COMP 480/580 — Probabilistic Algorithms and Data Structures

October 1, 2024

Lecturer:
Anshumali Shrivastava

Scribe By:
Abhishek Sharma (as537), Chandana Mohan (cm203), Prashanth Javaji(pj21), Wenhan Lee (sl305)

1 Heavy Hitters Problem

The heavy hitters problem involves finding elements in a data stream that occur most frequently, which poses a challenge when the stream size is too large to store entirely. The goal is to identify these frequent elements efficiently, often using streaming algorithms that save memory by estimating frequencies rather than storing the entire stream.

Probabilistic data structures and algorithms are well-suited to tackle this problem efficiently, using limited memory and processing time while allowing a small margin of error.

Some widely used techniques include:

- **Count-Min Sketch:** This data structure provides approximate frequency counts of elements in the stream, using hash functions to update and estimate counts in constant space.
- **Misra-Gries Algorithm:** This is an approximate algorithm that maintains a list of potential heavy hitters with a fixed memory size, ensuring efficient identification of frequently occurring items.
- **Space-Saving Algorithm:** Similar to Misra-Gries, it maintains a fixed-size summary of elements to approximate the most frequent items.

These algorithms use hashing and counters to estimate frequencies in sub-linear space, making them highly effective for detecting heavy hitters in massive data streams where exact tracking would be impractical.

Let's consider **Twitter** as an example to illustrate the Heavy Hitters Problem. Imagine that Twitter wants to identify the most trending hashtags in real time. With millions of tweets being generated every minute, it's impractical to store and analyze every hashtag in detail due to memory and computational constraints.

For each tweet, assume that we want to extract the 4 most contiguous words (known as 4-grams) and store them in a dictionary. The dictionary keys would represent all possible combinations of these four-word sequences, with a counter to track their frequency. As new tweets are processed, the corresponding count for each 4-gram would be incremented whenever it is encountered.

This naive approach is simple but presents significant challenges. Given a large vocabulary, the number of possible 4-grams results in an enormous number of combinations. Storing counts for all these possible phrases is unrealistic due to the vast memory requirements. Even if we tried to optimize by selectively analyzing 4-grams, the memory needed would still be extremely high, making the solution impractical for real-time processing of a continuous stream of tweets.

To efficiently solve the heavy hitters problem on Twitter, a more advanced, memory-efficient streaming algorithm, such as Count-Min Sketch, is required. This method uses probabilistic data structures and multiple hash functions to estimate the frequency of phrases without needing to store all possible combinations explicitly. This allows for detecting trending topics in real-time with feasible memory usage.

1.1 Applications of the Heavy Hitter Problem

The Heavy Hitters Problem has numerous real-world applications across different domains, where identifying the most frequent items is crucial. Here are some notable examples:

- **Network Traffic Analysis**

  In network monitoring, heavy hitters refer to IP addresses or data flows generating a significant amount of traffic. Identifying these addresses can help in detecting DDoS attacks, malicious behavior, or network congestion issues. Algorithms like Count-Min Sketch can be used for real-time monitoring of packet data.

- **Database and Query Optimization**

  Database Management Systems (DBMS) can use heavy hitter algorithms to identify the most frequently accessed rows, columns, or queries. This can be useful for caching, indexing optimization, and resource allocation to improve query performance.

- **Social Media Trend Analysis**

  In social media, platforms like Instagram, Facebook, or Twitter can use heavy hitter algorithms to identify trending hashtags, keywords, or even frequently shared videos and images. This helps in displaying trending content and promoting popular topics.

- **Natural Language Processing (NLP)**

  In NLP, heavy hitters may refer to the most frequently occurring words or phrases in a corpus of text, which can be useful for identifying stop words, building dictionaries, or improving the efficiency of text processing.

- **Sensor Networks and IoT** In Internet of Things (IoT) systems, heavy hitters could be the most frequently reported data points by sensors, such as temperature spikes or unusual activity from devices. This information can trigger alerts for monitoring industrial equipment or environmental factors.

2 Majority Element Problem

2.1 Ideation

The Heavy Hitters Problem is characterized by the fact that no algorithm can solve it for all inputs in a single pass while using sublinear auxiliary space. However, when the objective is to find the most frequent element in a data stream—specifically, one that makes up more than half the elements, there is a viable solution that can do this in one pass with sublinear space. This special case is known as the Majority Element Problem.

2.2 Problem Statement

Given an array A of length n with a guaranteed majority element that appears more than $n/2$ times, find the majority element.

2.3 Majority Element Algorithm

```
for i = 0 to n-1
{
    if i == 0
    {
        current = A[i];
        currentcount = 1;
    } else
    {
        if (current == A[i])
        {
            currentcount++;
        }
        else
        {
            currentcount--;
            if (currentcount == 0)
            {
                current = A[i];
                }
            }
        }
    }
```

2.4 Analysis of The Majority Element Algorithm

Algorithm Explanation: The algorithm operates by iterating through the array and maintaining a candidate for the majority element along with a count that tracks how many times it has been seen. Whenever the current element matches the candidate, the count increases; if it differs, the count decreases. When the count reaches zero, it indicates that the current candidate has lost all its support, meaning there are more different elements than the current candidate in the seen elements so far. At this point, the algorithm updates the candidate to the current element, effectively discarding the previous candidate.

Algorithm Correctness: The correctness of the algorithm is based on the guarantee that a majority element exists. As we process the array, any candidate that does not have enough support will eventually be discarded when its count reaches zero. Given that the majority element appears more than $\frac{n}{2}$ times, it will always have enough occurrences to outlast other candidates. Thus, at the end of the iteration, the remaining candidate will be the majority element, fulfilling the problem's requirements.

Running Time: This algorithm has a time complexity of $O(n)$ since it finds a solution in a single pass over the array A of length n.

Space Complexity: This algorithm uses $O(1)$ storage, utilizing only a single counter to store the count and a variable to track the most frequent element.

2.5 Problems with the Generalization of the Majority Element Method

General Case: This method does not work for arbitrary cases, as it only applies specifically when the majority element appears more than $n/2$ times in a data stream, which is not very common in real life.

Conditional Success: If the stream contains elements with large counts, finding the majority becomes feasible, as discussed later in Section 3.

2.6 Power Law in the Real World

Data in the real world often follows a power law distribution. This means that a small number of items are very common, while the majority are rare. An image representing this can be seen below. The elements in green have a higher frequency than the elements in yellow.

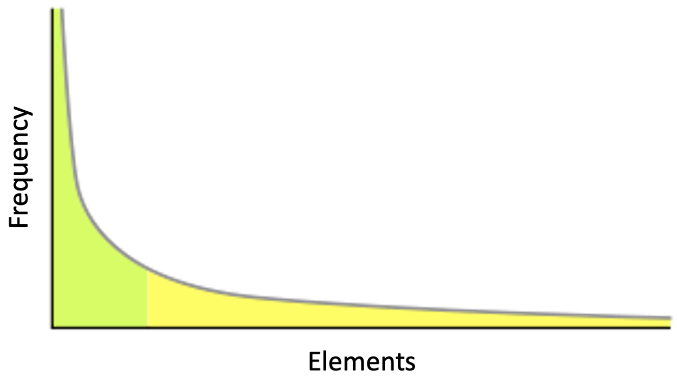

Figure 1: Power Law Visualization

Since we typically have a few items that are very common and many that are uncommon we can use this fact to derive a solution to the Heavy Hitters Problem.

3 Counting Bloom Filters (CBF)

Instead of bit arrays which can only store a value of 0 or 1, we use an array of length R to store counts.

3.1 How Does it Work?

- **Insertion:** Whenever an element s is added to the hash table, increment $h(s)$ by 1.
- **Querying:** To query element q (i.e. estimate count of its occurrences), simply return $h(q)$.
- Regarding a universal hash function, the probability of $Pr(h(s) = c) = \frac{1}{R}$.

3.2 Collisions

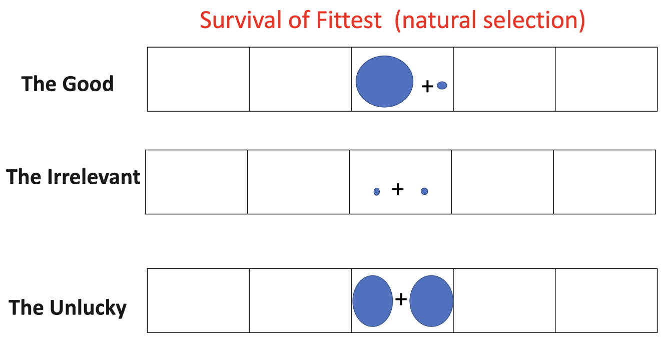

Figure 2: CBF Collision Scenarios

- Collisions always cause overestimation. Assuming that we only care about heavy-hitting objects, the cases we consider favorable are the ones in which $h(a)$ is fairly accurate for high frequency element a.
- **"Good" Case:** Heavy hitting (high frequency) element a collides with rare element b: $h(a)$ is fairly accurate, but $h(b)$ is very inaccurate
- **"Irrelevant" Case:** Rare element a collides with rare element b: $h(a)$ and $h(b)$ are both fairly accurate
- **"Unlucky" Case:** Heavy hitting element a collides with heavy hitting element b: $h(a)$ and $h(b)$ are both inaccurate

3.3 Error Analysis

- Let's say, c_s = count of element s, $\sum$ = sum of all c_s, and R = length of array.
- For each element s, every other element has a $\frac{1}{R}$ probability to collide with s
- Expected value of $h(s) = c_s + \frac{1}{R}(\sum - c_s) < c_s + \frac{\sum}{R}$.
- Therefore, the expected error is $< \frac{\sum}{R}$

3.4 Problems and Limitations

- **Collision Problem:** In CBF, if two or more elements collide in the hash table, their counts are overestimated. This becomes problematic, especially for heavy hitters (frequent elements).
- **Large Overestimates:** When elements with large counts collide, their estimated frequency becomes much higher than their true frequency.

4 Count-Min Sketch

Count-Min Sketch is a probabilistic data structure and is a simple technique used to summarize large amounts of frequency data. The count-min sketch algorithm keeps track of the count of things. i.e, How many times an element is present in the set. It addresses the issue of collisions in Bloom filters by utilizing multiple hash functions and tables.

4.1 Algorithm

Count-Min Sketch Structure

- Uses d independent hash functions: $(h_1, h_2, \ldots, h_d)$.
- Maintain d separate hash tables: $(A_1, A_2, \ldots, A_d)$ each with size R.

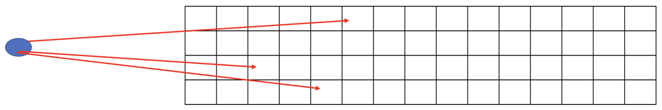

Figure 3: Count-Min Sketch

Updating Counts

- When a string s is seen:
  - Increase counters:
$$A_1[h_1(s)], A_2[h_2(s)], \ldots, A_d[h_d(s)]$$
- This takes $O(d)$ time.

Estimating Counts:

- To find the count for string s:
  - Look at the counters and pick the minimal count value:
$$\text{Estimate}(s) = \min(A_1[h_1(s)], A_2[h_2(s)], \ldots, A_d[h_d(s)])$$
- This takes $O(d)$ time.

4.2 Why this Works

- Every counter in CMS is either an overestimate (due to collisions) or the exact count for the element.
- By taking the minimum of multiple overestimates, CMS returns the count with the smallest overestimate thus giving the most accurate count estimate.
- It is unlikely that all counters will have collision at the same time.
- In a single universe (hash table), collisions between heavy hitters are rare. Across multiple universes (multiple hash functions), it's even rarer for the same set of elements to collide.

4.3 Error Analysis

- When using a Counting Bloom Filter with one hash function, each item s contributes to one counter based on the hash value.
- Due to potential collisions with other items, the count of s can be overestimated.
- Let $\epsilon = \frac{1}{R}$, where R is the size of the hash table.

- The expected error in counting is given by:
$$E(\text{err}) = \epsilon\Sigma$$
Here, Σ is the total count of all items.

Deriving the Probability Bound

Applying Markov's Inequality:

- Markov's inequality states that for any non-negative random variable X:
$$P(X \geq a) \leq \frac{E(X)}{a}$$
We will use this to bound the probability of the error being larger than a certain threshold.

Setting Up the Probability:

- Let us say we want to bound the probability of the error exceeding $2\epsilon\Sigma$:
$$P(\text{err} > 2\epsilon\Sigma)$$

Using Markov's Inequality:

- By applying Markov's inequality, we get:
$$P(\text{err} > 2\epsilon\Sigma) < \frac{E(\text{err})}{2\epsilon\Sigma}$$
- Substituting $E(\text{err}) = \epsilon\Sigma$:
$$P(\text{err} > 2\epsilon\Sigma) < \frac{\epsilon\Sigma}{2\epsilon\Sigma} = \frac{1}{2}$$

Considering d Hash Functions:

- When using multiple hash functions d, we can refine our estimates. Each additional hash function provides another independent estimate, which helps reduce the overall error.
- If we assume that using d hash functions maintains similar bounds on individual error probabilities, the probability of significant overestimation decreases exponentially:
$$P(\text{error} > 2\epsilon\Sigma) < 0.5^d$$

4.4 Memory Analysis

We aim for the condition:
$$c_s < \hat{c}_s < c_s + 2\epsilon\Sigma$$
c_s denotes the true count of string $s, \hat{c}_s$ denotes the estimate of $c_s,$ and δ is the probability of error for a given string s.

Here, ϵ relates to the size of the hash table and Σ is the total count of all items processed. δ is the probability of error; it represents how likely we are to deviate from this bound.

Probability of Error

We know that:
$$P(\text{error} > 2\epsilon\Sigma) \leq 0.5^d$$
This means that the probability of a significant error (overestimation) is bounded by 0.5^d, where d is the number of hash functions used.

From the previous relation, if we have:
$$0.5^d < \delta$$

we can take logarithms:

$$d \log(0.5) < \log(\delta)$$

Since log(0.5) is negative, we rearrange this to:

$$d < -\log(\delta)$$

Since we have d hash functions and each hash function uses a hash table of size R, the total memory requirement can be expressed as

$$\text{Memory} = R \cdot d$$

Thus, substituting our expression for d:

$$\text{Memory} < R \cdot (-\log(\delta))$$

In Terms of ϵ and δ
If we consider R in relation to ϵ (where typically R is inversely proportional to ϵ):

$$R = (\frac{1}{\epsilon})$$

Therefore, the memory requirement can be expressed as:

$$\text{Memory} = O\left(\frac{1}{\epsilon} \log\left(\frac{1}{\delta}\right)\right)$$

5 How to Identify top-k Elements

Given the problem of identifying the top-k elements within a stream while maintaining efficient time and space complexity, we can use heaps to arrive at an optimal solution.

Min-Heap Creation:

- A min-heap is created with a maximum size of k.
- This heap tracks the k items with the highest estimated frequencies observed so far.

Processing New Elements:

- As the stream progresses and a new element is encountered:
  - Compare its frequency (obtained from Count-Min Sketch) with the frequencies of the top-k elements in the min-heap.

Updating the Min-Heap:

- If the frequency of the new element s is higher than the minimum frequency in the heap:
  - Replace the minimum element in the heap with the new element s.
  - Update the heap accordingly.

Time Complexity:

- Each update operation takes $O(\log(k))$ time.
- In the worst case, if d updates are performed (for d different elements), the total time complexity would be:

$$O(d \cdot \log(k))$$