Adam, Daniel, Dennis, and Maggie
COSC-254
Prof. Riondato
May 19, 2021

Group 2 Final Project Report

Introduction:

As the need to extract insights from data streams in place of static datasets increases, data structures for data streams are becoming progressively efficient. Optimal data structures for data streams are subject to certain computational constraints - processing data only once to prevent archival (one-pass constraint), evolution of statistical properties about the data stream over time (concept drift), external data generation resulting in variability in quantity at arrival times (resource constraints), and processing interactions between an extremely large quantity of discrete values (massive-domain constraint). Several synoptic data structures are designed to leverage some or all of the preceding constraints, especially to address the massive-domain situation. For our project, we decided to focus on and implement the Bloom Filter, Count-Min Sketch, and Cuckoo Filter data structures for data streams.

The Bloom Filter is used for set membership queries, and consists of a set of independent hash functions. When an item is passed into the filter, each hash function maps an integer from size 0 to m-1 into a bit array of size m. Since all entries in the bit array are initialized to 0, when an entry is mapped to it is then set to 1. Since multiple items may share bit array entries, false positives on items that are not currently in the filter are possible.

The Count-Min Sketch similarly uses a set of independent hash functions, yet is designed to estimate the approximate count of an item. The filter consists of a set of multiple bit arrays, each of size m, where each array is associated with an independent hash function that maps onto entries 0 to m-1 in the bit array. As an item is passed through the filter, each hash function increments the associated entry in its corresponding bit array by 1. By having multiple bit arrays, the filter effectively reduces an excess of collisions that would falsely inflate the count for a single item.

The Cuckoo Filter is also used for set membership queries as with the Bloom Filter, and similarly uses a set of independent hash functions. In the cuckoo hash, when an item is inserted, a fingerprint of that item is first made. Then the item is mapped to a bucket in the hash table by a hash function. If that bucket is free, it is inserted there but if it is occupied then it goes through a second hash function defined as the first hash function xor-ed with a different hash of the fingerprint of the item. If this alternative bucket is also full, then one of the buckets is randomly selected to insert the item into and the item that was already there is ejected to its alternative location. Items will continually be ejected until an ejected item has an available alternative location. To lookup an item, the cuckoo filter checks both hash locations for its *fingerprint* and if it is in either of the possible hash locations then it is determined to probably be a member of the set. As such, the Cuckoo Filter supports deletions (unlike the Bloom Filter) and offers better

lookup performance and space efficiency than the Bloom Filter. Our tests will look to replicate these findings.

Methodology:

For Phase 3, we decided our goal was to produce working implementations of all 3 data structures. In order to efficiently divide up the work, Daniel worked on the Bloom Filter, Dennis worked on the Count-Min Sketch, and Adam and Maggie worked on the Cuckoo Filter. All work was shared via GitHub and coded in Python using Jupyter notebooks for convenience during the data analysis phase. After receiving feedback from Matteo about incorporating our own hash function instead of mmh3 into each data structure, Dennis created polyhash that was used as a hash function in the Count-Min Sketch.

For Phase 4, each group member ran a series of experiments on their data structure to test different aspects their performance. For the Count-Min Sketch, we conducted a series of experiments on the accidents dataset to compare the exact and estimation algorithms under different parameters and hash functions. For consistency, testing for set membership in the analysis of the Bloom Filter and Cuckoo Filter was conducted using the vocab.nyt.txt file of words taken from New York Times articles (https://archive.ics.uci.edu/ml/machine-learning-databases/bag-of-words/) mentioned in the initial Group 2 Project Proposal. Elements to insert were chosen using a random sampling of approximately 20% of the words in the NYT dataset and the remaining 80% of words were used for set membership queries. Experiments for the Bloom Filter and Cuckoo Filter included an analysis of the false positive rate compared to the size of the filter. As the size of the filter decreases the probability of hash function collisions increases so we expect the number of false positives to increase. We additionally compared the actual false positive rate to the theoretical false positive rate for the Bloom Filter and Cuckoo Filter for each of the various filter sizes. See table below for each filter and its associated expression which captures the relationship between size (bits per entry), load size, and false positive rate. A more thorough description of experiments and comparisons between the Bloom Filter and Cuckoo Filter can be found below in the Results section.

Filter	**Size (Bits per Entry)**
Bloom	$1.44 \log_2\left(\frac{1}{\varepsilon}\right)$
Cuckoo	$\frac{\log_2\left(\frac{1}{\varepsilon}\right) + 2}{\alpha}$

Note that α = Load size, or fraction of buckets filled

Results:

Bloom Filter

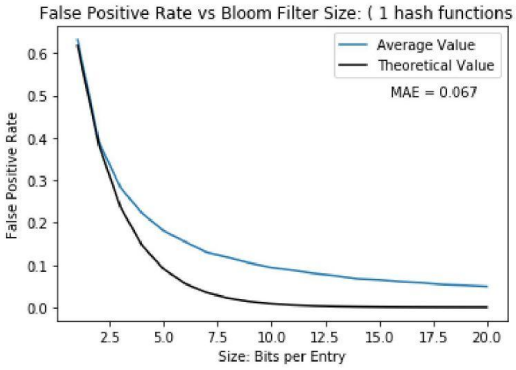
Testing Process-

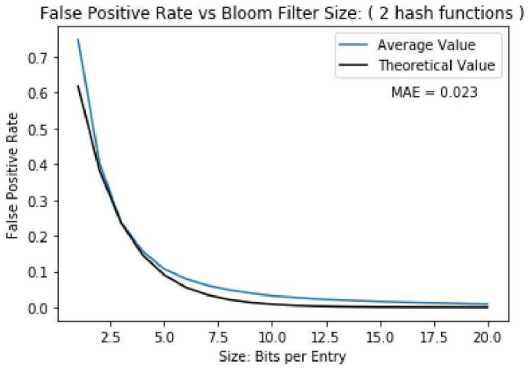
In our trials, we examined how changing the size of the bloom filter affects the rate of false positives for set-membership queries. To compute the false positive rate of a bloom filter, we randomly sampled 20% of the elements in the dataset to add to the filter. Then, we used the remaining 80% to perform set-membership queries on the bloom filter. If an element was not in the sample, but querying the bloom filter with that element returned True, then we recorded a false positive result. To determine the overall false positive rate for a specific size of bloom filter, we created five different bloom filters, one for each dataset in UCI Bag of Words Archive. We computed the false positive rate for each of the five bloom filters and then recorded the average false positive rate. We used this method to test 20 different sizes of bloom filters. We categorized bloom filter size by the number of bits allocated per element added to the bloom filter (bits per entry). We also experimented with using up to 8 different hash functions in the bloom filter. For each graph, we set a different num_hash parameter and we used that parameter to test all 20 different sizes of bloom filters.

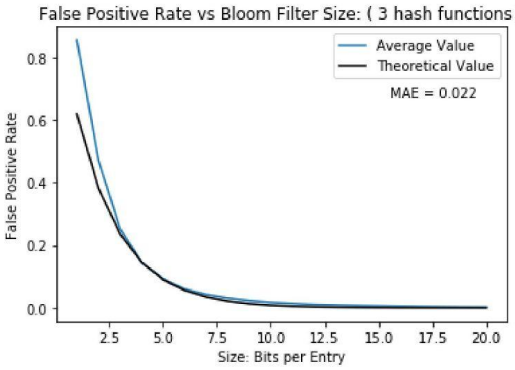
The second component of the bloom filter test involves checking our results by comparing them to the expected false positive values. We determined the expected false positive values by using the theoretical properties of bloom filters. We used the relationship between α (the size of the bloom filter in bits per entry) and ε (the false positive rate) to solve for the expected value of ε:

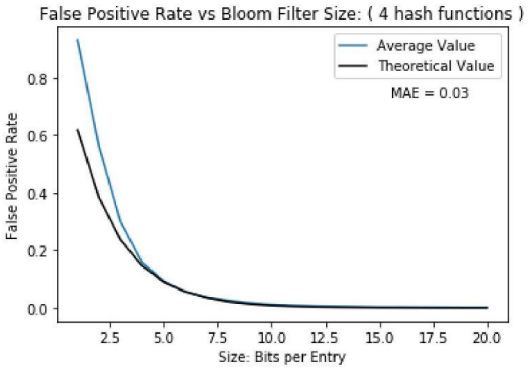
$$\alpha = 1.44 \log_2\left(\frac{1}{\varepsilon}\right) \Rightarrow \varepsilon = 2^{-\frac{\alpha}{1.44}}$$

Then, for each graph we plotted our results alongside the expected false positive rate. Lastly, to quantify the difference between the results and expected values, we computed the mean absolute error between the actual and expected false positive rates for each trial. See below for a series of graphs comparing the experimental false positive rate to the size of the filter, along with the theoretical false positive rate.

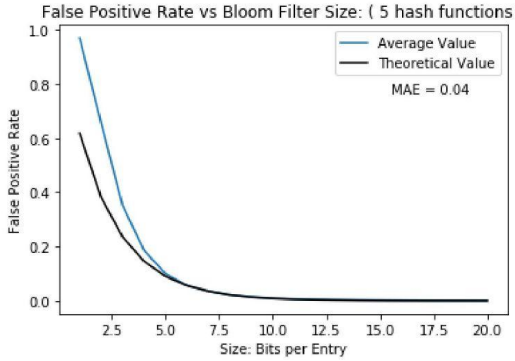

False Positive Rate vs Bloom Filter Size: (1 hash functions)
Average Value
Theoretical Value
MAE = 0.067
False Positive Rate
Size: Bits per Entry

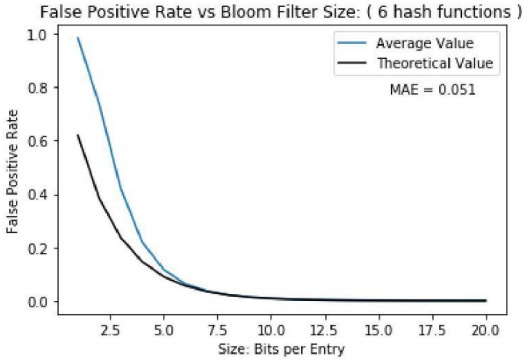

False Positive Rate vs Bloom Filter Size: (2 hash functions)
Average Value
Theoretical Value
MAE = 0.023
False Positive Rate
Size: Bits per Entry

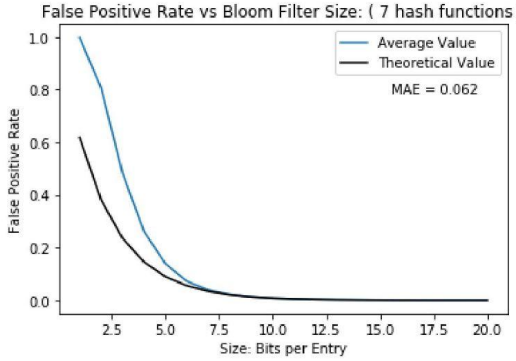

False Positive Rate vs Bloom Filter Size: (3 hash functions)
Average Value
Theoretical Value
MAE = 0.022
False Positive Rate
Size: Bits per Entry

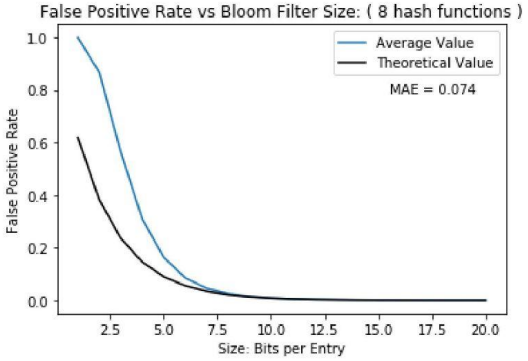

False Positive Rate vs Bloom Filter Size: (4 hash functions)
Average Value
Theoretical Value
MAE = 0.03
False Positive Rate
Size: Bits per Entry

\


False Positive Rate vs Bloom Filter Size: (5 hash functions)
Average Value
Theoretical Value
MAE = 0.04
False Positive Rate
Size: Bits per Entry


False Positive Rate vs Bloom Filter Size: (6 hash functions)
Average Value
Theoretical Value
MAE = 0.051
False Positive Rate
Size: Bits per Entry


False Positive Rate vs Bloom Filter Size: (7 hash functions)
Average Value
Theoretical Value
MAE = 0.062
False Positive Rate
Size: Bits per Entry


False Positive Rate vs Bloom Filter Size: (8 hash functions)
Average Value
Theoretical Value
MAE = 0.074
False Positive Rate
Size: Bits per Entry

Discussion-

Our results agree with the expected relationship between bloom filter size and the false positive rate. The graphs for each trial demonstrate exponentially decreasing false positive rates as the bloom filter size increases. However, the number of hash used affects the degree to which the empirical results match the expected values. For example, with only one hash function, the bloom filters from the datasets have significantly higher false positive rates than expected, especially for bloom filter sizes greater than 4 bits per entry. This is also reflected in the mean absolute error of .068 between the actual and expected values when using one hash function. When the number of hash functions is increased to 3, the empirical false positive rates most closely match the expected values. In this example, the graphs are extremely similar, with some discrepancies in the false positive rates for smaller bloom filters sizes. Furthermore, the mean absolute error for 3 hash functions was only .022. When the number of hash functions increases beyond 3, the expected and empirical false positive rates begin to diverge. For example, with 8 hash functions, the actual and expected values diverge greatly for bloom filter sizes less than 6 bits per entry. However, as the bloom filter size increases from 6 bits per entry, both graphs converge to zero. Lastly, the 8 hash function trial shows the greatest divergence between the actual and expected false positive rates, with a mean absolute error of .074.

Overall, by increasing the size of the bloom filter from 1 bit per entry, to 12 bits per entry, we observed dramatic improvements in the false positive rate. With a bloom filter size of 12 bits per entry, we recorded false positive rates of 1% or less, as long as we used between 3-8 hash functions. However, the impact of increasing the bloom filter size beyond 12 bits was very marginal. Therefore, since we want to achieve low false positive rates and high space efficiency, 12 bits per entry is the optimal size bloom filter to represent the UCI Bag of Words archive.

Count-Min Sketch

Testing Process -

The count min sketch was tested against the exact support algorithm, and the average absolute difference between the estimated and the true counts was measured under various values of epsilon to see the effect of the length of array on estimations. Also, we implemented our own polynomial rolling hash function generation algorithm. A delta of 0.1 was used - the equivalent of three pairwise independent hash functions. Testing was done on the accidents dataset.

Discussion -

Epsilon directly impacts the length of each array. The longer the arrays, the lower the chance of having collisions, which therefore reduces the probability of overestimations. Delta impacts the number of hash functions, which does not vary that much and has less of an impact on counting. However, very low deltas are discouraged because they directly impact the runtime as a result of increased hash functions. Epsilon is therefore the dominant parameter in count min sketches.

Average Difference(Delta = 0.1)

Epsilon	Average Difference(mmh3)	Average Difference(poly rolling hash)
0.05(3 by 55)	72014.64957264958	64423.11965811966
0.005(3 by 544)	232.25	434.38247863247864
0.0005(3 by 5437)	0.0	0.0

The two hash functions are also affected differently under different values of epsilon, but they yield more accurate results as the value of epsilon decreases, because of increased lengths of the arrays used which reduce collisions. Count min sketches save space and do not depend on the number of items present in the dataset - they utilize fixed space.

Cuckoo Filter

Implementation -

We implemented the Cuckoo Filter in a python jupyter notebook and we have implementations of a standard cuckoo filter as well as a Cuckoo Filter with multiple entries allowed in each bucket, which we called the Extended Cuckoo Filter. The Cuckoo Filter uses three hash functions: one to create a fingerprint of *n* bits, one to find the first potential bucket for an element, and a third to find the alternate bucket from the first bucket for any element. We used the mmh3 library for these three hash functions. At the time of construction, the size of the filter, number of entries allowed in each bucket, and size of each fingerprint (which has a default value of 7) are specified. The filter size and number of entries per bucket (for non-extended Cuckoo Filter this value is 1 making it a 1D array) are used to make a 2D numpy array to store entries and the fingerprint size is used to find the mod value to ensure all our integer fingerprints are stored as n-bit ints (i.e. a 7-bit fingerprint can be at most 127 so we do [hash_f(x)%128] to get a 7-bit fingerprint for x). Also, in these arrays we chose to represent empty buckets as numpy NaN values. In the jupyter notebook, the two class definitions are defined at the top followed by the code to run all the experiments with quick descriptions of each test.

Testing Process -

For the Cuckoo Filter, we conducted similar set membership experiments as with the Bloom Filter. We evaluated the false positive rate vs. size of the filter measured as bits per entry and compared that to the theoretical false positive rate for various filter sizes. To test this we created two text files of words randomly selected from a larger dataset of words (100,000 words). We created these files by selecting a random sample of 20% of the items in the text dataset which were added to the filter. The remaining 80% of items not present in the filter were

used to conduct set membership queries to observe the false positive rate (by analyzing if the Cuckoo Filter returned true when querying for an item not present in the filter). We also tested the Cuckoo Filter on a larger (500,000) dataset of integers to test other aspects of the Cuckoo Filter performance and confirm the results of the text dataset. This dataset was made by randomly sending integers between 0 and 1,000,000 to an insert.txt file with probability 0.2 and the rest to a lookup.txt file. The integers in the insert.txt file were inserted to the Cuckoo Filter and the integers in the lookup.txt file were again used as set membership queries to test false positive rate. This was the same process as the text file just with different inputs.

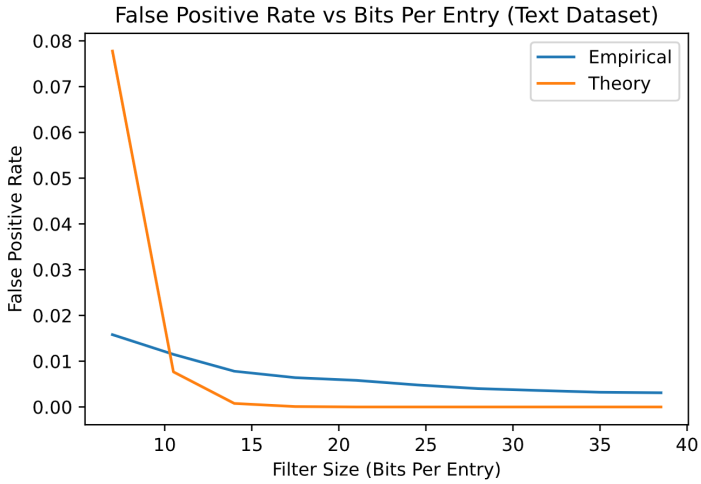
Beyond testing false positives vs bits per entry to compare to the Bloom Filter, we also tested the influence of fingerprint size and number of entries per bucket on the false positive rate. Our final tests tested how long it took to construct the filter and insert all the integers from the integer dataset. We decided to run this final test on time since it took a decent amount of time to run our false positive tests, and since data structures for data streams take on massive input loads, runtime is an important consideration.

Discussion -

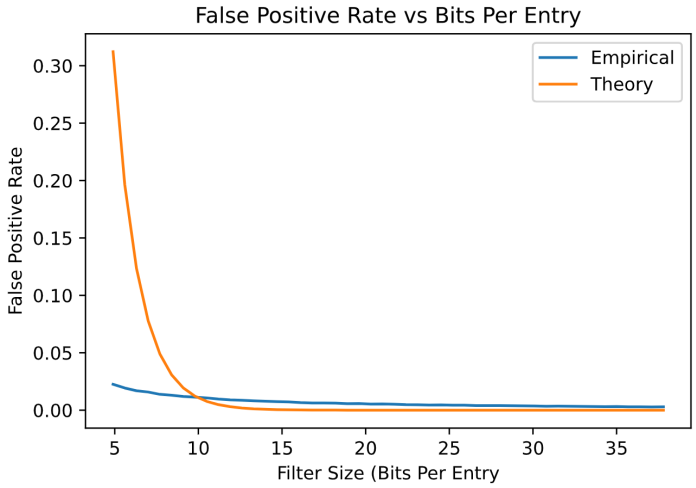
In theory the Cuckoo Filter has better space efficiency for similar performance when compared to the bloom filter. It achieves this by its addition of n-bit fingerprints rather than using a single bit such that if two elements are mapped to the same bucket, they must also have the same fingerprint to be counted as a false positive. Our results reflected this as the Cuckoo Filter performed better than the Bloom Filter for similar filter sizes (measured as bits per entry). Measuring bits per entry is different in the Cuckoo Filter compared to the Bloom Filter. Since each bucket in a Cuckoo filter holds a n-bit fingerprint rather than just a bit and we have a specified load factor, the actual bit per entry cost can be calculated by:

$$C = \frac{\text{table size}}{\#\text{ of items}} = \frac{f \cdot (\#\text{ of entries})}{\alpha \cdot (\#\text{ of entries})} = \frac{f}{\alpha} \quad \text{bits.}$$

where f is the size of the fingerprint and α is the load factor ($\alpha = \frac{\#\ of\ possible\ entries}{\#\ of\ buckets}$). By changing the number of buckets in the Cuckoo Filter and testing the false positive rates we were able to calculate the false positive rate vs bit per entry for direct comparison to the Bloom Filter. Here is a plot of our results:



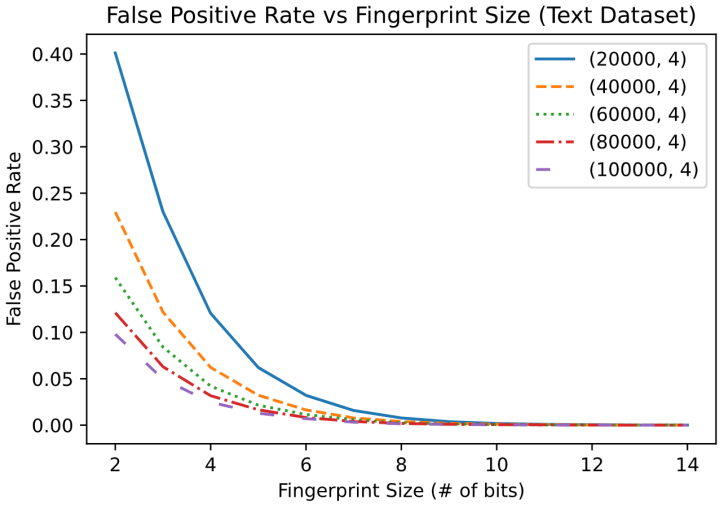
In our test the Cuckoo Filter obtained a false positive rate of about 0.015 at 7 bits per entry whereas the best bloom filter had a performance of about 0.05 at 7.5 bits per entry. To confirm these results we further tested this performance on the larger dataset of integers. The plot of those results are shown here:



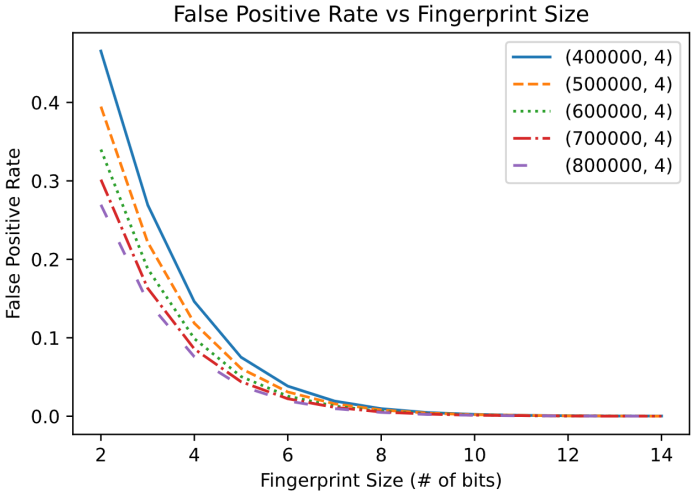
Here we got a very similar performance, even though it did perform slightly worse than the text dataset. However, it still gives a false positive rate of only about 0.03 at 7 bits per entry which is very good and better than the Bloom Filter. It is interesting to note that the empirical value matches the theoretical at 10 bits and greater but performs much better than the theoretical at low bits per entry sizes. The paper where the Cuckoo Filter was initially presented, and other discussion online, stated that the Cuckoo Filter often performs better than the theoretical value at high occupancy (i.e. small bits per entry values) which is exactly what we see here. However, strong conclusions cannot be entirely certain as these are tests on a single dataset over a small number of runs. In an ideal world, more tests would have been conducted but due to figuring out bugs in the code and a hard deadline, only a limited amount of testing was able to be done.

Overall, we can say that the Cuckoo Filter performs better bit for bit than the Bloom Filter in terms of false positive rate on lookups.

Now looking beyond the false positive rate we also individually tested the Cuckoo Filter against itself to see its performance as its parameters are changed. We found, unsurprisingly, that the performance of the Cuckoo Filter gets significantly better as the size of the fingerprint increases. However, the performance increase starts to flatten out after an 8-bit fingerprint. Our results are visualized here:



The effect of fingerprint size was tested over filters with different filter sizes in terms of numbers of buckets, but they each had 4 entries per bucket. Every filter size gets very close to zero after 10-bit fingerprints and all except the 20,000 bucket filter achieved very good false positive rates after 8-bit fingerprints. This analysis is important as bits per entry is directly proportional to the size of the fingerprint so choosing the smallest possible fingerprint while still remaining less than a desired false positive rate is important in achieving space optimization. The results are similar when tested on the integers:



Again we see that the performance increase beyond 8-bits is minimal. This agrees with published results that suggest using fingerprints between 7-10 bits. Also, smaller fingerprints performed better on the larger dataset than the smaller dataset so a future question to answer could be whether optimal fingerprint size decreases as filter size increases or if it is independent and this is just a coincidence arising from comparing only two tests.

More tests were also performed as described before. Our plots are in the jupyter notebook with quick descriptions and will not be presented here as they are not directly a part of our comparisons, however we will quickly note our findings. We found that the number of entries per bucket has a minimal effect on the false positive rate and it both increased and decreased the false positive rate. The rate could possibly increase because there are more entries at each bucket so it is more likely that two entries with the same fingerprint could end up at the same bucket before relocations need to occur. It also had a greater effect when the fingerprint size was smaller which is likely due to the fact that the alternate bucket for any element is calculated based on the fingerprint so for each bucket there are only 2^n(where n is the fingerprint size) possible alternate buckets. So smaller fingerprints result in more collisions and potential lookup confusion. We then tested the time to insert elements into the filter. We found that increasing both filter size and number of entries per bucket dramatically decreased the time it took to insert the 500,000 elements from the integer dataset into the fitler. The entries per bucket increase had more of an effect than filter size, however, as it has more of an influence on the number of relocations necessary when inserting, which is the most costly aspect of insertion.

In the end, we found that creating a Cuckoo Filter with fingerprints of size 7 with 4 entries per bucket will provide excellent performance when measured by its false positive rate, and that it can achieve this performance with more space efficiency than the bloom filter. This comes at a cost of complexity in terms of code and if you surpass your desired load factor, insertions will begin to take a lot of time as many relocations will need to occu, which may also lead to increased failed insertions. It also should be noted that the Cuckoo Filter also provides deletion as a property which is not possible with the Bloom filter. However, when the data stream becomes dynamic and deletions occur, the filter can begin to perform more poorly if the stream frequently adds duplicates. To maintain the no false negative property, multiple duplicate fingerprints may need to be added at the same bucket to ensure that if two elements have the same fingerprint at the same bucket and one is deleted, it doesn't also delete the other element from the filter. This ultimately begins increasing false positives, for example if you insert the same element twice and delete it once it would then still be in the filter. We unfortunately did not get to do anything more than play around with this feature but it is still an improvement over the Bloom Filter for data streams where deletions are desired. And it is also an improvement over the Bloom Filter when deletions aren't desired.

Next Steps/Acknowledgments:

While we hoped to implement a counting version of either the Bloom Filter (Counting Bloom Filter) or Cuckoo Filter (Counting Cuckoo Filter) in order to compare with the

Count-Min Sketch, we ran out of time to conduct analysis and were unable to finish said implementations. This would've made our analysis of the Count-Min Sketch more robust, and would've been a top priority if we'd had more time. On a final note, we would like to thank Matteo and TAs Kaitlin and Scott for their help throughout the project process.