Project Plan Assignment
Group 2 - Data Structures for Data Streams
Maggie Drew, Dennis Mbae, Daniel Feldman, Adam Gibbs
24 March 2021

Data Mining Task

As data streams form dynamic datasets and become increasingly analyzed in lieu of static datasets, resource constraints due to lack of easily accessible storage and associated time costs necessitate the use of efficient data structures to process data streams in real time. 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. Bloom filters utilize independent hash functions that hash to a bit array, and are used to determine set membership of discrete elements. Removing duplicates from a data stream, filtering prohibited queries such as removing spammer email addresses in an email stream, and keeping track of if a user has visited a webpage are all potential applications of bloom filters. Count-Min sketches similarly use independent hash functions, but instead map each hash function to a unique single array where the entry mapped to by the hash function is then incremented by one when an item passes through the filter. Count-Min sketches can process count-based queries for discrete elements, and can be used to determine quantiles of data streams and frequent items/"heavy hitters". Cuckoo filters extend the functionality of bloom filters by using a cuckoo hash function (in which a fingerprint of an inserted item is made prior to insertion into a space in the hash table) instead of a standard independent hash function, which enables efficient lookups and deletions for elements. Also used for set membership queries and with limited discrete element counting capabilities, Cuckoo filters can easily be applied to similar cases as Bloom filters (database queries, removing duplicates in a data stream), and with added deletion capabilities.

Data Structures

*Bloom Filter**:***

A bloom filter is used for set-membership queries (determine if an item is already present without storing the item, therefore saving space). A single item is passed through a number of various **independent** hash functions that map uniformly an integer from 0 to m-1 in a bit array of length m. The bit array entries are initialized as 0. Once an entry is mapped, it is set to 1. To search if an item is present, all the bits mapped by the hash functions should be 1. Multiple items can share partial or full entries in the bit array therefore **deletion is not possible**. Also because items can share bit array entries, there is a possibility of **false positives** on items not present on

the dataset. Therefore, the bloom filter is referred to as a probabilistic data structure. There are other bloom filters in literature that improve upon the standard bloom filter and support counting (counting bloom filters), deletions (d-left counting bloom filters and quotient filters), and performance optimizations (blocked bloom filters).

*Count-Min Sketch**:***

A count min sketch is used to estimate the count of an item. It consists of a set of w different numeric arrays each of length m. Each array is associated with an independent hash function(pairwise independent) that maps from 0 to m-1. An item is passed through each hash function. The entry that is mapped by this hash function is incremented by 1. Similar to bloom filters, various items can share these bit entries, therefore to get a lower bound on the count of an item, the minimum value of all the entries it gets mapped to is taken. This is also the motivation to use many bit arrays instead of one - to reduce collisions that may inflate the count for a singular item (Items may share a value in one bit array but the chance that they share the same entries in all arrays is relatively lower). It is also probabilistic in nature.

*Cuckoo Filter**:***

The Cuckoo Filter is another algorithm to approximate set membership in a data stream. It is similar to the Bloom Filter, however, the implementation of the Cuckoo filter provides the following improvements over the bloom filter: deletion, better lookup performance, and it usually uses less space. The Cuckoo filter makes these improvements by using a cuckoo hash rather than the standard hash function used for the bloom filter. 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 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 probably a member of the set, the cuckoo is another probabilistic data structure so we can only place error bounds on whether an item is in the set. However, there cannot be any false negatives. For deletions, the item is looked up and if it is found then it is removed. The deletion attribute of the cuckoo filter makes it a better candidate for multiple tasks and the small number of hash functions makes looking up values quicker. Further, the amortized time needed to insert a value is still O(1), so it is not slower than the traditional bloom filter. In terms of variants, the cuckoo filter itself can be altered by changing the number of buckets and the number of fingerprints per bucket, and there are studies showing the optimal sizes for both depending on the data. Other than that, there are some space efficiency upgrades for the cuckoo filter also present in literature

Evaluation and Experimentation

After implementing the bloom filter, cuckoo filter, and the count-min sketch we will perform some tests to analyze their performance under different conditions. The bloom and cuckoo filter are set-membership queries and they ensure no false negatives but can result in some false positives. They are also variable in size as their purpose is to make massive domains manageable. So here we will analyze the relationship between filter size and frequency of false positives. Since the bloom and cuckoo filters are simply set-membership queries we will also analyze the count-min sketch which counts the number of times an item has occurred. Again here we have no false negatives but false positives are technically possible. But more importantly we want to test the accuracy of estimating the support of each item in the dataset by the count-min sketch data structure. The datasets we are using will be specified in the next section of the project plan and the datasets will be abstractly referred to in this section.

False Positivity vs Size:

For assessing the Bloom and Cuckoo Filter we would like to analyze the tradeoff between size and accuracy of the filter data structures. As the size of the structure decreases, the probability for hash collisions increases, and therefore the likelihood for false positives must also increase. We can quantify this relationship by comparing the false positive rate (ϵ) to the size of the filter (bits per filter entry):

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

To test this, we will take one of our datasets and create a random subset of the data. We will then query our data structures with that random subset and at the end of the subset we will lookup all the items that were in the dataset but not in the random subset. Here we can count the number of false positives by counting the number of positive lookups on items we know were not provided to the data structure. We will test multiple different sizes of both the bloom and cuckoo filter and compare our experimental results to the theoretical optimal values for each data structure. We will then compare the false positivity rates per size between the two data structures to see the improvements between the two.

Support/Frequency Estimation Accuracy:

For the count-min sketch we will run the algorithm on multiple datasets, both sparse and dense. We will also run the exact algorithm for finding the support of items in a dataset as we did in our implementations of the frequent items homework assignment. Here we can compare the experimental and exact number of occurrences of each item. If time permits, we would implement an extension of the count-min sketch such as the count-median-min sketch and compare the accuracy time trade off of those two algorithms. We would do this by comparing experimental to exact supports of items while recording the time taken to run the algorithm. We could then plot accuracy vs time to visualize the trade off and see if either performs significantly better at a certain size of dataset (with the assumption that larger datasets take more time). Our intentions are to do this test on multiple datasets, with varying densities to see how each performs on different types of data.

Datasets

- **List of words taken from NY Times articles**: this is a massive .txt file of words taken from NY Times articles over the years. This will be used as is for the set-membership tests. It is large enough that we can take many different random samples to use for our false-positivity tests. We can also manipulate it by replicating different random samples random amounts of times and using it to test the count-min sketch. This will be a sparse dataset as most words will only occur once.
    - https://archive.ics.uci.edu/ml/datasets/Bag+of+Words
- **Webdocs, Kosarak, Accidents from FIMI:** these datasets from the first assignment will be good datasets to test our filters on. We will use their initial size to ensure our implementations are correct and then replicate them to make their sizes much larger. We will read in each line and each item on each line once to simulate a data stream. Their repetition will also make these datasets good for both the set-membership queries and the support estimation for count-min sketch.
    - http://fimi.uantwerpen.be/data/

Work Breakdown

For each of the data structures listed above, we intend to use the following plan -

1. Prepare datasets for our project
2. Create smaller samples of datasets for initial easy testing
3. Architect the data structure on paper/pseudocode
    a. Determine reasonable hash functions for datasets.
4. Implement algorithm in code
5. Test on smaller sample dataset to ensure accuracy of implementation
6. Perform tests on full datasets
7. Investigate results (e.g. false positive rate)
8. Formalize results and methods into a report of our findings

9. Extend/improve the data structure (if time permits)
    a. Bloom filter extensions
        i. Counting Bloom filter, Block Bloom filter
    b. Count-Min sketch
        i. Count-Mean-Min sketch
    c. Cuckoo filter extensions
        i. Counting Cuckoo filter, Adaptive Cuckoo filter

Logistics Information

_____Our plan is to collaborate when writing the code for these implementations and to do so we will be using GitHub to share code and progress, as well as having a centralized backup. In the repository we will have implementations of each algorithm in python along with a description of each algorithm in a separate .txt file. We will also have the code used to run our tests along with a .txt file explaining where we got our datasets from and any adjustments we made to them. We will add you (Matteo) as a contributor to the GitHub repository and for each phase submission we will submit all code by pushing a commit "Phase x delivered" and sending an email (or moodle submission, whatever you choose) with a link to the repository and all reports in PDF format.

Phase 3 deliverables:

- Working bloom filter implementation
- Working cuckoo filter implementation
- Working count-min sketch implementation
- Code to test false positivity rate
- Code to find exact support of items in a dataset

Phase 4 deliverables:

- Results from analysis of false positivity rate vs filter size for bloom and cuckoo filters
- Results from comparison of false positivity rates of bloom and cuckoo filters
- Results from analysis of the accuracy of count-min sketch
- Video of our project including:
    - Our algorithms and their purpose (high level description)
    - What we tested about the algorithms
    - How we tested the algorithms
    - Summary of results and whether they support the claims from the literature

Resources

Aggarwal, Charu C. Data mining: the textbook. Cham[Switzerland]; New York: Springer, 2015.

Cormode, Graham. “Count-Min Sketch”. http://dimacs.rutgers.edu/~graham/pubs/papers/cmencyc.pdf.

Cormode, Graham and Muthukrishnan, S. “An Improved Data Structure Summary: The Count-Min Sketch and Its Applications”. https://dsf.berkeley.edu/cs286/papers/countmin-latin2004.pdf.

Fan, Bit et. al. “Cuckoo Filter: Practically Better Than Bloom”. https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf.

Li, Shangsen et. al. “Multiset Synchronization with Counting Cuckoo Filters” https://arxiv.org/pdf/2003.03801.pdf