Finding Frequent Groups of Items in Data Streams

Abhishek Sharma
MS, Computer Science
NYU, Tandon
as10686@nyu.edu

Sofiia Barysheva
BS, Computer Science
NYU, Tandon
sb6266@nyu.edu

Abstract

We lay the groundwork for estimating the frequency of the most frequent group of items in a data stream by building on top of the Charikar et al. [2] COUNTSKETCH data structure. Our method exploits the underlying similarity between frequently occurring items and uses Locality Sensitive Hashing to estimate the frequency of a group of similar items in a data stream. We discuss, provide the analysis for 3 different approaches in order to achieve a bound based on the similarity score between incoming items in the data stream. Subsequently, our implementation with synthetic data further solidifies our theoretical results.

1 Introduction

Finding frequent items (or heavy hitters) is generally a well researched problem in the domain of streaming algorithms, with one of the most widely known solutions being COUNTSKETCH (Charikar et al. [2]), with COUNT MIN SKETCH (Cormode [3]) being another famous approach. In general, the problem is defined as finding the most frequent, $top-k$, items occurring in a given data stream constrained to the number of passes allowed (usually 1-pass) and the overall space complexity required - which makes this problem a bit non-trivial.

To give a formal introduction to the problem, we first need to introduce some terminology that will be used throughout this report. Let $S = q_1, q_2, q_3, \cdots, q_n$ be a representation of the data stream, with each object $q_i \in \Omega = \{o_1, o_2, \cdots, o_m\}$. Furthermore, we define the frequency of each item $f_i = n_i/n$, where n_i is the number of times o_i occurs in S. Next, the COUNTSKETCH (CS) data structure uses t hash tables, with each hash table containing b buckets, essentially making it an array of $t \times b$ counters.

Given this setup, the problem of finding heavy-hitters could be formalized as finding a list of ℓ elements such that the $top-k$ occurring elements are present in that list. This problem, for a general input distribution is hard to solve since there can exist a situation where $n_k = n_{l+1} + 1$ and finding just ℓ elements containing $top-k$ would not be possible. [2] presented an approximate version of this problem which estimates the frequency of the $top-k$ occurring elements up to a relative error of the true frequency, such that for each item o_i occurring in the list, $n_i > (1-\epsilon)n_k$. This version of the problem is what the COUNTSKETCH data structure aims to solve throughout the analysis presented in [2].

The COUNTSKETCH data structure is essentially an array of counters of size $t \times b$. [2] introduces 2 separate hash functions, h and s, such that $h : [m] \to [b]$ and $s : [m] \to \{+1, -1\}$, with s being a pair-wise independent uniformly random hash function. The algorithm works in the following way - each incoming item from the data stream, q_i, is hashed to one of the b buckets using $h_j[q_i]$ and a random sign using $s_j[q_i]$. Next, the count at the location is updated as $CS[h_j[q_i]] += s_j[q_i]$, and this procedure is repeated for all the $j \in [t]$ available hash tables. At the end of the update, the frequency of any given input query, a, is estimated by taking a median over all the t buckets that $h_j[a]$ hashed to, along with the hashed sign $s_j[a]$ as $median_j(h_j[a]s_j[a]) \quad \forall j \in [t]$. During implementation, [2] uses a heap to maintain the $top-k$ counts amongst all the elements seen at a particular step.

We introduce a simple, yet non-trivial update to the approach presented in [2] of using COUNTSKETCH data structure by adding a Locality Sensitive Hashing (LSH) scheme for hashing the incoming items of the data stream. We make use of the property that, for Jaccard similarity metric, J, there exists a locality sensitive hash function h, such that for any two items q_i, q_j, $\mathbb{P}[h[q_i] = h[q_j]]$ depends on $J(q_i, q_j)$. More intuitively, for any two incoming data items, if they're similar (to an extent) then using LSH would result in them being hashed into the same buckets w.h.p., while if they're not that similar then probability of them hashing to the same bucket is quite low.

Our proposed addition to the existing COUNTSKETCH data structure would let us support answers to new types of queries, while maintaining the original query for estimating the frequency of each individual item. That is, we can return the group frequency of a query input based on the group similarity as well as return the group frequency for an unseen item in a data stream. We present and describe in detail 3 different approaches to achieve the said update to the COUNTSKETCH data structure in section 2, along with their analysis and implementation in section 3 and section 4 respectively, and finally conclude with our observations in section 5.

2 The Locality sensitive COUNTSKETCH (LoSC)

Before we present our algorithm to update the COUNTSKETCH structure to be locality sensitive, we begin with a brief discussion of the intuition behind it.

2.1 Intuition

Recall that our objective is to find the most frequent group of items occurring in the data stream. The first approach that we thought of was that of a clustering algorithm, which we could manipulate into preserving the total count for each group, so that we could represent the items in the group by that count. This approach, not only was too hard to formalize, but also even harder to visualize to make any progress. Instead, we thought of the inherent similarity between items occurring in the same group and how that similarity can be extracted. This way it is easier, in comparison to clustering, to formulate a problem where similar items all contribute together to a single count. This count would then be representative of the final frequency of a group of similar items.

This directly led us into exploring Locality Sensitive Hashing and how its applications are very closely related to our problem statement. We hypothesized that, for any 2 given items q_i, q_j, the similarity between these 2 items should dictate their final hashed bucket. This, from our initial understanding, should be enough to give us the results that would have similar items mapping to the same bucket and hence contributing towards the group to which they must belong. However, upon further exploration and analysis, as presented in section 3, this intuition was only half-right.

Not only did we need the locality sensitivity in our hash function, h, that maps items to buckets, but we also needed to update the pair-wise independent random sign hash function, s, to be locality sensitive, for we wanted similar items contributing to the count and not cancelling out their effective counts - as was the case in the original paper [2]. More formally, we wanted similar items to have the same sign w.h.p., such that their contributions get effectively added through the course of processing the data stream and not cancel out due to the uniformly random nature of the sign function.

The process of updating the random sign hash function, s, to be locality sensitive turned out to be a revelation in its own as well. We first used a direct implementation of LSH to update s into mapping from our MinHash-ed (Broder [1]) inputs to $\{+1, -1\}$ signs, with similar items getting the same sign w.h.p.. This proved to work well, but we were interested if adding an extra randomness layer on top of our sign function would help improve the results.To do that instead of mapping the MinHash-ed inputs directly to a bipolar sign, we first mapped them to an intermediate set of slots, and then hashed the value of those slots to $\{+1, -1\}$ randomly. This approach proved to give the same results as the previous approach. This also made us think of generalizing our results when it comes to Locality Sensitive Hash functions. It seems probable that using a different LSH will produce meaningful results on the frequency of the the data item based on its group.

2.2 Algorithm

Given a stream of data, as defined in the paper, we start by assuming that each input item is represented in an equivalent sketch representation of $\{0,1\}^d$ d-dimensional vectors, where d is the maximum size for each vector possible. For example, in a data stream of queries, we could choose the bag of words and tokenize input sentences accordingly.

Modifying the existing structure for the array of counters, we have the $CS_{i,j}$ entry in the COUNTSKETCH structure storing the effective value of the count as before, but now only for similar items in the data stream. That is, for Jaccard similarity, $J(q_i, q_j)$, and MinHash implementation of LSH, we can restrict the mapping of incoming items based on their similarities.

Next, we want to hash our incoming d-dimensional binary vectors, $\{0,1\}^d$, to one of the b buckets in each of the t hash tables subject to their MinHash calculations. We define r MinHash such that - $c_i : \{0,1\}^d \rightarrow [0,1] \quad \forall i \in [r]$. We then update the existing hash function, h, for mapping inputs to buckets by taking as input from the r bands and mapping them to one of the b buckets, such that our new hash function - $h_j : [0,1]^r \rightarrow [b] \quad \forall j \in [t]$. As discussed earlier, we also modify our sign function by trying out 3 different approaches:

COUNTSKETCH Sign Function (CSS): We choose to not update the pairwise independent random sign function, s, and only modify its input parameter to accept the new d-dimensional binary vector representation of the items, such that - $s_j : \{0,1\}^d \rightarrow \{+1,-1\} \quad \forall j \in [t]$.

1-Step LSH Sign Function (1-SLS): We update the sign function, s, to be a locality sensitive hash function. We first define ℓ MinHash such that - $p_i : \{0,1\}^d \rightarrow [0,1] \quad \forall i \in [\ell]$. Then, we define $u_j : [0,1]^\ell \rightarrow \{-1,1\} \quad \forall j \in [t]$. And finally, we replace our old sign function, s, with a new sign function, $\bar{s}$, so that $\bar{s_j}(q) = u_j(p_1(q), p_2(q), \cdots, p_n(q)) \quad \forall j \in [t]$.

2-Step LSH Sign Function (2-SLS): We further updated our sign function by introducing, s', to be a 2-step locality sensitive hash function. We leave the ℓ MinHash we defined above the same such that - $p_i : \{0,1\}^d \rightarrow [0,1] \quad \forall i \in [\ell]$. Next, we define new $w_j : [0,1]^\ell \rightarrow [g] \quad \forall j \in [t]$. And finally, we define, z to be a uniformly random hash function $z_j : [g] \rightarrow \{-1,1\} \quad \forall j \in [t]$. Thus, the new hash function, s', will be defined as, $s'(q) = z_k(w_k(p_1(q), p_2(q), \cdots, p_n(q))) \quad \forall j \in [t]$

And as discussed in the paper, [2], we still maintain the two operations supported by COUNTSKETCH data structure, in addition we could easily swap the sign function with the 1-SLS or 2-SLS functions:

ADD(CS, q_i): $CS[h_j[q_i]] += s_j[q_i] \quad \forall j \in [t]$

ESTIMATE(CS, q_i): return $median_k(CS[h_j[q_i]]s_j[q_i]) \quad \forall j \in [t]$

Once we have made our updates to the COUNTSKETCH data structure, the algorithm remains the same as the one mentioned in the paper, [2]. That is, we keep a track of all the incoming queries by updating the COUNTSKETCH data structure and also maintain a heap data structure to keep a track of the $top - k$ frequency counts encountered at a given point.

3 Analysis

As discussed earlier, we'll analyse 3 different implementations of the sign function. Our analysis throughout the sections 3.1, 3.2, and 3.3 will be based on analysing only 1 of the t available hash tables for any particular query item, a. While, in section 3.4, we present the space complexity of our approach.

Before we begin, we introduce some additional terminology which will be used throughout this section. We define an indicator random variable, $Y_i = \mathbb{1}[\mathbb{P}(h(a)) = \mathbb{P}(h(q_i))]$, for indicating the data items which hash to the same bucket as the query in question, a. Additionally, let v_i be the Jaccard similarity between $J(a, q_i)$, for any q_i in the stream, with $v_{i,j}$ to be a generic value of Jaccard

similarity between any two input vectors q_i, q_j.

$$X = s(a) \sum_{i \in [m]} s(q_i) f(q_i) Y_i \tag{1}$$

$$\textit{also,} \quad \mathbb{E}[Y_i^2] = \mathbb{E}[Y_i] = \mathbb{E}[\mathbb{1}[\mathbb{P}(h(a)) = \mathbb{P}(h(q_i))]] = v_i^r + \frac{1 - v_i^r}{b} \sim v_i^r$$

Where we assume X to be the estimated output from our data structure using the ESTIMATE operation defined earlier.

3.1 COUNTSKETCH Sign Function (CSS)

We analyse our approach of not choosing to update the sign function, s, to be locality sensitive and just adding the locality sensitivity to the hash function, h, mapping the inputs to the b buckets.

$$\mathbb{E}[X] = \mathbb{E}\left[s(a) \sum_{i \in [m]} s(q_i) f(q_i) Y_i \right] \quad \textit{from (1)}$$

$$= \mathbb{E}\left[s(a)^2 f(a) Y_a + \sum_{i \in [m] \setminus \{a\}} s(a) s(q_i) f(q_i) Y_i \right]$$

$$\mathbb{E}[X] = f(a) + \sum_{i \in [m] \setminus \{a\}} \mathbb{E}\left[s(a) s(q_i) f(q_i) Y_i \right] = f(a) + 0 = f(a)$$

Hence, our approach is an unbiased estimator of the true frequency, as presented in [2] and there is no dependence on the similarity between the query a and the items in the data stream.

We proceed with calculating the variance as,

$$Var[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2$$

$$= \mathbb{E}\Bigg[s(a)^2 \sum_{i \in [m]} s(q_i)^2 f(q_i)^2 Y_i^2 + 2 s(a)^2 \sum_{\substack{i,j \in [m] \\ j > i}} s(q_i) s(q_j) f(q_i) f(q_j) Y_i Y_j \Bigg] - f(a)^2$$

$$= \mathbb{E}\left[s(a)^2 \sum_{i \in [m]} s(q_i)^2 f(q_i)^2 Y_i^2 \right] + \mathbb{E}\left[s(a)^2 \sum_{\substack{i,j \in [m] \\ i \neq j}} s(q_i) s(q_j) f(q_i) f(q_j) Y_i Y_j \right] - f(a)^2$$

$$= \sum_{i \in [m]} \mathbb{E}\left[s(q_i)^2 f(q_i)^2 Y_i^2 \right] + \sum_{\substack{i,j \in [m] \\ i \neq j}} \mathbb{E}[s(q_i) s(q_j) f(q_i) f(q_j) Y_i Y_j] - f(a)^2$$

$$= \sum_{i \in [m]} f(q_i)^2 v_i^r + 0 - f(a)^2 = \sum_{i \in [m] \setminus \{a\}} f(q_i)^2 v_i^r$$

$$Var[X] \geq \sum_{\substack{i \in [m] \setminus \{a\} \\ J(a, q_i) \geq v}} f(q_i)^2 v^r = v^r \sum_{\substack{i \in [m] \setminus \{a\} \\ J(a, q_i) \geq v}} f(q_i)^2$$

Therefore, the variance of the new data structure is approximately the same as that of the original COUNTSKETCH if $v = (\frac{1}{b})^{1/r}$.

3.2 1-step LSH Sign function (1-SLS)

Next, we analyse our approach of updating the sign function, $\bar{s}$, to be locality sensitive by adding a 1-step locality sensitivity to the hash function along with the same, h, mapping the inputs to one of the b buckets. We begin with calculating the expectation of the product of sign functions as,

$$\mathbb{P}[\bar{s}(a) = \bar{s}(q_i)] = v^\ell + (1 - v^\ell)/2 = (v^\ell + 1)/2\,; \quad \mathbb{P}[\bar{s}(a) \neq \bar{s}(q_i)] = (1 - v^\ell)/2$$
$$\mathbb{E}[\bar{s}(a)\bar{s}(q_i), a \neq q_i] = v^\ell \quad \text{(since } \bar{s} \text{ is not longer pair-wise independent)}$$

$$\mathbb{E}[X] = \mathbb{E}[\bar{s}(a) \sum_{i\in[m]} \bar{s}(q_i)f(q_i)Y_i] = \sum_{i\in[m]} f(q_i)v_i^{r+\ell} \quad \textit{from (1)}$$

$$\mathbb{E}[X]^2 = \left(\sum_{i\in[m]} f(q_i)v_i^{r+\ell}\right)^2$$
$$= \sum_{i\in[m]} f(q_i)^2 v_i^{2(r+\ell)} + \sum_{\substack{i,j\in[m]\\ i\neq j}} f(q_i)f(q_j)v_i^{r+\ell}v_j^{r+\ell}$$

$$\mathbb{E}[X^2] = \mathbb{E}\left[\left(\bar{s}(a)\sum_{i\in[m]} \bar{s}(q_i)f(q_i)Y_i\right)^2\right]$$
$$= \mathbb{E}\left[\sum_{i\in[m]} Y_i^2 f(q_i)^2\right] + \mathbb{E}\left[\sum_{\substack{i,j\in[m]\\ i\neq j}} Y_iY_j\bar{s}(q_i)\bar{s}(q_j)f(q_i)f(q_j)\right]$$
$$= \sum_{i\in[m]} v_i^r f(q_i)^2 + \sum_{\substack{i,j\in[m]\\ i\neq j}} v_i^r v_j^r v_{i,j}^\ell f(q_i)f(q_j)$$

$$Var[X] = \sum_{i\in[m]} v_i^r f(q_i)^2 + \sum_{\substack{i,j\in[m]\\ i\neq j}} v_i^r v_j^r v_{i,j}^\ell f(q_i)f(q_j) -$$
$$\sum_{i\in[m]} f(q_i)^2 v_i^{2(r+\ell)} - \sum_{\substack{i,j\in[m]\\ i\neq j}} f(q_i)f(q_j)v_i^{r+\ell}v_j^{r+\ell}$$
$$= \sum_{i\in[m]} v_i^r f(q_i)^2(1 - v_i^{r+2\ell}) + \sum_{\substack{i,j\in[m]\\ i\neq j}} v_i^r v_j^r f(q_i)f(q_j)(v_{i,j}^\ell - v_i^\ell v_j^\ell)$$
$$\geq v^r(1 - v^{r+2\ell})\sum_{\substack{i\in[m]\\ J(q_i,q_j)\geq v}} f(q_i)^2 + v^{2r}(v^\ell - v^{2\ell})\sum_{\substack{i\in[m]\\ J(q_i,q_j)\geq v, i\neq j}} f(q_i)f(q_j)$$
$$= v^r \sum_{\substack{i\in[m]\\ J(q_i,q_j)\geq v}} f(q_i)^2 + v^{2r+\ell}\sum_{\substack{i\in[m]\\ J(q_i,q_j)\geq v, i\neq j}} f(q_i)f(q_j) - v^{2r+2\ell}\sum_{\substack{i\in[m]\\ J(q_i,q_j)\geq v, i\neq j}} f(q_i)^2 + f(q_i)f(q_j)$$
$$= v^r \sum_{\substack{i\in[m]\\ J(q_i,q_j)\geq v}} f(q_i)^2 + v^{2r+\ell}\sum_{\substack{i\in[m]\\ J(q_i,q_j)\geq v, i\neq j}} f(q_i)f(q_j) - v^\ell(f(q_i)^2 + f(q_i)f(q_j))$$

$$Var[X] = v^r \sum_{\substack{i\in[m]\\ J(q_i,q_j)\geq v}} f(q_i)^2 + v^{2r+\ell}\sum_{\substack{i\in[m]\\ J(q_i,q_j)\geq v, i\neq j}} f(q_i)f(q_j)(1 - v^\ell) - v^\ell f(q_i)^2$$

Here we can see that the result of the query is in fact dependent on the frequency of a group that a belongs to (the data items similar in value to a).

Unsurprisingly, $Var[X] = 0$ if $v_i = 1 \quad \forall i \in [m]$. This is consistent with the fact that if all values are the same, they map to the same place no matter what. Thus, variance in that case would be 0. We

can also see that the variance is close to 0 when v is close to 1. So the more well separated the groups are, the smaller the variance of that group.

3.3 2-step LSH Sign function (2-SLS)

Finally, we analyse our approach of updating the sign function, s', to be locality sensitive by adding a 2-step locality sensitivity to the hash function along with the same, h, mapping the inputs to of the b buckets. We begin with calculating the expectation for the product of sign functions as,

$$\begin{aligned}
s'(x) &= z(w(p_1(x), p_2(x), \cdots, p_n(x))) \\
q'_i &= w(p_1(q_i), p_2(q_i), \cdots, p_n(q_i)) \\
a' &= w(p_1(a), p_2(a), \cdots, p_n(a)) \\
\mathbb{P}[s'(a) = s'(q_i)] &= \mathbb{P}[a' = q'_i] = v^\ell \\
\mathbb{P}[s'(a) \neq s'(q_i)] &= \mathbb{P}[a' \neq q'_i] = 1 - v^\ell \\
\mathbb{E}[s'(a)s'(q_i)] &= \mathbb{P}[a' = q'_i]\,\mathbb{E}[z(a')z(q'_i), a' = q'_i] + \\
&\qquad \mathbb{P}[a' \neq q'_i]\,\mathbb{E}[z(a')z(q'_i), a' \neq q'_i] = v^\ell
\end{aligned}$$

Therefore, it seems that adding another level of randomness does not change our results in any way, which means that 2-SLS an 1-SLS are equivalent.

Note, that we could have used SimHash[4] (the locality sensitive hash function for cosine similarity) instead of MinHash. It would have given us results consistent with our findings for Jaccard similarity. Thus, instead of v^ℓ for $\mathbb{E}[s(a)s(q_i)]$ the we would have gotten $(1 - \frac{\theta}{\pi})^\ell$ where θ is the inverse of cosine similarity between a and q_i. Therefore, it seems very likely that we can pick and choose the similarity function that is the best fit for our data set and retrieve meaningful results by applying any LSH to the COUNTSKETCH data structure.

3.4 Space Complexity

We have not added anything to the data structure apart from introducing new hash functions. Therefore, for each hash function that existed before, we added at most 3 new hash functions. Thus, we changed the space complexity by constant order, therefore not changing the big-O overall.

Additionally, our analysis depends on the fact that there is an effective and efficient way of representing data items as binary d-dimensional vectors.

4 Implementation

To test our theoretical findings we decided to implement LoSC in python [1]. First, we needed to acquire hash functions for h, s, and $\bar{s}$. Here we got into a bit of a problem - a library that implements MinHash by taking in d-dimensional $[0, 1]$ vector as a parameter does not exist (or is not available to the public). Therefore, we had to define our own MinHash as well as the uniform random hash function that maps $[0, 1]^\ell \to [b]$, $[0, 1]^r \to [b]$, as well as $\{0, 1\}^d \to \{-1, 1\}$. For MinHash function, our research online showed that the closest we can get to implementing a uniform random function that maps an index to $[0, 1]$ is by using a formula $((a * val + b) \mod c)/c$, where a and b are uniformly randomly generated and c is a preset value that should always be prime (we set it to be $c = 2^{31} - 1$).

The difficulty in creating a uniformly random hash function that maps the array of floats to a range of integers is to represent the array in a format that can be hashed. The only possible way we found to do so was by assigning each item in the array a precomputed random weight and adding up the numbers to create a unique representation of each array and map it to one of the b buckets uniformly randomly. We predefined the randomized weights for every hash function that needs to hash a vector to an integer. Although this hash function is not uniformly random anymore because of the manipulations with the vector, it is the closest we could get to creating a uniformly random hash function and

[1] Github

we confirmed the randomness (up to a degree) for each of the created hash functions by running simulations.

To test our code we used synthetic data sets, thus, not addressing the conversion of real-life data to its d-dimensional representation. We noticed that LoSC 1-SLS works well for dense data vectors. This is consistent with our theoretical findings since both expectation and variance depend on Jaccard similarity by $O(v^{r+\ell})$. Thus, larger and in general more dense d-dimensional binary vectors would give more accurate results than their smaller counterparts. This makes us think that LoSC would also work well for use cases such as analyzing large data sets by using a bag of words strategy.

Over the process of implementing the algorithm we had to make a few sacrifices and assumptions about the hash functions that we created. Our first limitation was the fact that the hash functions that we defined were not in fact uniformly random. We saw this play a role in our algorithm. During testing it was evident that certain (always random) groups of indices were prioritized because of the randomly assigned weights.

While our test sets were on a smaller side, they produced results consistent with our analysis. In the future we see an opportunity to expand our testing by generating data sets representative of a general distribution of larger size.

5 Conclusions

We make a final note of comparison between COUNTSKETCH and LoSC. We were able to show that basic changes in the mapping scheme, such as using an LSH hash function instead of the mapping technique of COUNTSKETCH, does not modify the results that COUNTSKETCH produces. The variance of CSS is within a constant factor (subject to the similarity constraints). Moreover, we managed to modify the sign function as well to estimate the frequency of a group of similar items using 1-SLS and 2-SLS. We also do not rely on the fact that the query is run on a data item that we've seen before or that it belongs to our original data set. The only requirement that we have is that the data item can be converted to a d-dimensional vector in order to get mapped using LoSC.

In addition, it would be valuable to apply different types of LSH functions (apart from Jaccard similarity and cosine similarity) to COUNTSKETCH to see if there is a particular similarity function/data type that could benefit more from the combination of COUNTSKETCH and an LSH (or would not require conversion to a d-dimensional vector). Moreover, it would be interesting to further explore the possibility of analyzing the effects of COUNTSKETCH algorithm on LoSC and see if it in fact produces the $top - k$ most frequent groups. The difficulty with that is the fact that the same group (if frequent enough) might map to different buckets in the same table. This is a problem because, then the $top - k$ list might return values representative of the same groups. Therefore, we cannot claim (hopefully just yet) that $top - k$ algorithm in COUNTSKETCH will produce the correct results. We are hopeful that modifications to the algorithm or the size of the tables could allow us to produce more accurate estimates of the $top - k$ most frequent groups.

References

[1] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *Combinatorial Pattern Matching*, pages 1–10. Springer Berlin Heidelberg, 2000.

[2] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*, volume 29, pages 693–703. Springer Berlin Heidelberg, 2002.

[3] Graham Cormode. Count-min sketch. In *Encyclopedia of Database Systems*, pages 511–516. Springer US, 2009.

[4] Monika Henzinger. Finding near-duplicate web pages. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '06*. ACM Press, 2006.