the continuation of a run shifted from its canonical slot, etc.

A query is comprised of finding the initial fingerprint and following the fingerprints to where that value should be placed. If the remainder in that position matches r we return true. Recall, however, that multiple values could hash to identical qs and rs, thereby giving us a possibility of false positives.

An insertion works much in the same way as a query. Once the slot where the value should be is found, if the remainders do not match, the subsequent slots are shifted further along and their fingerprints are updated if required.

2.2.1 Probability of False Positives

Given a quotient filter of size m, which is occupied by n elements, we define its load factor to be $\alpha = n/m$. Therefore, as described by Bloom, et. at, [2], given a good hash function, the probability of a false positive is described as:

$$1 - e^{\alpha/2^r} \leq 2^{-r} \tag{7}$$

Where r, again, is the number of bits describing the remainder.

2.2.2 Costs

The time required for insertion can, it its very worst case, require up to $O(n)$ time. This could occur if the load factor was quite large or through the use of bad hashing techniques.

Bender, et al.[2] argue, through the use of Chernoff bound analysis, that with a reasonable load factor and a uniformly distributing hash function, there is a high probability that most runs will be of length $O(1)$ and that all runs will be of length $O(logm)$.

Due to the fact that, with proper load factors and hashing functions, most insertions and queries are $O(1)$ time, quotient filters are often faster than Bloom filters due to the fact that only a single hash is generated [7].

Deletions are possible with quotient filters. However, much like insertions can provide false positives, there is no guarantee that the value you are removing is the actual value you intended without a priori knowledge of the contents. There is also the possibility that a second item attempted to insert itself into the quotient filter but was ignored due to a false positive which would effectively result in the deletion of multiple items.

Much like the bloom filter, the quotient filter requires $O(m)$ space. However, while the bloom filter required only $O(1)$ bits for each slot, the quotient filter requires $O(3+q)$ bits for each slot. In practical applications this results in a 10-15% increase in space compared to a Bloom filter of similar capacity and false positive rate [7].

2.2.3 Variations and Extensions

As noted by Bender[2], quotient filters have the innate ability to map their values to integers allowing two quotient filters to be merged (in a process similar to merge sort) without rehashing any values. This allows an extension known as the Cascade Filter, a collection of quotient filters organized into a data structure similar to a Cache-Oblivious Lookahead Array[12].

3. Performance Analysis

3.1 Implementation Details

3.1.1 Bloom Filter

While Bloom filters are fairly straightforward to implement, there is some room for improvements. As an alternative to having k unique hash functions or techniques, Kirsch et. al [8], describe a technique to simulate any number of hashes from two original hash functions. Given two hash functions $h_1(x)$ and $h_2(x)$ a new hash function $g_i(x)$ can be generated by:

$$g_i(x) = h_1(x) + i \cdot h_2(x) \tag{8}$$

Where i is any real number. In our code we generate k hash functions from two original functions and a for loop over $0 \leq i < k$.

3.1.2 Quotient Filter

In our quotient filter we have two distinct bit vectors, one for the fingerprints and one for the remainders. We also restrict the size of r to that of an unsigned integer, giving us a bit vector a size of $q \cdot sizeof(unsignedinteger)$. In this way writes and reads to the remainder vector require constant time and require no bit fiddling to find its index. Fingerprints are also restricted to the minimum three bits such any fingerprint will occupy at most two indices in the fingerprint bit vector which has a size of $3 \cdot sizeof(unsignedchar)$. In this way we can unroll our lookup and read loops, slightly increasing the speed of our queries.

3.1.3 Hashing Functions

While cryptographically secure hashing functions would most likely provide better distribution of values, resulting in fewer hard and soft collisions, their associated run-times would dramatically increase the constants associated with our insertions and queries. The naive implementation of a Bloom filter, in particular, would suffer greatly. In addition, our efforts to deal with collisions (e.g., the Bloom filter's k hashes) reduce our need for a perfect distribution. In one real-world implementation of a Bloom filter [1] a speed increase of nearly 800% was achieved by switching from the cryptographic hash function md5 to murmur hashing.

For our implementation we've used Google's CityHash [2] for the quotient filter and Murmur3 [3], in addition to CityHash, for the Bloom filter's second generating hash function. Both hashing functions were chosen for their speed, resulting hash distribution, and ease of use. In earlier versions of our software original implementations of the hashing functions were created. However, these proved to be unreliable and were quickly replaced with the official implementations as an original hash implementation was already beyond the scope of this project.

3.2 Evaluation Environment and Tools

All testing of our data structures was performed on the Aviary computer systems hosted at The University of Manitoba. They were chosen for up-to-date performance, collection of profiling software, as well as their availability to us.

All timing operations were performed using the built-in c time functions included in $< time.h >$.

All cache performance was evaluated using the valgrind[4], and more specifically the cachegrind[5], tool-set. Cachegrind emulates a Harvard architecture cache for a given program and records the number of instruction, as well as data, read, and write cache hits and misses. For our purposes we are only concerned with data cache misses, and thus the instruction misses are ignored in our discussion.

3.3 False-Positive Performance

To test the false positive rate of our data structures, we varied the free parameters in each corresponding equation and attempted to query 2^{26} (67108864) values known to not be in the set. In the figures below we plot the rate of false positives versus one variable while holding the

[1]Dablooms pull notes – https://github.com/bitly/dablooms/pull/19

[2]Google's CityHash family of hash functions – http://code.google.com/p/cityhash/

[3]MurmurHash3 – http://code.google.com/p/smhasher/wiki/MurmurHash3

[4]Valgrind – http://valgrind.org/

[5]Cachegrind – http://valgrind.org/info/tools.html#cachegrind

others constant for both the Bloom and Quotient Filter. For comparison we also plot the expected theoretical number of false positives using the equations derived earlier.

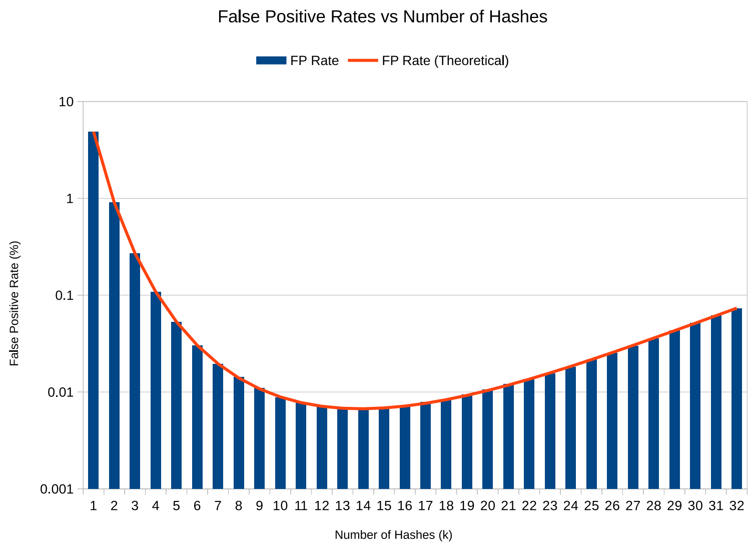

Figure 1: *Rate of false positives on a Bloom Filter with a constant inverse load factor of 20 and varying the number of hashes used. Note the minimum at* $20 \cdot ln(2) = 13.86$ *– the theoretical optimal number of hashing functions.*

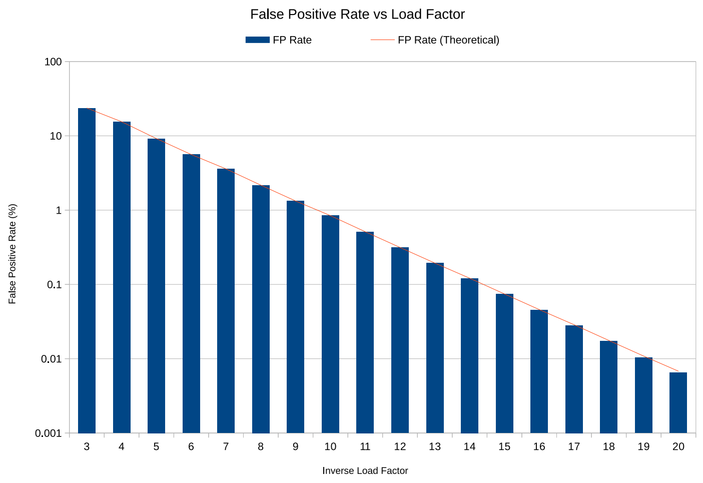

Figure 2: *Rate of false positives on a Bloom Filter using the optimal number of hash functions while varying the load.*

3.4 Run-Time Performance

To evaluate the query performance, we evaluated both filters with varying load factors under optimal conditions. For the Bloom Filter optimal meaning the optimal number of hash functions for each load factor. The Quotient Filter, on the other hand, was provided enough remainder

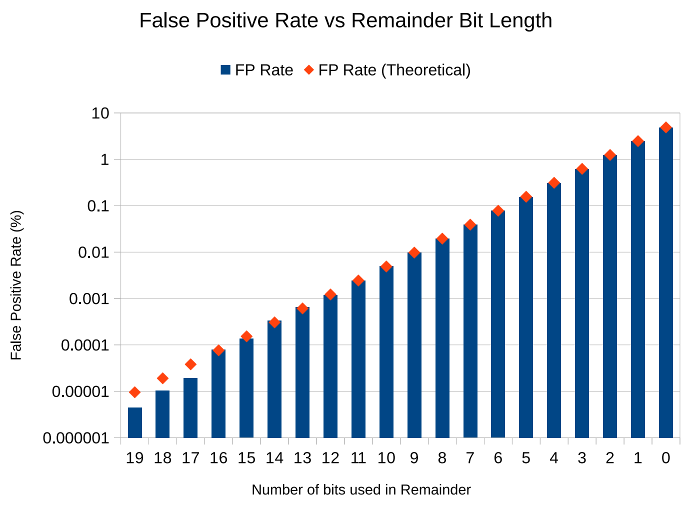

Figure 3: *Rate of false positives on a Quotient Filter with a constant inverse load factor of 20 while varying the number of bits stored by the remainder.*

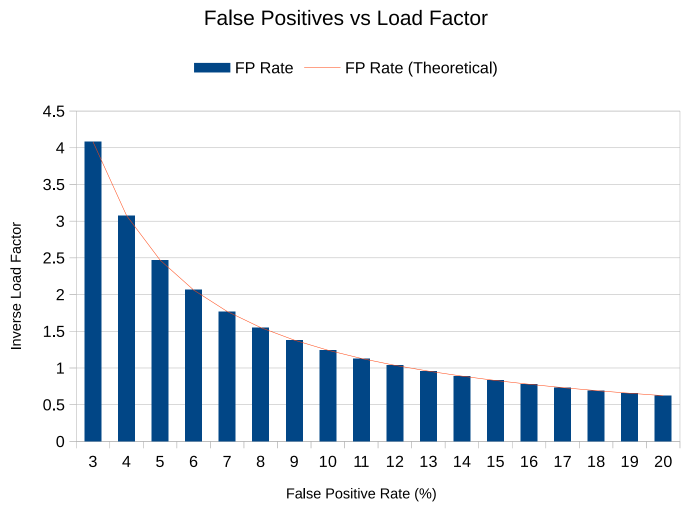

Figure 4: *Rate of false positives on a Quotient Filter using a constant remainder bit length of three while varying the load.*

bits to effectively have a zero chance of encountering a false positive. Multiple batches of 2^{27} (134217728) random queries were performed at each load factor, which were then averaged and plotted with (standard deviation) error bars.

3.5 Cache Performance

First we evaluated the number of cache read and writes to ensure that a similar number of cache reads and writes were being performed by both the Quotient and Bloom filter. Comparing Read

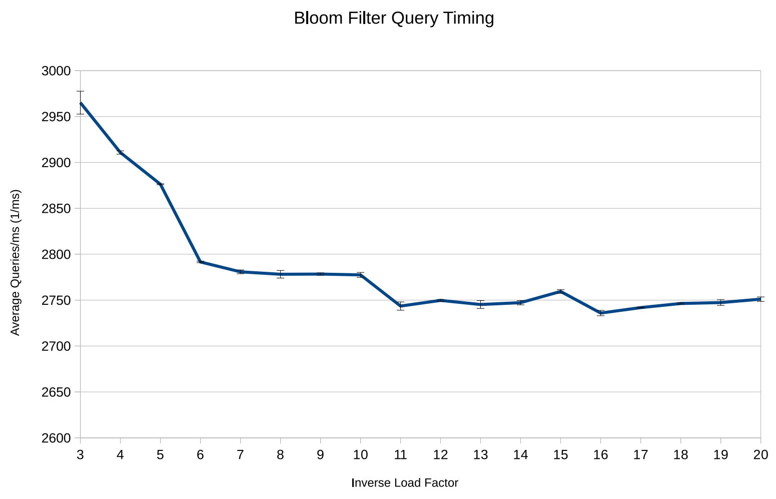

Figure 5: *Number of random search queries per millisecond the Bloom Filter can perform at varying levels of load factors (higher is better).*

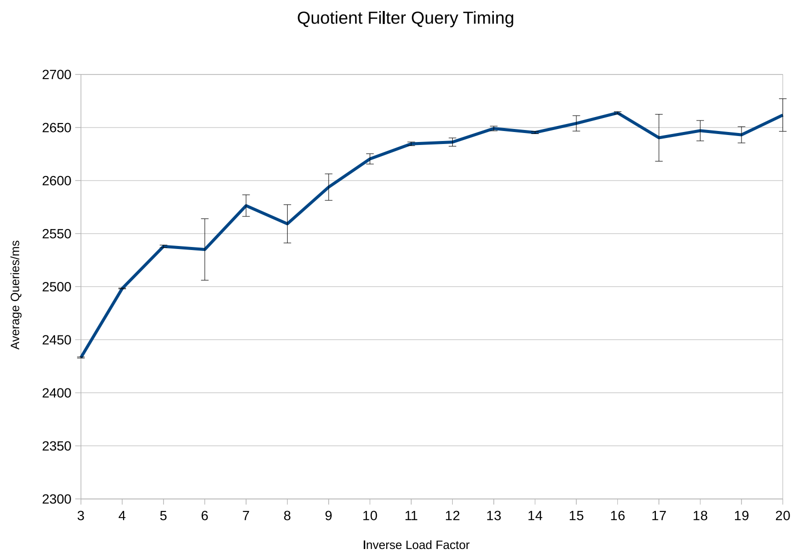

Figure 6: *Number of random search queries per millisecond the Quotient Filter can perform at varying levels of load factors (higher is better).*

and Write miss ratios would be very misleading otherwise. As described in the graphs below, this is definitely the case. At average load factors both filters perform nearly identical amounts of cache reads.

Similar to our other profiling techniques we looked at both filters, varying their loads and performing a large number of queries. As it turns out Valgrind/Cachegrind increase the runtime significantly so the number of queries was reduced to 2^{25}.

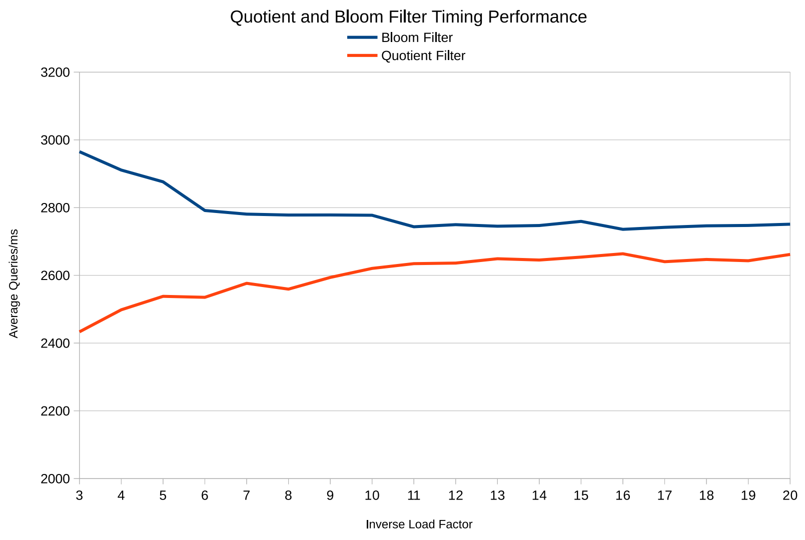

Figure 7: *Number of random search queries per millisecond both the Bloom and Quotient Filter can perform at varying levels of load factors (higher is better).*

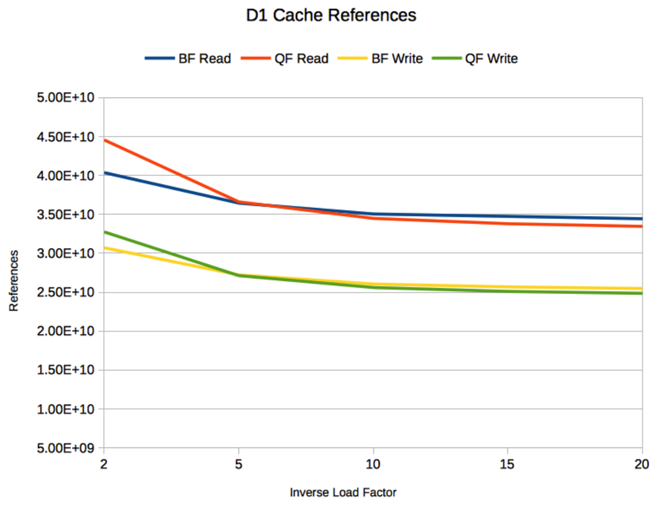

Figure 8: *Number of cache reads and writes performed by both the Quotient and Bloom filter at varying inverse load factors.*

4. Discussion

We found our data structures to very closely follow the theoretical false-positive rates while maintaining a zero false-negative rate. Therefore it is very likely that both the statistical methods of Bloom and Bender are correct, and that our data structures were properly constructed to specification.

Our data also shows that in run time performance the Quotient Filter does not live up to its expectations. At all load levels the Bloom filter is able to perform more query operations per

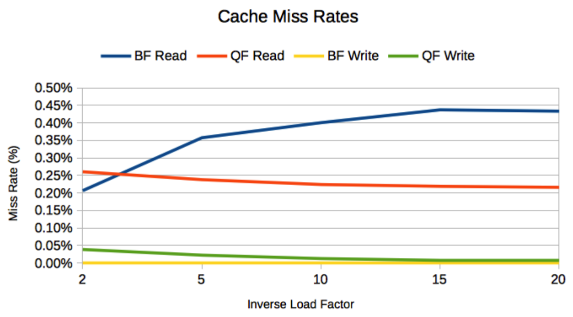

Figure 9: *Cache read and write miss ratios for both the Quotient and Bloom filter.*

second. This may be due to the fact that we have optimized the generation of our hash functions beyond the naive implementation originally described by Bloom. The higher variance in the Quotient Filters timings may be explained by the fact that Quotient Filters only have expected constant insertion and searching. Certain sequences of insertions and queries could result in slightly higher running times, although it is very unlikely.

There is the possibility that some of the Quotient Filters performance issues were due to suboptimal implementation. However, Bender et al., [2] do not provide much more than a guideline and pseudo-code which was, of course, carefully followed and implemented to the absolute best of our ability.

Cache performance was less illuminating that we had hoped. While there very clearly similar numbers of cache accesses at reasonable load factors, and demonstratively less cache read misses for the Quotient Filter in these same regions, the differences are not particularly significant. Further cache analysis could be performed at higher RAM loads, i.e., when there is significant swapping to virtual memory. However, we decided an appropriate analysis for this type of data structure would be at typical use cases. Therefore the analysis was done where the data structures could be contained entirely in RAM and with average load factors.

5. Conclusion

We found that a well implemented Bloom Filter to have better run-time performance of queries on a static data structure at all load factors. The Quotient Filter, on the other hand, could provide much better false positive rates given adequate space while also having slightly better cache read performance.

Currently a case can be made for use of both data structures, in particular if further modern optimizations are made to the Bloom Filter. It remains up to the user to determine what are acceptable values of space, time, and false positive ratios.

A. Source Code

All source code for both the Bloom Filter and the Quotient Filter are available on our GitHub page[6].

B. Special Thanks

A special thanks to our close, personal friend, Michael Zapp for entertaining our numerous and lengthy questions about software performance and cache profiling.

References

[1] Bloom, B. H "Space/Time Trade-Offs in Hash Coding with Allowable Errors"

[2] Bender, M. A; et al, "Don't Thrash: How to Cache your Hash on Flash", PVLDB, 5(11):1627-1637, 2012.

[3] Chang, Fay; et. al (2006), "Bigtable: A Distributed Storage System for Structured Data", Seventh Symposium on Operating System Design and Implementation

[4] Hearn, M (2012), "Bitcoin Improvement Proposal 0037: Connection Bloom Filtering" *https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki* (Accessed March 22, 2014)

[5] Baldi, P; et. al (2008), "Speeding Up Chemical Database Searches Using a Proximity Filter Based on the Logical Exclusive OR", Journal of Chemical Information and Modeling, 2008, 48, 1367–1378

[6] Knuth, D (1973). The Art of Computer Programming:Searching and Sorting, volume 3. Section 6.4, exercise 13: Addison Wesley.

[7] Spillane, R. P; (2012), "Efficient, Scalable, and Versatile Application and System Transaction Management for Direct Storage Layers", Ph.D. Thesis, Stony Brook University

[8] Kirsch, A.; Mitzenmacher, M; (2006), "Less hashing, same performance: Building a better bloom filter", In Proc. the 14th Annual European Symposium on Algorithms

[9] L. Fan, P. Cao, J. Almeida, and A. Z. Broder; (2000), "Summary cache: a scalable wide-area Web cache sharing protocol", IEEE/ACM Trans. on Networking, 8(3):281-293

[10] Chazelle, Bernard; Kilian, Joe; Rubinfeld, Ronitt; Tal, Ayellet (2004), "The Bloomier filter: an efficient data structure for static support lookup tables", Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 30âĂŞ39

[11] Almeida, Paulo; Baquero, Carlos; Preguica, Nuno; Hutchison, David (2007), "Scalable Bloom Filters", Information Processing Letters 101 (6): 255âĂŞ261

[12] Bender, M. A; et al, "Cache-Oblivious Streaming B-Trees" Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS), pages 399-409, 2000.

[13] Mitzenmacher, M.; Upfal E.; "Probability and Computing: Randomized Algorithms and Probabilistic Analysis"

[6] Abloom on GitHub – `https://github.com/swansonr/Abloom`