PA1 Report

Bloom Filters

Alex Huynh

Com S 435
Iowa State University

1. To generate k hash functions, I added additional characters to the string. This notion is called salting. More specifically, for every ith hash function, I would add i "+" signs and then the value of i. For example, if the string was "ABC" and the number of hash functions was 3, then the 3 values to be hashed would be ABC0, ABC+1, ABC+1++2. The rationale behind this process is that I needed a way to add additional randomness in a deterministic way. The way FNV works is that it disperses the intrinsic randomness in the bytes of the string. So while it may be the case that "Alice" and "Bob" have the same hash code, it is possible that "Alice0" and "Bob0" do not have the same hash code. I built on top of this idea by adding multiple additional characters in the form of "+" and numbers corresponding to the number of hash functions. I found this to result in less false positives compared to a more simple salting regime.

2. To generate k hash values, I first randomly generated all the primes between M and $2M$. I then select 1 random prime from that set. That random prime is p. Next, I had to generate values of a and b, of which there are k of each. a and b were randomly selected from $\{0, \ldots p-1\}$ k number of times. The sets formed from this process are A and B. Then finally to generate a hash value for a string s, for $i \leq i \leq k$, I would compute $(A_i + hash(s) + B_i)\%p$, where $hash(s)$ is Java's built-in hashCode() method for strings.

3. To compute false positives, I first determined two sets that were disjoint. I went with $A =$ the set of the first 100000 prime numbers. The second being $B =$ the set of the first 100000 composite numbers. Thus easily $\forall x \in B \implies x \notin A$. I stored all the elements of A in the bloom filter. Then I iterated through all of the elements of B and I checked whether the bloom filter said "Yes" to a search query. If it did, then that counted as a false positve as composites are not stored in the bloom filter, so I incremented a counter. To compute the false positve rate, I divided my counter by $|B| = 100000$. Also, I chose $|A| = |B| = 100000$ due to the law of large numbers and false positive rates converge as the number of elements approach infinity.

4. Using the approach laid out in the previous point, here are the false positive rates for 4,8, and 10 bits per element for the four implementations of bloom filter.

```
4 bits per element:
FNV BF: 15.114%
Ran BF: 26.686%
MM BF: 5.002%
Naive BF: 25.248%

8 bits per element:
FNV BF: 2.318%
Ran BF: 6.722%
MM BF: 0.367%
Naive BF: 11.86%

10 bits per element:
FNV BF: 0.837%
Ran BF: 4.946%
MM BF: 0.234%
Naive BF: 10.786%
```

Figure 1: False Positive Rates by Filter and Bits Per Element

By increasing bits per element, the false positive rate decreases.

Holding bits per element constant, and across all variations of bits per element, we see that the MultiMultiBloomFilter implementation had the lowest false positive rate, meaning it had the best performance in that regard. It had quite a bit lower false positive than even the next best performing filter, FNV. That is because of how MultiMultiBloomFilter works. It has k tables but also 1 hash function per table. Thus given the fact that false positives occur at a rate of $(0.618)^{M/N}$, where in this case $M = N = 1 \implies \frac{M}{N} = 1$, we can expect to exponeniate 0.618^k for we have k tables. Moreover, $1 \leq \frac{M}{N} < 2$ as we are using random hash functions and the size of the table can be as large as $2M - 1$. Thus this 0.618 ratio is only an upper bound, and we have a lower bound of $0.618^2 \approx 0.382$. Thus for a false positive to occur in MultiMultiBloomFilter, for all tables, whose probability of a tabular false positive is between 0.382 and 0.618, we multiply their rates together to get the total probability of a false positive. In the worst case where all tables have a rate of 0.618, then the false positive rate is $0.618^k = (0.618)^{bitsPerElement}$. This is exactly how the theoretical deterministic or random bloom filter's false positive rate is calculated. In the best case, the false positive rate is 0.382^k, which is much smaller than 0.618^k.

We now compare the difference between FNV and random bloom filter. Across the board, the FNV filter had lower false positive rates. That is due to FNV's ability to disperse randomness combined with the fact that salting makes the string have even more randomness in a way. The random bloom filter has worse performance because it can select a small prime p and limit the

size of the table as the size of the table is determined by p. Thus by chance if the random prime selected from $\{M, \ldots, 2M\}$ is near M, then the performance will be lower by chance. That is based on the false positive rate equation $0.618^{\frac{M}{N}}$. Additionally, selection of a and b can cause collisions if a and b are multiples among the k of them.

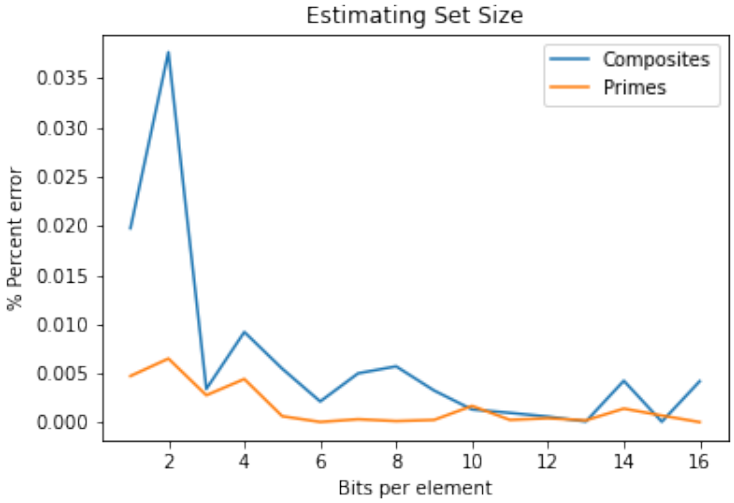
Naive bloom filter had the highest false positve rate out of all because while the size of the bloom filter was only proportional to bitsPerElement, the was only 1 hash function. In other words, $k = \ln 2\frac{M}{N}$ was not optimized. While its performance does increase as bitsPerElement increases, it does not increase as much as the other filters.

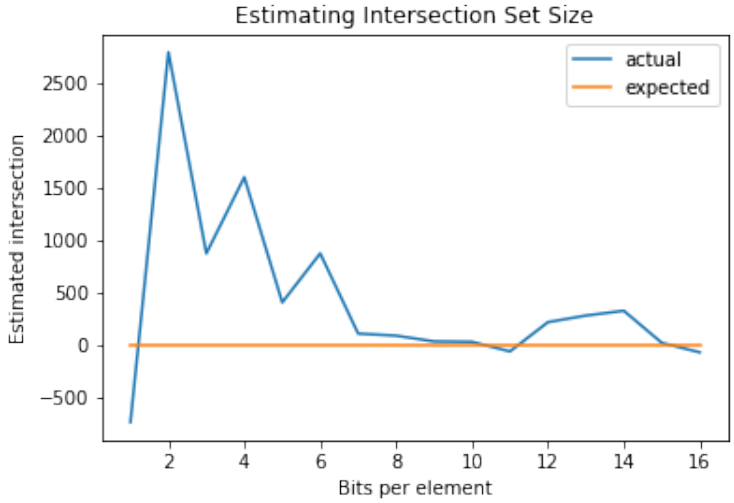
The theoretical false positive rates for the FNV and random bloom filter is approximately $(0.618)^{m/n}$. Thus in this application, we can simplify the false positive rate to be approximately $(0.618)^{bitsPerElement}$. We have the corresponding bits per element to false positive rate:

$$4 \to 14.59\%$$
$$8 \to 2.13\%$$
$$10 \to 0.81\%$$

FNV bloom filter had pretty close performance to this. Random bloom filter was a bit far off, but it stayed within a reasonable amount.

5. After implementing the equations to estimate bloom filter set size and the intersection set size, I tested the implementation using 2 bloom filters: one with 100000 composite numbers and one with 100000 prime numbers. Therefore, the expected set size of each should be 100000. This was the case as shown in the first figure below since the percent errors were low. We generally see that increasing bits per elements gives us a better (lower) percent error. Regarding intersection size, we know that the cardinality of their intersection set is 0. Since we cannot compute percent error with an expected value of 0, I simply plotted the estimated size of the intersection set. We see that the value approaches 0 as bits per element increases. Overall, these formulas and their implementation have good accuracy. (These were plotted using Python's matplotlib)





6. Using the assumption that access times to bloom filter are 0.001 seconds and access to any other data structure or file are 1.0 seconds, we assess the performance of the differential implementations by using the first 10000 lines of grams.txt as keys and measuring the amount of time to retrieve the records with these keys.

```
Total time: 18993.0
Average time: 1.8993
```

Figure 2: Naive Differential Results

It took approximately 1.9 seconds on average to retrieve the record given a key from grams.txt. In the best case, the key is in the hash map, so it takes 1.0 second to retrieve the key. In the worst case, the key is not in the hash map and so database.txt must be searched. This takes $1.0 + 1.0 = 2.0$ seconds in total. This happens $\approx 90\%$ of the time as only $\approx 10\%$ keys are in the hash map. Thus it makes sense for the average time to be 1.9 seconds. On the other hand, using bloom differential, access times were much lower.

```
Total time: 10202.000000001588
Average time: 1.0202000000001588
```

Figure 3: Bloom Differential Results

It took approximately 1.02 seconds on average to retrieve the record given a key from grams.txt. In the best case, the bloom filter says the key is in the differential file, and it takes $0.001 + 1.0 = 1.001$ seconds to retrieve the record. If the bloom filter says the key is not in differential file, then it still takes $0.001 + 1.0 = 1.001$ seconds to retrieve the record by going through the database file. In the worst case, the bloom filter answers with a false positive and so retrieval of the key takes $0.001+1.0+1.0 = 2.001$ seconds, having needed to access both files. The false positive rate is controlled by the parameters to construct the bloom filter. For realistic cases such as this, we had a false positive rate of $\approx 2\%$ (I used 8 bits per element in my test case), so overall our average time is still about just 1.02 seconds.

7. I additionally did test cases for false negatives and memory consumption. For false negatives, I added 100000 composite numbers and checked if each was in the bloom filter. We expected there to be none, and that is the case.

```
4 bits per element:
FNV BF: 0.0%
Ran BF: 0.0%
MM BF: 0.0%
Naive BF: 0.0%

8 bits per element:
FNV BF: 0.0%
Ran BF: 0.0%
MM BF: 0.0%
Naive BF: 0.0%

10 bits per element:
FNV BF: 0.0%
Ran BF: 0.0%
MM BF: 0.0%
Naive BF: 0.0%
```

Figure 4: False Negative Rates by Filter and Bits Per Element

To test memory consumption, I subtracted runtime.totalMemory() - runtime.freeMemory() before and after creating and filling the data structure with 100000 composite numbers and having set bitsPerElement = 100 (I did this one at a time by commenting out different parts of the code). The FNV filter had a surprisingly large size. This may have been due to salting.

```
100 bits per element:

319 MB
```

Figure 5: FNV Bloom Filter Memory

The random FNV filter had a much lower size.

```
100 bits per element:

56 MB
```

Figure 6: Random Bloom Filter Memory

MultiMultiBloomFilter had a small size relative to its performance.

```
100 bits per element:

69 MB
```

Figure 7: MultiMulti Bloom Filter Memory

The naive implementation of course had the smallest size.

```
100 bits per element:

33 MB
```

Figure 8: Naive Bloom Filter Memory

Overall, MultiMultiBloomFilter has the best ratio of performance to false positives (in my implementation at least).