Stretch The Sketch: Dynamic Sketch Error

ABSTRACT

The ability to calculate a particular function on a stream is a fundamental problem in various network domains. For example, flow frequency estimation or identifying heavy hitter flows. Answering such a query precisely requires linear processing of some data medium. In some cases, linear processing is infeasible, and we are willing to settle on an approximate answer but within sub-linear time. The well-established method of using sketches assumes a fixed size of allocated memory. That is, these sketches are configured with parameters according to the available memory. To address this issue, we present algorithm, Sketch-Ring (SR), for approximate a given function over a stream in case of available flexible memory. It is adaptive to currently available memory.

1 INTRODUCTION

High-performance stream processing is essential for many applications such as financial data trackers, intrusion-detection systems, network monitoring, and sensor networks. Recently, asking questions in real-time about incoming streams of data has become important. Streams typically involve extremely high-volume of data arriving relatively rapidly. Hence, it is often not feasible to store the entire stream and definitely not realistic to scan all the data in order to answer queries about it [1]. Such applications require algorithms that are both time and space efficient to cope with high-speed data streams. Space efficiency is needed, due to the memory hierarchy structure, to enable cache residency and to avoid page swapping. This residency is vital for obtaining good performance, even when the theoretical computational cost is small (e.g., constant time algorithms may be inefficient if they access the DRAM for each element). To that end, stream processing algorithms often suggest an approximate data structures and algorithms, also known as *sketches*. Specifically, when N is the maximal possible answer, sketches return an answer that may deviate by $\pm\epsilon N$ from the correct answer; this is known as a *multiplicative error*. Similarly, a solution may return an answer that may be $\pm\epsilon$ from the correct result, known as an *additive error*.

Previous approaches, sketch based techniques, use static pre-allocated memory according to a given error, ϵ. For a given ϵ and δ, for example, frequency estimation algorithms have an additive error of $N\epsilon$ with a probability of at least $1 - \delta$. These methods can mainly answer point queries. That is, given a flow identifier, the sketching method generates an estimation for that flow. The most famous sketches of this technique are Counting Sketches such as Multi Stage Filters [2] and Count Min Sketch [3].

However, previous works fail to represent streams in dynamic memory. A naïve solution is to construct a new sketch with the newly available memory and move all the elements from the old one to the new one. But this approach costs a significant run time overhead.

Consequently, we design an algorithm called Sketch-Ring (SR) algorithm for estimating a given function over a stream. SR algorithm is flexible in terms of dynamic memory.

Paper roadmap: We briefly survey related work in Section 2. We state the formal model and problem statement in Section 3. The *SR* algorithm is described in Section 4. The performance evaluation of our algorithms is detailed in Section **??**. Section 6 discusses extensions of our work.

2 RELATED WORK

Count-min sketch [3] is composed of a set of d hash functions, and a 2-dimensional array of counters of width w and depth d. To add an item x of value v_x, Count-Min sketch increases the counters located at $CM[j, h_j(x)]$ by v_x, for $1 \leq j \leq d$. Point query for an item q is done by getting the minimum value of the corresponding cells. UnivMon [4] is a generic sketch scheme but its size cannot be changed dynamically. Memento [5] is a family of algorithms for the Heavy Hitters problem which uses a sliding window for better accuracy, however the window size is fixed.

Elastic Sketch [6] is a generic and elastic; it is generic in terms of its measurement tasks as it keeps packet information; and it is generic in terms of platforms as it can be implemented both in software and hardware. Although it also supports dynamic resizing, their algorithm is tailored to its underlying CM (count-min sketch [3]).

[7] solves the problem of approximate dynamic set membership and decouples the dependency between the length of the filter and the indices of the buckets that store the elements. It considers the elasticity of the capacity and provides an algorithm called CCF, which is space-efficient and can be flexible. CFF is composed of multiple I2CFs. I2CF uses a consistent hash ring of multiple buckets. Elements are mapped to these buckets using k hash functions. CFF resize its capacity by adding I2CFs or merging ones.

3 PRELIMINARIES

Given a *universe* $\mathcal{U}$, a stream $\mathcal{S} = x_1, x_2, \ldots \in \mathcal{U}^*$ is a sequence of universe elements. A query calls for computing certain functions of interest on $\mathcal{S}$. Given function f, we defined a query as follow: We define the S-f problem to be the result of activating f on the elements in $\mathcal{S}$.

Given estimation accuracy ϵ, initial number of sketches, k, and hash function that maps the elements from universe $\mathcal{U}$ to the sketches, we consider approximate algorithm that answer an estimation of the queries on element x with bounded error of ϵ, denoted by $\widehat{f_\epsilon}(x)$.

An approximate algorithm for an S-f problem supports two operations:

- **Add(x)** - append x to $\mathcal{S}$
- **Query(x)** - given $x \in \mathcal{U}$, return an estimation $\widehat{f_\epsilon}(x)$ of $f(x)$
- **Shrink()**- decrease the allocated memory, which translated to new allowed error, $\epsilon_{new} = 2\epsilon$.
- **Expand()**- increase the allocated memory, which translated to new allowed error, ϵ_{new}, which is decreases $\frac{k}{k+1}$ times, $\epsilon_{new} = \frac{k}{k+1}\epsilon$.

Symbol	Meaning
$\mathcal{S}$	the data stream
N	number of elements in the stream
$\mathcal{U}$	the universe of elements
f	the function of interest
$\mathcal{S}_f$	the result of activation f on $\mathcal{S}$
$f_\epsilon(x)$	asking for the function f on an element x in $\mathcal{S}$
$\widehat{f_\epsilon}(x)$	an estimation of c_H
ϵ	estimation accuracy parameter
δ	probability of failure
k	number of initial sketches

Table 1: List of Symbols

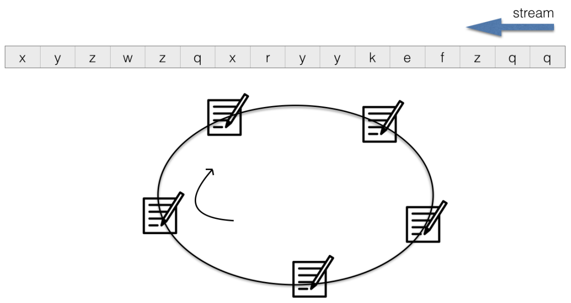

Figure 1: The stream is logically mapped to the sketches in the consistent hash ring.

Examples of interesting f functions include, e.g., frequency, heavy hitters and count distinct.

We now formalize the required guarantees.

Definition 1. Given an error of ϵ and function f, algorithm $\mathcal{A}$ solves S-f problem if given a Query(x) $\mathcal{A}$ satisfies
$f(x) \le \widehat{f_\epsilon}(x) \le f(x) + N\epsilon$.

Note that in our setup the available free memory changes dynamically, which is translated to dynamically changed error. Our data structure can dynamically allocate and free appropriate memory according to the given error ϵ while keeping these changes occur fast and provide accurate answers. Table 1 summarizes the main notations used in this work.

4 SKETCH-RING ALGORITHM

To represent stream in elastic memory restrictions and support generic functions (heavy-hitters, frequency, count-distinct, etc.), we use a sketch that corresponds to the desired function as a black box algorithm in our model. If we are interested in serving multiple functions simultaneously, we use the universal sketch UnivMon [4].

In this section we describe the design of the Sketch-Ring algorithm (SR) in detail, including its data structure, operations and resizing strategies. Basically, SR consists of multiple sketches, according to the desired function we wish to compute on the stream. Given initial ϵ, we configure each sketch with error of $k \cdot \epsilon$, where k is the number of the sketches in the beginning. Note that the number of the sketches changes according to the available memory. This number can increases or decreases. The sketches are mapped onto a consistent hash ring [8] [9] ranging from 1 to $M-1$ as shown in Figure 1. We use also hash function that maps elements from $\mathcal{U}$ to the range $[1 \cdots M]$ to determine the sketch of an element x of the stream. Then, the nearest sketch in a clockwise order is regarded as the processor of element x. Besides, each sketch has a counter of number of the elements inserted to the sketch.

Insertion/Query: To insert/Query element x, the hash function maps x onto the consistent hash ring. Based on the generated value, SR determines the processor sketch of element x, denoted by S. Then, in insertion, x is inserted to S and in query we ask S about the frequency of x.

Algorithm 1 SR Algorithm

```
1: function UPDATE($x$)
2:     Calculate the hash value $h(x)$
3:     $S \leftarrow sketches[nearestIndex(hash(x))]$
4:     $S.Add(x)$
5: function QUERY($x$)
6:     Calculate the hash value $h(x)$
7:     $S \leftarrow sketches[nearestIndex(hash(x))]$
8:     return $S.Query(x)$
```

Algorithm 2 SR Resize

```
1: function SHRINK($SR$)
2:     Select the least-loaded sketch $S_l$ in SR
3:     $HH = getHeavyHitters(S_l)$
4:     $SR = SR \setminus S_l$
5:     for each $e \in HH$ do
6:         $S_{l+1}.Add(e)$
7: function EXPAND
8:     $j \leftarrow$ the index of the most loaded sketch
9:     $S_{new} \leftarrow$ empty sketch mapped between i and j sketches
10:    $elementsToInsert = getElementsIndexes(S_j)$
11:    for each $e \in elementsToInsert$ do
12:        if $i \le e.index \le new$ then
13:            $S_j.remove(e)$
14:            $S_{new}.update(e)$
```

4.1 Resizing of SR

An essential challenge for our problem is the unpredictable available memory, which changes dynamically. This leads to the requirement for our data structure to resizing. The available memory may increases or decreases. Accordingly, we propose two operations, **Expand**, which adds sketches to the SR. This operation occurs when the available memory increases, which leads to more accurate answers (decreases ϵ). The second operation is **Shrink**, which removes sketches from the SR. This operation occurs when we need to deallocate memory from our data structure, which leads to an increase in the error of the estimated answers. SR achieves elasticity using these methods. Now we describe how to insert and remove sketch in these operation:

Expand, When a new sketch is added into the SR, only the elements stored in the successor sketch may be affected. First, we search the most-loaded sketch according to its counter, which counts the number of elements inserted to the corresponding sketch, denote by j the index of that sketch. We consider that a new sketch S_{new} is mapped between two existing sketches, S_i and S_j such that S_j is the successor of S_{new}. Thus, there are elements in S_j that might have to be reallocated to the inserted sketch. The reallocated elements satisfy that their indexes are between i and the new index of S_{new}. To that end, we scan the elements of S_j, for each element e, if its index, i.e., $h^{-1}(e)$, is greater or equal than i and less or equal than new, we move it from S_j to S_{new}. The pseudo-code of the **Expand** operation described in Line 7 in Algorithm 2.

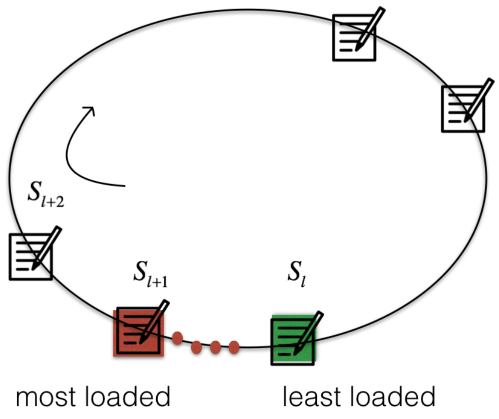

Figure 2: The case where the successor of the least loaded sketch is the most loaded one.

Shrink, In the same way, SR can remove sketches in case of reducing the available memory for the algorithm. The nominated sketch to be removed is the least-loaded sketch, we can search for this sketch according to the counters that pinned along with the sketches in the SR structure. Denote by S_l the least loaded sketch. We assume that the used sketches support heavy hitters query, i.e., returns the most massive elements. We get the heavy hitters of sketch S_l and insert them to S_{l+1} to maintain the correctness of the algorithm. Then we can remove S_l safely. Thus when we look for an element which was previously mapped to S_l, and S_l now is removed, we now search for it in S_{l+1} according to the algorithm of the query operation. The pseudo-code of the **Expand** operation described in Line 1 in Algorithm 2.

Note that we assume that SR resizing occurs once a while, and the most frequent operations are update and query. Otherwise, we can maintain the counters of the sketches in two heaps, maximum and minimum heaps, and each sketch contains a pointer to its counter. Thus, we improve the actions of getting the most-loaded and the least-loaded sketch, which is part of expand and shrink operations. Maintaining these heaps affects the update operation because, in an update, we increment the relevant counter, which leads to sift-up/sift-down operation in the two heaps.

4.2 Optimizations

In **Shrink** operation, up till now, we search for the least loaded search, S_l, move its heavy hitters to the next sketch, S_{l+1}, the successor sketch, and then remove it from the SR. We might get the case that the successor sketch is the most loaded in the SR, so adding elements to this sketch increases its error. It means many elements get indexes between l and $l+1$. This case is illustrated in Figure 2. To overcome this, we can remove both sketches, S_l and S_{l+1}, and place a new sketch between them, let's donate that sketch by $S_{l'}$. Heavy hitters that their indexes are between l and l' are placed into $S_{l'}$ and the indexes between l' and $l+2$ are places into S_{l+2}. Figure 3 describes the dealing with this case.

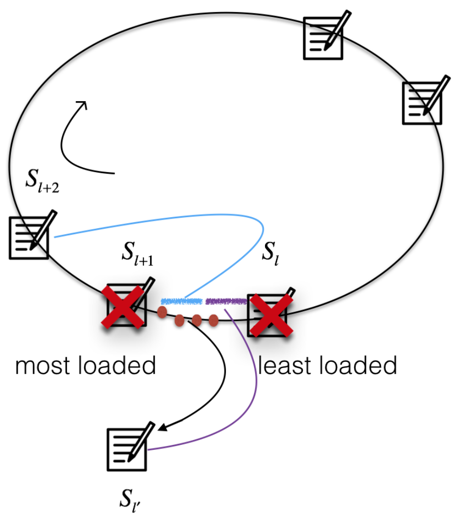

Figure 3: Dealing with the case of the successor of the least loaded sketch is the most loaded one.

5 ANALYSIS

In this section we theoretically analyze the algorithm operations time complexity and algorithm correctness. In the analysis we consider the problem of elements frequency, and we use count-min sketch [3] which is described in the related works. We start to prove the correctness of our algorithm according to Definition 1.

THEOREM 2. *Algorithm 1 solves* S*-frequency problem*

Proof: Given ϵ, we need to prove that upon Query(x), SR produces an estimation $\widehat{f(x)}$ that satisfies: $f(x) \leq \widehat{f(x)} \leq f(x) + N\epsilon$.

We assume that the workload distributed equally between the k sketches (the current number of the sketches is k). The algorithm uses an instance of a count-min sketch, and in SR query, we compute the relevant sketch index and call for a query of the count-min sketch. That is, we need to prove that the estimation of one sketch, $\widehat{f(x)}$, offers the following guarantees:

- $f(x) \leq \widehat{f(x)}$
- With probability at least $1-\delta$, $\widehat{f(x)} \leq f(x) + N\epsilon$

The idea of the proof is to analyze the expected error for one row of the sketch.

The first part results directly for the fact that we overestimate the correct value of $f(x)$ since the updates of the count-min sketch cannot be negative. That mean that the correct value, $f(x)$ is contained in our estimation, $\widehat{f(x)}$.

For the second part, we define $\widehat{f_j(x)}$ as $CM[j, h_j(i)]$ which is the estimation for $f(x)$ in row j. Because of our observations of part 1, we can say that $CM[j, h_j(i)] = f(x) + X_{i,j}$. which mean the estimation consists of the correct value f(x) and an error term. Our goal is to bound the error with $N\epsilon$. We define indicator $I_{i,j,k}$ that indicate if there is a hash collision of the index i and a different index k concerning the hash function j.

$$E[I_{i,j,k}] = Pr[h_j(i) = h_j(k)] = \frac{\epsilon'}{e}$$

Because h_j is chosen from a family of universal hash function as described in [3]. Formally, $X_{i,j}$ can be expressed:

$$X_{i,j} = \sum_{k=1}^{N'} I_{i,j,k} * f_k(x)$$

Using linearity of expectation we get, where N' is the number of elements at one sketch:

$$E[X_{i,j}] = E[\sum_{k=1}^{N'} I_{i,j,k} * f_k(x)] \leq \sum_{k=1}^{N'} f_k(x) \cdot E[I_{i,j,k}]$$

Applying $E[I_{i,j,k}] \leq \frac{\epsilon'}{e}$ and the definition of the L_1 norm:

$$E[X_{i,j}] \leq N' \cdot \frac{\epsilon'}{e}$$

.

We considers three cases:

- Case 1: The case of SR without resize operation
- Case 2: The case of SR with one **Expand** operation
- Case 2: The case of SR with one **Shrink** operation

Case 1*:* Using the assumption that the elements distributed equally between the k sketches, we get that $N' = \frac{N}{k}$. Besides, $\epsilon' = \epsilon \cdot k$ as mentioned in Section 4. We get that

$$E[X_{i,j}] \leq N \cdot \frac{\epsilon}{e}$$

.

Case 2*:* **Expand** operation leads to add one sketch so we get that $N' = \frac{N}{k+1}$, the sketch configured with $\epsilon' = \epsilon \cdot k$ as mentioned in Section 4. As described in section 3, **Expand** operation decreases the error such that $\epsilon_{new} = \frac{k}{k+1}\epsilon$. We get that

$$E[X_{i,j}] \leq \frac{N}{k+1} \cdot \frac{k\epsilon}{e}$$

$$E[X_{i,j}] \leq N \cdot \frac{\epsilon_{new}}{e}$$

Case 3*:* Shrink operation leads to remove one sketch so we get that $N' = \frac{N}{k-1}$, the sketch configured with $\epsilon' = \epsilon \cdot k$ as mentioned in Section 4. As described in section 3, **Shrink** operation increase the error such that $\epsilon_{new} = 2\epsilon$. We get that (assuming $k \geq 2$)

$$E[X_{i,j}] \leq \frac{N}{k-1} \cdot \frac{k\epsilon}{e} \leq \frac{N}{e} \cdot \frac{k}{\frac{k}{2}} \cdot \epsilon = \frac{N}{e} \cdot 2\epsilon$$

.

$$E[X_{i,j}] \leq N \cdot \frac{\epsilon_{new}}{e}$$

In Shrink operation, there is another source of error which caused by moving only the heavy hitters elements of the removed sketch. That is, if we invoke $Query(x)$ such that the element x was in the removed sketch, and it is **not** heavy hitter element. The error in this corner case is bounded by $f(x)$ because there is no representation of x in other sketches right after removing the sketch. Formally, $E[X_i] \leq f(x)$. By heavy hitters definition, given θ we get that: $f(x) \leq (\theta - \epsilon') \cdot N'$. Which leads to:

$$E[X_i] \leq f(x) \leq (\theta - \epsilon') \cdot N' = \frac{N}{k-1} \cdot (\theta - k\epsilon)$$

We can choose θ to guarantee a new ϵ error which is 2ϵ.

The requirement is to analyze the error for our whole estimation and not only the error of one row. The analysis depends on the bound of $E[X_{i,j}]$, which we already prove it, and Markov inequality, and it is identical to the proof of count-min sketch, which was introduced in [3]. □

Theorem 3. *Given ϵ, SR offers running time for update or Query of $O(ln(\frac{1}{\delta}))$, while the space complexity is $O(\frac{1}{\epsilon} \cdot ln(\frac{1}{\delta}))$*

Proof: According to Algorithm 1, in update(x) and query(x), Lines 1 and 5, we calculate the hash value of x in $O(1)$ in order to get the index of the relevant sketch. Then, we call add/query of count-min sketch, which costs $O(ln(\frac{1}{\delta}))$. In total we get that the running time of SR query or update is $O(ln(\frac{1}{\delta}))$. As mentioned before, given ϵ and k sketches, each sketch one is configured with error $\epsilon \cdot k$. The space consumption of one count-min sketch $O(\frac{1}{\epsilon k} \cdot ln(\frac{1}{\delta}))$. SR consists of k count-min sketches and k counters. In total the space consumption of SR is $O(\frac{1}{\epsilon} \cdot ln(\frac{1}{\delta}))$. □

Theorem 4. *The resize operations time complexity is $O(\frac{N}{k} \cdot ln(\frac{1}{\delta}))$*

Proof: According to Algorithm 2, both resize operations, **Expand** and Shrink, find the most/least loaded sketch which costs $O(k)$. Then **Expand** scan the elements of the relevant sketch, and in **Shrink** get the heavy hitters from that sketch, this operation is bounded by $O(\frac{N}{k}$ (Using the assumption that the elements distributed equally between the k sketches). After that, we update one sketch with these elements, according to count-min sketch [3], this costs $O(ln\frac{1}{\delta})$. In total we get $O(\frac{N}{k} \cdot ln(\frac{1}{\delta}))$. □

6 EXTENSIONS

In SR algorithm, we use a hash function to map the elements from the universe $\mathcal{U}$ to the sketches using the hash function. We assume that the hash function distributes the elements equally; however, for such function, there is at least a workload that doesn't distribute equally between the sketches. Thus, we can extend the SR algorithm to resolve this issue. To ensure better load balancing between the sketches, we maintain T hash functions that map the elements from the universe to the range $1 \cdots M$. In $Update(x)$, we calculate the T hash functions on x that maps x to T candidate sketches. Then choose the least loaded sketch to insert x. In $Query(x)$, we also calculate the T hash functions on x and invoke query of the candidate sketches **in parallel** and merge the results, e.g., when f is counting the points. Alternatively, we can merge the sketches first and query the merged sketch in a case f is count-distinct, for instance. One primary requirement is that the sketches we maintain are merge-able [10].

REFERENCES

[1] Tova Milo. Getting Rid of Data. VLDB keynote talk, 2019.

[2] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 323–336, 2002.

[3] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[4] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 101–114. ACM, 2016.

[5] Ran Ben Basat, Gil Einziger, Isaac Keslassy, Ariel Orda, Shay Vargaftik, and Erez Waisbard. Memento: Making sliding windows efficient for heavy hitters. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 254–266, New York, NY, USA, 2018. Association for Computing Machinery.

[6] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: adaptive and fast network-wide measurements. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 561–575. ACM, 2018.

[7] Lailong Luo, Deke Guo, Ori Rottenstreich, Richard TB Ma, Xueshan Luo, and Bangbang Ren. The consistent cuckoo filter. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 712–720. IEEE, 2019.

[8] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[9] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking*, 11(1):17–32, 2003.

[10] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. *ACM Transactions on Database Systems (TODS)*, 38(4):1–28, 2013.