The Great Firewall of Santa Cruz

Moore Macauley
Glorious People's Republic of Santa Cruz

November 18, 2022

1 Introduction

Now that I have created filters to protect the citizens of the Glorious People's Republic of Santa Cruz (see DESIGN.pdf), their efficiency must be tested. After all, it would not do for our subjects to have their internet access slowed too much by the filter. So, to ensure that our subjects don't notice the filter- I mean, we don't worsen their user experience, we must ensure the effectiveness of the filters, and optimize their settings to make sure they work best. All of the below tests were done using the included test_design.tex, which is an older version of my design source file.

2 Bloom Filter Bits Examined Per Miss and Bloom Filter Size

To begin with, I compare the number of times the bloom filter misses compared to the total size of the bloom filter, which can be seen in figure 1.

Looking at this graph, we see that up until the bloom filter is around size 6000 or so, the number of bits examined per miss sits at 0. This is for a very simple reason, the bloom filter is completely full. There are about 14 thousand badspeak words included, and every one sets in (provided no collisions) 5 bits in the bloom filter. With a size of only 6000 or less, every bit is full, so every word maps to 5 set bits. Once we get past this initial hurdle, the bloom filter is only very close to being full, so having no misses is still not uncommon, and when there is a miss, the filter usually has to look through all 5 bits before it encounters one that is not set. This continues on until around 10,000, where the still overloaded bloom filter starts

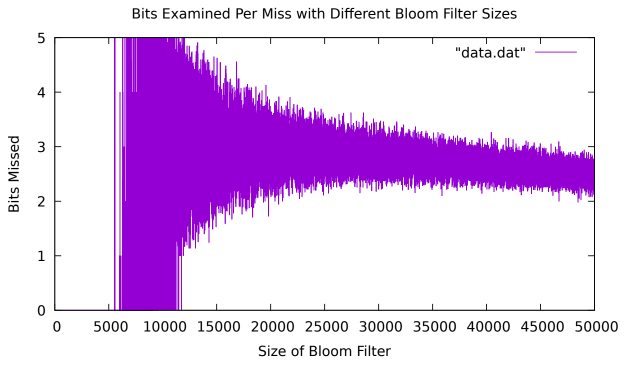

Figure 1: An interesting shape

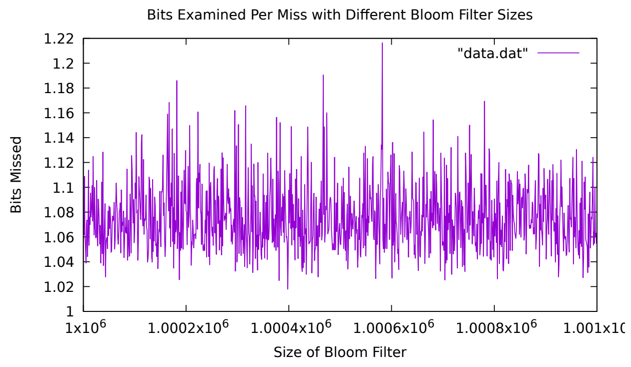

Figure 2: Around 1-ish

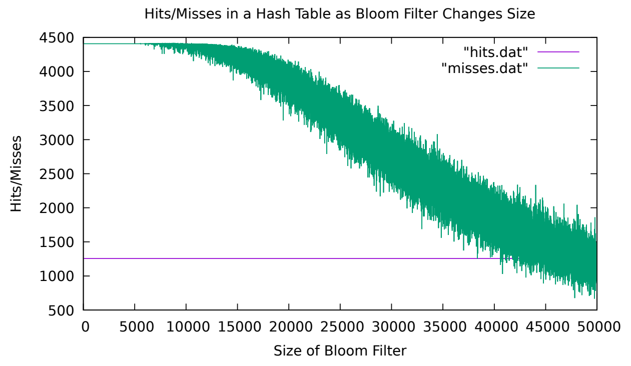

Figure 3: Another odd shape

only having to look through 4 bits to find a miss. We then see a slow but noticeable downward trend, with the average number of bits examined sitting around three and decreasing.

The conclusion of this trend can be seen in figure 2, shows the same thing as figure 1, but with the bloom filter sizes around a million. Here, we see that the number of bits examined per miss approaches 1, which makes a lot of sense. As the filter gets bigger, the odds of having conflicts goes down greatly. As such, when the filter misses, it usually only has to look at one unset bit before recognising that the word will not be part of the hash table.

3 Bloom Filter Size, Hash Table Lookups, and the False Positive Rate

When comparing hash table lookups, I began by graphing the size of the bloom filter against the number of times the hash table contained an item the bloom filter said it should (hits), and the number of times the bloom filter was wrong (misses). This graph can be seen in figure 3.

Here, we see that hits stay the same throughout. This is rather unsurprising, as the test file always contains the same amount of bad and oldspeak, so hits should stay the same. However, for misses, we see as the bloom filter size increases, the number of misses decreases. To understand why, we need to look at figure 4.

In figure 4, we compare the percentage of false positives of the bloom filter to the size of the bloom

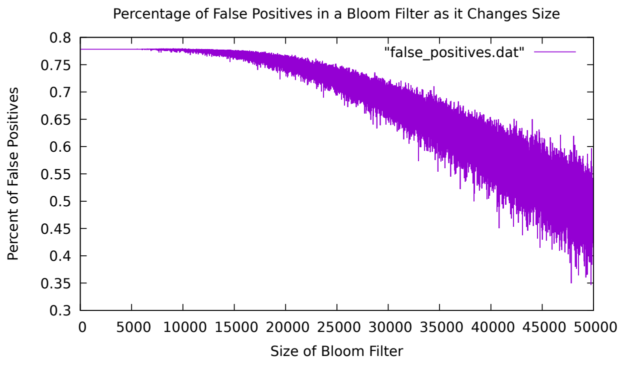

Figure 4: This graph looks rather familiar...

filter. In this scenario, a false positive is when the filter flags a word as being unacceptable, only for it to actually not be on the list of old/badspeak. We start around 78%, which is simply the percentage of acceptable words in the test file. Then, as the size increases, the percentage of false positives goes down. This makes sense due to the reasons described above. The bloom filter being used is so small compared to the number of badspeak words and oldspeak words, that every bit inside the filter is set, making it recognise every single word as being unacceptable. As the size increases, however, fewer internal bits are set, allowing it to not flag unacceptable words. Eventually, once the bloom filter gets big enough, the rate of false positives will reach 0, as it will be able to store every bad/oldspeak word without conflicts.

The relationship between this and hash table misses is very simple. A hashtable miss only occurs when the hash table is told to look up a word that isn't actually on the list of old/badspeak. My hash table is only told to look up a value after the bloom filter flags it as unacceptable, and as such, can only miss when the bloom filter gives a false positive. As such, not only does the number of hash table misses increase as the number of bloom filter false positives increases, the misses and false positives are actually the same.

4 The Effects of Move to Front and the Size of a Hashtable on Lookup Speeds

Our hash tables are chained hash tables, so at every element inside the table, there is a linked list. One of the options for our linked list is that every time an element in the list is looked up, we move that element to be right behind the head. To see the effects of this on the efficiency of our hash tables, I have graphed the number of linked list links examined in the hash table with move to front on and off, vs the size of the hash table. This can be seen in figures 5 and 6, with figure 5 showing the links traversed at small hash table sizes, and figure 6 showing the same for larger table values.

Unsurprisingly, when the table is very small and as such very overloaded, it has to traverse a lot of links. After all, when the hash table has a length of 1, it is, in effect, simply a linked list, as every value will hash to 1. Linked lists, with their O(n) for lookup, are much slower than the O(1) for hash table lookup (provided no collisions), so it follows that this makes the table very slow. But things improve quickly, as with a table length of even two, the number of elements that need to be iterated over every time we look a word up is halved. As we get larger and larger, the number of links traversed gets smaller and smaller. As can be seen in figure 6, this will eventually approach the number of bad/oldspeak words in the entire file, which in this case is 1258 (provided the bloom filter is also large enough and creates no

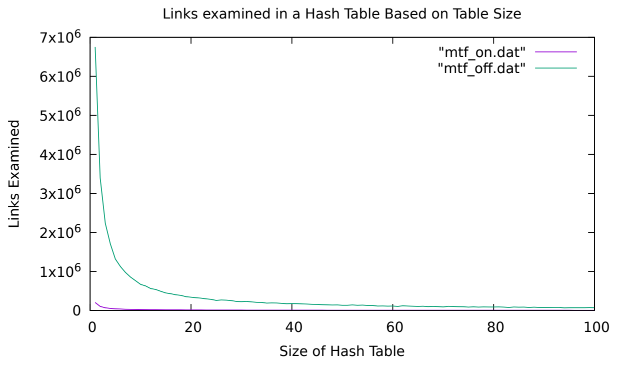

Figure 5: The hash table is rather inefficient when it is small

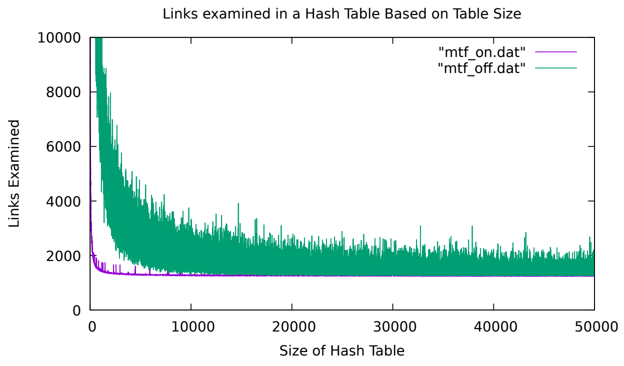

Figure 6: Things get better as it gets larger

false positives). This is because eventually, every word will simply have its own linked list, independent of any other values. So when the table hashes to that element, the only link that needs to be traversed is from the head to the first element, at which point the value is found. Therefore, as the size of the hash table increases, the number of linked list elements that needs to be traversed decreases, due to the number of collisions decreasing, and as such the table gets faster at lookups.[1]

When it comes to the move to front option, we can see in both figures 5 and 6 that the number of linked list links traversed decreases drastically when move to front is enabled. This is for a rather simple reason. In my infinite wisdom as the leader of the Glorious People's Republic of Santa Cruz, I have elected to make some very common words, such as "a" and "I" badspeak. On the other hand, I have also made some very uncommon words badspeak, such as "inveigb" or "zwounds". If we consider a hypothetical scenario, where "a" and "zwounds" are both used by a user, where they both hash to the same hash value, and "zwounds" is used first, every time "a" is used by the user from that point on, the hash table will have to iterate over the node containing "zwounds" first, greatly increasing the number of links traveled. If the hash table was very small and particularly overloaded, the table could need to iterate over many uncommon words to get to something like "a", making the situation even worse. With move to front enabled, however, any common words used repeatedly would be shifted over to the front, and the uncommon words would simply not be touched while they were being used. As we can see,

[1]This paragraph is my answer for question 4. Fit better before the question 3 answer.

this greatly decreases the number of links traversed especially when the table is small and overloaded. However, in figure 6, we also see that two cases gets closer as time goes on. This is because move to front only improves the links traversed where there are collisions and multiple elements in the internal linked lists. As such, as the hash table gets bigger and the collisions decrease, as discussed above, the effects of move to front being enabled decrease. Eventually, once the table is big enough and there are no collisions, enabling move to front will make no difference at all, as a linked list with one node will always have that node first.

5 Conclusion

Putting this all together, we find that as the size of a bloom filter increases, the number of bits it has to examine decreases. On a similar note, every time the bloom filter has a false positive, the corresponding hash table fails to find a value. As we increase the size of a hash table, the number of collisions decreases, and the speed of lookups increases until there are no collisions, and enabling move to front on the table's internal linked lists helps the most when the table is overloaded.

Using this information, the best banhammer possible for the GPRSC is probably one with a large hash table and bloom filter, large enough that there are no/very few conflicts with the current list of bad/oldspeak. We then should have move to front on, simply because the list of things that annoy me person- I mean corruptive for elements in the government is ever growing, and we want the filter to continue to run fast, should it expand beyond what the bloom filter and hash table can contain without colliding.

Now that my subjects are safe from the internet, I plan to live out the rest of my days ruling in luxu- I mean in equal quarters to all the workers and crushing political disaden- I mean removing foreign spies from the country, all before dying of a preventable stroke after purging all the doctors. As every good communist leader should.