School of Engineering and Applied Science (SEAS), Ahmedabad University

Probability and Stochastic Processes (MAT 277)

Special Assignment Report

Group Name: hp-a1-01-s1

Team Members:

- Manan Vadaliya AU2040264
- Harshrajsinh Vaghrola AU2040195
- Mitsu Sojitra AU2040157
- Pruthviraj Dodiya AU2040175
- Divyashree Jadeja AU2040261

Algorithm: Count Min Sketch

I. Team Activity Learning and Concept Map

For the concept map we collected information about what count mean sketch is and how it works, where it fails and its applications. We broke information into parts which are cohesive and self explanatory and made a whole trail of points connecting each other. So that when a third person reads it then that person can understand those basic information about count mean sketch algorithm. It was really fun making a concept map as we made progress making it, we all concepts related to the algorithm started to make a whole lot of sense. The course is about probabilistic data structure. The count–min sketch (CM sketch) is a probabilistic data structure used to count frequency of a stream of data. It was a really great experience learning the real time application of probabilistic models and how they work. We learnt many new entities like stream of data, stream sampling, sub-linear, heavy hitters, basic properties of probability. We encountered many information and attractive properties about the algorithm and how it is widely used in many places where there is streaming big data. The task also taught us how randomisation is getting applied here in the case of streaming data sets. These tasks were connected to our course topics and what we learned in class. It motivated us to revise concepts that had been done in class which helped us in exams. Special assignment tasks were also responsible for teaching the importance of communication, punctuality, team work etc which are the factors very much needed at this stage of career.

II. Background

The data structure's concepts have a long history. The sketch was first formalized in early 2003, with a technical report published in the DIMACS series in June 2003 (original technical report), followed by LATIN 2004 and the Journal of Algorithms. The sketch was inspired by work on tracking histograms on data streams, which was later influenced by work on discovering frequent items appearing in data streams from 2002 . The algorithm was created by Moses Charikar, Kevin Chen, and Martin Farach-Colton in order to speed up Alon, Matias, and Szegedy's AMS Sketch for approximating the frequency moments of streams in data. It was a replacement for a number of different sketch techniques, including the Count sketch and the AMS sketch. The goal was to provide a simple sketch data structure with a precise characterization of the dependence on the input parameters. The sketch has also been viewed as a realization of a counting Bloom filter which requires only limited independence randomness to show strong, provable guarantees. The simplicity of creating and probing the sketch has led to its wide use in disparate areas since its initial description.

Count–min sketches are effectively the same data structure as the counting Bloom filters, which were first introduced in 1998. A count–min sketch has a sublinear number of cells, which is connected to the sketch's target and approximation quality. whereas a counting Bloom filter is normally sized to match the number of elements in the set.

The concept is virtually identical to John Moody's Feature hashing algorithm, except it uses low-dependence hash functions, which makes it more practical. The median trick, rather than the mean, is used to combine many count sketches in order to maintain a high probability of success.The algorithm has evolved itself within the last few years. The algorithm has some properties which makes it different from other sketches. The basic version of the count–min sketch's purpose is to consume a stream of events one by one and count the frequency of the various categories of events in the stream.

Count Min Sketch used a variety of ways to compress data, and these ways often started to result in random errors in data. When you create a sketch, you can give it as much room as you desire. The sketch is more likely to deliver imprecise results if your set is too huge or if you don't allocate enough space. Size versus precision is a clear tradeoff, and it was up to researchers what to choose. They made different versions of CMS to overcome this. They Also Noticed that the fewer hash functions they take, there will be a high probability of collision. Therefore, they suggested that it is preferable to take more hash functions. From count min sketch word 'sketch' means 2-dimensional matrix. In matrix number of rows is equal to number of hash functions and number of columns depends on maximum number which hash function gives as output we have predefined size of sketch that's why it is called as sublinear.

The internal structure of a Count-Min Sketch is a table, similar to that of a Hash Table. However, while Hash Tables use a single hash function, Count-Min Sketches use multiple hash functions, one for each column. To calculate the frequency of a given element we need to pass that data to all hash functions. If one or more elements got the same hash values in that case value increases because of hash collision. Therefore researchers suggested using more hash functions for better results. They used distinct hash functions rather than just one. These hash functions should not be dependent on one other meaning it should be "pairwise independent". To update a count, they hashed item a with all d hash functions and then increment all indices obtained in this manner. Researchers only increment the cell once if two hash functions map to the same index.Unless they expanded the available space, all this accomplished raised the number of hash collisions for the time being. That was gone in the following time period. For the time being, researchers decided to proceed with the concept. By using different hash functions researchers made sure that they have distinct cells to look at if they want to count.The natural option is to take the lowest of all of these values. The cell having the fewest hash clashes with other cells will be this one.

III. Motivation

In the last few years, sketching techniques have advanced significantly. They're particularly well-suited to data streaming scenarios. Nowadays a lot of data is coming in and the sketch summary has to keep up with it and be updated on a regular basis in a concise and timely manner. Count min sketch is considered as the one which has also been combined with group testing to solve problems. The concept of algorithm is created in such a way that updates produced by each new piece of data is minimized. Regardless of the some drawbacks in current condition this algorithm choice is unique. they're easier to parallelize and faster to process.

Though sketches have the shortest history of all the methods for approximation query processing yet it have had the direct impact on practical systems thus far. Nonetheless, their versatility and power indicate that they will very certainly become a standard feature in the next generation of approximation query processors. They have undoubtedly had a substantial impact in a variety of specialized sectors that process massive amounts of structured data, particularly those that entail data streaming.

"Frequency based sketches" are used to summarize a dataset's observed frequency distribution. Accurate estimates of individual frequencies can be derived from these sketches. As a result, techniques to discover the estimated heavy hitters (things that account for a significant portion of the frequency mass) and quantiles have been developed (the median and its generalizations). Estimating (equi)join sizes between relations, self-join sizes, and range queries all utilize the same principles. These can be utilized as primitives in more sophisticated mining operations, as well as to extract the histogram representations of streaming data.

The term "streaming" has been popular in recent years to describe circumstances where the observer only has one chance to see the information as it "streams" past them. Many such transactions are watched every second in the processing of financial data streams (streams of stock quotes and orders), and a system must handle these as they are viewed, in real time, to support real-time data analysis and decision making. Typical streaming algorithms have helped to compress the large amount of data.

You wouldn't use Count-Min Sketches to track values that fluctuate up and down because they don't have the ability to reduce counts. Finally, Count-Min Sketches have the unique property of never underestimating the number of events — they can only overestimate the number of events.

For computing approximate counts, the Count-Min Sketch is a probabilistic data structure. It's useful when space constraints are a concern but precise results aren't required. The data structure is set up in such a way that you can freely trade off accuracy and storage space.

The relative frequencies of each item or character within a data stream determine the entropy of the stream. B. Laksminath and Ganguly demonstrated how to estimate this entropy to within relative error using Count-Min Sketches within a larger data structure based on additional hashing techniques.

The Count-Min sketch is a versatile data structure that is finding applications in Data Stream systems, as well as Sensor Networks, Matrix Algorithms, Computational Geometry, and Privacy-Preserving Computations, as seen by the variety of applications stated above. It's helpful to think of the structure as a basic primitive that can be used wherever approximate entries from high-dimensional vectors or multisets are required and one-sided error proportional to a small fraction of the total mass can be tolerated (much like a Bloom filter should be considered whenever a list or set is used and space is limited)With this in mind, the report shows more implementations of the algorithm in application section.

IV. Algorithm Description

There is limited space, limited time and we don't get the chance to look at the data once again.

The streaming platform should calculate the frequency or should run the query. Due to limited storage we go with sub-linear space and we perform all this operations in REAL-TIME. Now we select a small data set from the infinite stream of data to approximate the frequency of a particular value from the selected data set. As we are representing the whole infinite data stream by a compressed data set this generates the randomness and hence we might also get some amount of inaccuracy in the solution.

Although the approximation could be done through various data structures like Hashing and Sampling but we get more accurate approximation through Count-Min Sketch data structures as it uses multiple hash functions at a time which provides more accurate probability of success in the sub-linear space.

How Count-Min Sketch works ?

The sketch is 2-D matrix or array with dimensions $(w \times d)$ where,
d = number of Hash functions w = working array

All the hash functions are pairwise independent with each other. The full CMS data structure involves three operation Initializing, increment count and retrieving count.

Initially we start with the a zero matrix $(w \times d)$ and then perform upadtion in the matrix. To update a count, we then hash item **a** with all **d** hash functions, and subsequently increment all indices we got this way. In case two hash functions map to the same index, we only increment its cell once.

If we now want to retrieve a count, there are up to **d** different cells to look at. As the name suggests Count-Min Sketch we take the minimum value of all of these. This is going to be the cell which had the fewest hash conflicts with other cells. While this is the fundamental idea of the Count-Min Sketch.

Pseudo code for CMS:

Initialization - $\forall$**i** $\in$ **(1,...,d)**, **j** $\in$ **(1,...,w)**, **:count[i,j] = 0**

Increment count (of element a) - $\forall$**i** $\in$ **(1,...,d):count[i,**h_i**(a)] += 1**

Retrieve count (of element a) - **:**$(min)d(i = 1)$**count[i,**hi**(a)]**

Algorithm - 1 :

CountMinInit(w,d,p)

C[1,1]....C[d,w]$\longleftarrow$ 0;

for j$\longleftarrow$1 **to** d **do**

Pick a_j, b_j uniformly from [1 .... (p-1)];

N=0

Algorithm - 2 :

CountMinUpd(i,c)

N$\longleftarrow N + c$

for j$\longleftarrow$1 **to** d **do**

$h_j(j) = (a_j \times b_j \bmod p) \bmod w$;

C[j,$h_j(j)$] $\longleftarrow$ C[j,$h_j(j)$] + c;

Algorithm - 3 :

CountMinRetr(i)

e $\longleftarrow \infty$;

for j$\longleftarrow$1 **to** d **do**

$h_j(j) = (a_j \times b_j \bmod p) \bmod w$;

e $\longleftarrow min(e, C[j,h_j(j)])$;

return e

This is the full CMS data structure.

Example for better understanding:

Suppose we get a data stream :

Stream $\longrightarrow [A, B, K, A, A, K, S \cdots \infty]$

Stream counter is on 1st A.

Below are all the hash functions and their values.

	H1	H2	H3	H4
A:	1	6	3	1
B:	1	2	4	6
K:	3	4	1	6
S:	6	2	4	1

Pass A through all hash functions and update the sketch with those hash values.

Initialize the sketch with all zeroes entries.

H1	0	0	0	0	0	0	0
H2	0	0	0	0	0	0	0
H3	0	0	0	0	0	0	0
H4	0	0	0	0	0	0	0

$$H_1(A) = 1$$

$$H_2(A) = 6$$

$$H_3(A) = 3$$

$$H_4(A) = 1$$

So after seeing the value of A in H1 we increment the column 1 by '1'.

So after filling the sketch y going through the first letter we get the sketch ads follows:

H1	0	1	0	0	0	0	0
H2	0	0	0	0	0	0	1
H3	0	0	0	1	0	0	0
H4	0	1	0	0	0	0	0

Similarly increment the stream counter by 1, pass the data through all the hash function and then increment the value in the sketch by '1' in the respective output (of hash table) column.

So after successfully loading the frequency in the sketch, we get the sketch as follows:

H1	0	4	0	2	0	0	1
H2	0	0	2	0	2	0	3
H3	0	2	0	3	2	0	1
H4	0	4	0	0	0	0	3

To calculate the frequency of 'A':

Pass 'A' through all the hash functions.

$$H_1(A) = 1$$

$$H_2(A) = 6$$

$$H_3(A) = 3$$

$$H_4(A) = 1$$

For H_1 we get the hash function's output as 1 so in the sketch we go to the sketch at the same position and we get the value 4

Similarly for H_2, H_3 and H_4 we get the values 3,3 and 4 respectively.

$[4, 3, 3, 4]$

Now as the algorithm say 'CMS' we will count minimum of those counts. So the frequency of A is 3.

Now, there might be some case where we face collision. What if one or more element has same hash values and then they get incremented. So if there's collision with A we will get the count as 5,4,4,5 to avoid collision we can use more hash functions.

NOTE: CMS will never undercount.

V. Application

The Count-Min Sketch has been used in many different scenarios and has a wide range of applications. Here's a rundown of some of the ways it's been used or modified.

Compressed sensing:

The compressed sensing challenge entails creating a set of linear measurements in order to recover a sparse or skewed signal. Since its formalization in 2004, this subject has attracted the attention of researchers in CS and EE. Sketch data structures, such as the CM sketch, have been found to be useful in addressing compressed sensing issues, with a "decoding" stage that is more easier and faster than methods based on LP-solving.

Networking:

In recent years, network anomaly detection, as a method of detecting network security threats, has grown in importance in the field of information and communication technology (ICT). Anomaly detection is the detection of infrequent events or items in a data stream that differ considerably, raising suspicions. They're commonly used to detect network intrusions. Because of the processing overhead and memory requirements, network bandwidth is now too high, making it difficult to detect anomalies.
For years, various efficient data structures have been created to detect anomalies in data streams while maintaining guaranteed error boundaries.
This means that in a distributed paradigm like count min sketch can be valuable for large-scale data analysis. Each machine has the ability to create and emit a sketch of its own local data. The count min sketch of a potentially large collection of data can therefore be combined at a single machine (a single reducer just sums up the sketches by input). In terms of network connectivity (and thus time and other resources), this strategy can be far more efficient than obtaining accurate counts for each item and filtering out the low numbers.

Database:

The Count-Min Sketch algorithm is not only effective for data streams, but it is also very beneficial for databases, and it is utilized in a variety of database applications. Some of such applications will be covered in this section, particularly those that were introduced in the paper "Spectral Bloom Filters."

There is another bloom filter that was introduced before the count min sketch algorithm. The Spectral Bloom Filter (SBF) is a Bloom Filter (BF) data structure introduced two years prior to the Count-Min Sketch. It gives users count approximations in the same way that the Count-Min Sketch does, but in a different way. The differences between those two implementations will not be discussed in detail in this work. The key difference is that, like the conventional Bloom Filter, the Spectral Bloom Filter only has a single array.

Instead of bit fields like the BF, the SBF keeps counters in this array. In contrast, the Count-Min Sketch uses a different array for each hash function it applies. This is applied in the database.

Security:
Count min sketches that limit the quantity of data kept appear to be a suitable candidate for preserving data privacy. However, proving privacy necessitates further caution. The Count-Min sketch can be used to compute a sketch of a vector privately . The Count-Min sketch can be made pan-private, which means that information about the persons who contributed to the data structure is kept private.

The count-min sketch is a useful data structure for accurately recording and predicting the frequency of string occurrences in sub-linear space, such as passwords. It cannot, however, be used to infer conclusions about groups of strings that are similar, such as those with similar Hamming distances. The count-min sketch is modified in this study to enable predicting counts within a defined Hamming distance of the query string. This design, like the original drawing, can be used to restrict users from using popular passwords, but it also provides for a more efficient manner of analyzing password statistics.

NLP:
All of these issues are ultimately based on estimates of item counts (such as n-grams, word pairs and word-context pairs). As a result, we concentrate our efforts on resolving this essential issue in the context of NLP applications. Answering point queries with sketch methods saves both memory and time. In NLP, it was recently proved (Goyal and Daume III, 2011a) that a variation of the Count-Min sketch accurately solves three large-scale NLP issues with a modest constrained memory footprint.

However, there are a number of different drawing algorithms, and it's unclear why this one should be chosen above the others. In this paper, we examine and analyze a variety of sketch strategies for answering point inquiries, with a focus on large-scale NLP applications. While sketches have been examined for frequent item detection and join-size estimation in the database community, this is the first comparison research for NLP challenges. Three contributions are included in our work: By extending the concept of conservative update to existing sketches, we provide fresh versions. We propose the Count sketch with conservative update and the Count-mean-min sketch with conservative update.

Heavy hitters:
In the heavy hitters problem, the input is an array A of length n, and also a parameter k. You should think of n as very large (in the hundreds of millions, or billions), and k as modest (10, 100, or 1000). The goal is to compute the values that occur in the array at least n/k times.1 Note that there can be at most k such values; and there might be none. The problem of computing the majority element corresponds to the heavy hitters problem with k 2 for a small value ¿ 0, and with the additional promise that a majority element exists.

An IP address/port or a combination of both can be a network's heavy hitter. These heavy hitters generate high volumes of traffic that exceed a network's predefined threshold, resulting in abnormalities. The Heavy hitters problem can be summarized by examining an array of elements A with a length of n and a parameter of k. In this case, k is a small number (100 or 1000) in the case when all potential source and destination IP address pairs are considered, and n is enormously huge (billions or trillions). The goal is to locate the array members that appear at least n/k times. It's worth noting that there can only be k such elements in the array, or there may be none at all. The heavy hitter dilemma, on the other hand, guarantees the existence of a majority element. Heavy hitters, for example, are data packets that account for more than 15% of total network traffic and break the service agreement between the two nodes.

Due to processing and memory limits, monitoring heavy hitters in real time is

becoming increasingly difficult as the Internet grows in size and complexity. indicates that the system should scale up to at least 2104 keys (here keys are denoted as IP address/port) to detect any such powerful hitters. "The number is derived from the number of possible five-tuple flows: source IP address (32 bits), destination IP address (32 bits), source port (16 bits), destination port (16 bits), and protocol" (8 bits). Because not all potential combinations of these fields are possible, this figure may be much lower for realistic network flow". The goal of heavy-hitter detection is to quickly identify the set of flows that account for a significant portion of the link capacity while reducing error rates and memory consumption.

If we already have a stored array of elements A in memory, the solution is simple: we only need to populate the result if the element occurs at least n/k times. But the question is: can we handle the same heavy-hitter problem in real-time, without a local copy, and with restricted memory space ? According to many researchers there is no algorithm that can solve the heavy hitter problem in a timely manner. But the CMS algorithm sure helps as we wish to identify only the largest values in a data stream. This idea is captured by the heavy hitters problem - we suppose that there are only a few large or "heavy hitting" elements in the stream, and we want to have a streaming algorithm that can identify them.

VI. Mathematical Analysis

Finding the upper error bound, by the linearity of expectation,

$$\mathrm{E}[\mathrm{X}_{i,j}] = E[\sum_{k=1}^{n} I_{i,j,k} a_k] = \sum_{k=1}^{n} a_k E[I_{i,j,k}]$$

in above eq_n value of $E[I_{i,j,k}]$:

$$\sum_{k=1}^{n} a_k E[I_{i,j,k}] \leq \sum_{k=1}^{n} \frac{\epsilon}{e} a_k = \sum_{k=1}^{n} a_k$$

Finally, since $\sum_i a_i \leq ||a||_1$

$$E[I_{i,j,k}] = \frac{\epsilon}{e} ||a||_1$$

This expectation of the error bound of a single row can be used to calculate the probability of the overall query $\epsilon||a||_1$ error bound.

We claim that $\widehat{f_q} \leq f_q + W$ for some over count value W. So how large is W?

Consider just one hash function h_i. It adds to W when there is a collision $h_i(q) = h_i(j)$. This happens with probability $\frac{1}{k}$.

Random variable $Y_{i,j}$ represents the overcount caused on h_i for q because of element $j[n]$. That is, for each instance of j, it increments W by 1 with probability $\frac{1}{k}$, and 0 otherwise. Each instance of j has the same value $h_i(j)$, summing up all these counts. Thus

$$Y_{i,j} = \begin{cases} f_j & \textit{with probability } \frac{1}{k} \\ 0 & \textit{otherwise} \end{cases}$$

$$E[Y_{i,j}] = \frac{f_j}{k}$$

Then let X_i be another random variable defined

$$X_i = \sum_{j\epsilon[n], j\neq q} Y_{i,j}, and$$

$$E[X_i] = E[\sum_{j\neq q} Y_{i,j}] = \sum_{j\neq q} \frac{f_j}{k} = \frac{F_1}{k} = \epsilon\frac{F_1}{2}$$

Now we recall the Markov Inequality. For a random variable X and a value $\alpha > 0$, then $Pr[|X| \geq \alpha] \leq \frac{E[|X|]}{\alpha}$.Since $X_i > 0$, then $|X_i| = X_i$, and set $\alpha = \epsilon F_1$. And note $\frac{E[|X|]}{\alpha} = \frac{(\frac{\epsilon F_1}{2})}{\epsilon F_1} = \frac{1}{2}$

It follows that

$$Pr[X_i \geq \epsilon F_1] \leq \frac{1}{2} \tag{1}$$

This was only for 1 hash function h_i. Now we extend this to t independent hash functions:

$$Pr[\widehat{f_q} - f_q \geq \epsilon F_1] = Pr[min_i X_i \geq \epsilon F_1] = Pr[\forall_{i\epsilon[t]}(X_i \geq \epsilon F_1)]$$

$$= \prod_{i\epsilon[t]} Pr[X_i \geq \epsilon F_1] \leq \frac{1}{2^t} = \delta$$

since $t = log(\frac{1}{\delta})$

So that gives us PAC bound. The count min sketch for any q has,

$f_q \leq \widehat{f_q} \leq f_q + eF_1$

the first inequality always holds, and the second holds with probability at least 1-δ

Space: log(m) space is required by kt counters, the total space is ktlog(m). But we also need to store t hash function, these is made to take log(n)

space each. Then since $t = log(\frac{1}{\delta})$ and $k = \frac{2}{\epsilon}$ it follows total space is $((\frac{2}{\epsilon}log(m) + log(n))log(\frac{1}{\delta}))$.

1. **Improved analysis of count min**

   Our goal is to construct a estimate $\widehat{x}$ of the histogram of the stream , x ϵ $\mathbb{R}^n$, given a stream $u_1 \dots u_n \in [n]$. where $x_u = |u_i : u_i = u, i \in [N]|$

   Count-Min algorithm The Count-Min algorithm assumes a strict turnstile model, since the analysis assumes the final histogram $x \geq 0$

   In order to obtain an estimate for x, pick independently R pairwise independent hash functions $h : [n] \longrightarrow [B]$, for parameters R and B to be chosen. The algorithm will then store R linear sketches $y \in \mathbb{R}^B$ of x , where the jth entry in the ith sketch is given by:

   $$y_j^i = \sum_{u:h_i(j)=j} x_u$$

   Storing these sketches will require O(RB) space. To obtain our final estimator $\widehat{x}$, we compute, for
   each $u \in [n]$,

   $$\widehat{x}u = \min i \in [R] {y_{h_{i(u)}}}^{(i)}$$

2. **Analysis of the above sketch**

   Lemma 1: Take $\widehat{x} \in \mathbb{R}^n$ to be the estimate of x output by Count-Min, using R sketches each of length B. If we choose $R = 2logn$, then with probability $1 - \frac{1}{n}$

   $$||\widehat{x} - x||\infty = \frac{2 \cdot ||x||1}{B}$$

Proof sketch. Observe that, in a strict turnstile model,$y^i_{h_{i(u)}}$ is an estimate of x_u plus some additional terms $x_{u'}$. In particular, denoting $\widehat{x}u^i = yh_{i(u)}{}^i$, we have that:

$$x_u \leq \widehat{x}u^i = xu + \sum_{\substack{v \neq u \\ h_i(v)=h_i(u)}} x_v$$

Therefore, by pairwise independence of the hash functions,

$$\begin{aligned}
\mathbb{E}[\widehat{x}u^i - xu] &= \sum_v \mathbb{E}[x_v \mathbf{1}\{v \neq u, h_i(v) = h_i(u)\}] \\
&\leq \sum_v x_v \mathbb{P}(h_i(v) = h_i(u)) \\
&= \frac{||x||_1}{B}
\end{aligned}$$

As a consequence, by Markov's inequality, $\widehat{x}u^i \leq xu + 2 \times \frac{||x||_1}{B}$ with probability at least $\frac{1}{2}$.Therefore, by taking the minimum of all of our estimators, we may boost the success probability to $1 - \frac{1}{2^R}$. So, if we take R = 2 log n, then $||\widehat{x} - x||\infty = \frac{2 \cdot ||x||1}{B}$ with probability $1 - \frac{1}{n}$

VII. Code(with description of each line)

```
1
2  import java.util.Scanner;
3
4  // declare the class CountMinSketch1
5  class CountMinSketch1
6  {
7      private int[] h1;
8      private int[] h2;
9      private int[] h3;
10     private int size;
11     private static int DEFAULT_SIZE = 11;
12
13     //  Constructor and define the size of table(array) h1,h2 and h3
14     public CountMinSketch1()
15     {
16         size = DEFAULT_SIZE;
17         h1 = new int[ size ];
18         h2 = new int[ size ];
19         h3 = new int[ size ];
20     }
21
22     // clear Function will clear all tables and make them new table */
23     public void clear()
24     {
25         size = DEFAULT_SIZE;
26         h1 = new int[ size ];
27         h2 = new int[ size ];
28         h3 = new int[ size ];
29     }
30
31     // insert Function will be use for inserting the value
32     public void insert(int val)
33     {
34         int hash1 = hashFunc1(val);
35         int hash2 = hashFunc2(val);
36         int hash3 = hashFunc3(val);
37         //  increment counters
38         h1[ hash1 ]++;
39         h2[ hash2 ]++;
40         h3[ hash3 ]++;
41     }
42     // sketchCount Function will be
43     public int sketchCount(int val)
44     {
45         int hash1 = hashFunc1(val);
46         int hash2 = hashFunc2(val);
47         int hash3 = hashFunc3(val);
48         return min( h1[ hash1 ], h2[ hash2 ], h3[ hash3 ] );
49     }
50     // it's compare the given input and find the minimum among them
51     private int min(int a, int b, int c)
52     {
53         int min = a;
54         if (b < min)
55             min = b;
56         if (c < min)
57             min = c;
58         return min;
```

```
59      }
60      // Hash function 1: return the value which modulo by size and with increment
61      private int hashFunc1(int val)
62      {
63          int temp1=val % size;
64          System.out.println();
65          System.out.print("H1 position: ");
66          System.out.println(temp1+1);
67          return temp1;
68      }
69      // Hash function 2: return the val integer with add with 3 and multiplie by val
            and modulo with size and with increment
70      private int hashFunc2(int val)
71      {
72          int temp2=((val * (val + 3)) % size);
73          System.out.println();
74          System.out.print("H2 position: ");
75          System.out.println(temp2+1);
76          return temp2 ;
77      }
78      // Hash function 3: return the value of size-1 with subtract with modulo of val
            and size
79      private int hashFunc3(int val)
80      {
81          int temp3=(size - 1) - val % size;
82          System.out.println();
83          System.out.print("H3 position: ");
84          System.out.println(temp3+1);
85          return temp3;
86      }
87      /* Funtion to print all tables */
88      public void print()
89      {
90          System.out.println("\nCounter Tables : \n");
91          // print the all vulaes of h1 array
92          System.out.print("h1 : ");
93          for (int i = 0; i < size; i++)
94              System.out.print(h1[i] +" ");
95          // print the all vulaes of h2 array
96          System.out.print("\nh2 : ");
97          for (int i = 0; i < size; i++)
98              System.out.print(h2[i] +" ");
99          // print the all vulaes of h3 array
100         System.out.print("\nh3 : ");
101         for (int i = 0; i < size; i++)
102             System.out.print(h3[i] +" ");
103         System.out.println();
104     }
105 }
106
107 public class CountMinSketchTest
108 {
109     public static void main(String[] args)
110     {
111         Scanner scan = new Scanner(System.in);
112         System.out.println("Count Min Sketch Test\n\n");
113         // Object of CountMinSketch
114         CountMinSketch1 cms = new CountMinSketch1();
115
116         char ch;
```

```
117         //  CountMinSketch operations for that run the do while loop because it's
                should be run minimum 1 time for demonstration
118         do
119         {
120             // get the input from user That's what they wants to do
121             System.out.println("\nCount Min Sketch Operations\n");
122             System.out.println("1. insert ");
123             System.out.println("2. get sketch count");
124             System.out.println("3. clear");
125 
126             int choice = scan.nextInt();
127             switch (choice)
128             {
129             case 1 :
130                 // choice 1 for user insert the value
131                 System.out.println("Enter int value");
132                 cms.insert(scan.nextInt() );
133                 break;
134             case 2 :
135                 //choice 2 to get sketch count
136                 System.out.println("Enter int value");
137                 int val = scan.nextInt();
138                 System.out.println("\nSketch count for "+ val +" = "+ cms.sketchCount(
                        val ));
139                 break;
140             case 3 :
141                 // to clear the table
142                 cms.clear();
143                 System.out.println("Counters Cleared\n");
144                 break;
145             default :
146                 // default choice
147                 System.out.println("Wrong Entry \n ");
148                 break;
149             }
150             //  Display the counter table
151             cms.print();
152 
153             System.out.println("\nDo you want to continue (Type y or n) \n");
154             ch = scan.next().charAt(0);
155         } while (ch == 'Y'|| ch == 'y');
156     }
157 }
```

VIII. Results

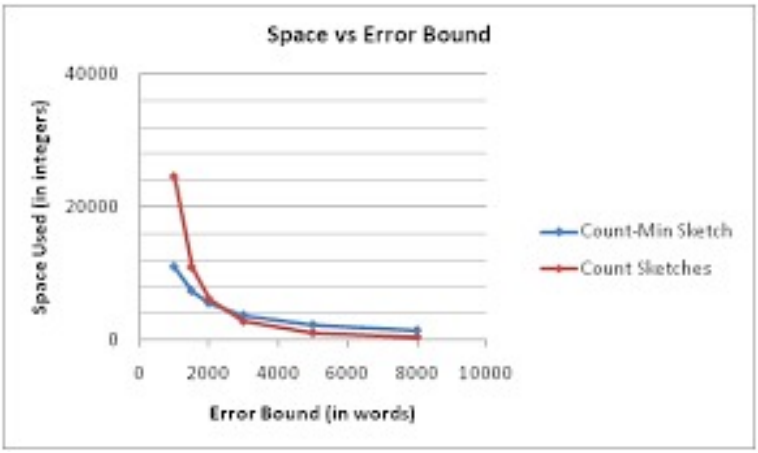
COUNTMIN and related algorithms have been the subject of a number of experimental tests for a variety of computing models. These results suggest that the technique is both accurate and quick to implement. On desktop PCs, implementations accomplish millions of updates per second, with IO performance being the primary limiting factor. Count-Min Sketch has also been included into high-speed streaming systems such as Giga scope and customized. It's capable of processing multi-gigabit packet streams. Lai and Byrd have also described how they implemented Count-Min drawings on a low-power stream processor which can process 40-byte packets at up to 13 Gbps. This translates to approximately 44 million updates per second.

So far, we've talked about a few issues.
When computing the norms of data stream inputs, our CM sketch is ineffective. These can be used to compute correlations across data streams and track the number of distinct components in streams, which are both very useful.

Let's say a popular website wishes to maintain track of the searches people use to find it. After a certain amount of time, maintaining the whole log of inquiries becomes costly and infeasible (in terms of storage space and query processing). Furthermore, perfect statistics are not required in such situations; a modest approximation is sufficient. In this case, a count-min sketch is a handy data structure to keep track of.

How to Build an Estimator

	Count-Min Sketch
Step One: Build a Simple Estimator	Hash items to counters; add +1 when item seen.
Step Two: Compute Expected Value of Estimator	Sum of indicators; 2-independent hashes have low collision rate.
Step Three: Apply Concentration Inequality	One-sided error; use expected value and Markov's inequality.
Step Four: Replicate to Boost Confidence	Take min; only fails if all estimates are bad.



There is a problem, however, with the ‘1-error guarantee that we have just obtained. In many real world data streams, the coefficients decay at a rate of i , for (0.5, 1). Several examples are shown in Figure 1. This rate of decay is problematic because, while kx Hk(x)k1 is not suitable.

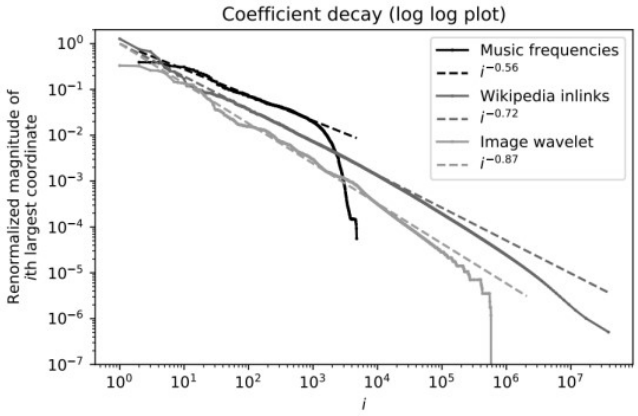


Figure 1: Coefficient decay in three example signals of different domains. In each example, the largest coordinate has magnitude decaying as i for (0.5, 1)

Inference:

The computer science community has mostly developed sketching as a probabilistic data compression approach. Numerical calculations on large datasets can be excruciatingly slow; sketching algorithms solve this problem by creating a smaller surrogate dataset. In most cases, inference is done on the compressed dataset. Random projections are commonly used in sketching algorithms to condense the original dataset, and this stochastic generation process allows for statistical analysis.

IX. References

(n.d.). Sketch Techniques for Approximate Query Processing. Retrieved April 22, 2022, from https://people.cs.umass.edu/ mcgregor/711S12/sketches1.pdf

(n.d.). An Improved Data Stream Summary: The Count-Min Sketch and its Applications. Retrieved April 22, 2022, from https://dsf.berkeley.edu/cs286/papers/countmin-latin2004.pdf

(2019, July 21). Count-Min Sketch. Retrieved April 22, 2022, from https://florian.github.io/count-min-sketch/

Compressing Gradient Optimizers via Count-Sketches. (n.d.). Anastasios Kyrillidis. Retrieved April 22, 2022, from https://akyrillidis.github.io/pubs/Conferences/count$_s$ketch.pdf

$Count-MinSketch.(n.d.).DIMACS.RetrievedApril22, 2022, from$
$http://dimacs.rutgers.edu/\ graham/pubs/papers/cmencyc.pdf$

$Count–minsketch.(n.d.).Wikipedia.RetrievedApril22, 2022, from$
$https://en.wikipedia.org/wiki/Count\%E2\%80\%93min_sketch$

$Countsketch.(n.d.).Wikipedia.RetrievedApril22, 2022, from$
$https://en.wikipedia.org/wiki/Count_sketch$

$Haber, I.(2016, August18).CountMinSketch: TheArtandScienceofEstimatingStuff|$
$https://redis.com/blog/count-min-sketch-the-art-and-science-$
$of-estimating-stuff/$

$Shukla, K.(2018, July16).BigDatawithSketchyStructures, Part1|theCount-$
$MinSketch.TowardsDataScience.RetrievedApril22, 2022, from$
$https://towardsdatascience.com/big-data-with-sketchy-structures-$
$part-1-the-count-min-sketch-b73fb3a33e2a$

$StatisticalAnalysisofSketchEstimators.(n.d.).UFCISE.RetrievedApril22, 2022, fro$
$https://www.cise.ufl.edu/\ adobra/Publications/sigmod-2007-statistics.pdf$

$Java Program to Implement Count Min Sketch - Sanfoundry. (2013, October 12). Sanfou$
$//www.sanfoundry.com/java-program-implement-count-min-$
$sketch/$