LETFramework: Let the Universal Sketch be Accurate

Ruijie Miao, Xiangwei Deng, Zicang Xu, Ziyun Zhang, Tong Yang
Peking University, China

***Abstract*—Sketching algorithms are considered as promising solutions for approximate query tasks on large volumes of data streams. An ideal general-purpose data processing engine requires a sketch to achieve (1) high *genericness* in supporting a broad range of query tasks; (2) high *fidelity* in providing accuracy guarantee; and (3) high *performance* in practice. Although the universal sketch achieves high genericness and fidelity, its accuracy falls short of expectations. In this paper, we propose LETFramework (short for Lossless ExTraction Framework) to optimize the performance of the universal sketch. With the key technique of *lossless extraction*, LETFramework losslessly extracts the main body of the frequent items and stores the remaining information in the universal sketch, thereby achieving higher accuracy while maintaining high fidelity. We further introduce a unified methodology to incorporate the substitution strategies from top-k algorithms into LETFramework. Experiment results show that, LETFramework outperforms the universal sketch, achieving accuracy improvements ranging from 1 to 3 orders of magnitude on most query tasks and up to 15.73 times higher throughput. All the related source code is open-sourced and available at Github.**

***Index Terms*—Data Stream; Sketch; Top-k**

I. Introduction

Processing large volume of data often incurs huge overhead in terms of both memory and time. To mitigate this overhead, a common approach is to trade the query accuracy for improved memory efficiency and processing speed. Research community has explored many approximate algorithms for various query tasks, including frequency estimation [1], [2], [3], inner-production estimation [4], [5], cardinality estimation [6], [7], [8], and more [9], [10], [11], [12], [13], [14], [15], [16], [17].

Sketching algorithms are considered as promising solutions for approximate query tasks, which provide accuracy guarantees with sub-linear memory consumption. Sketches can be classified into two categories based on the number of supported query tasks. The first category is the *dedicated sketches designed for specific query tasks*. Most sketching algorithms fall into this category, such as CU sketch [3] designed for frequency estimation, HyperLogLog [7] designed for cardinality estimation, Unbiased SpaceSaving [12] designed for subset sum estimation, and more [1], [4], [5], [18], [19]. The second category is the *generic sketches designed for multiple query tasks*. Examples of generic sketches include DHS [11] and Elastic sketch [10], which support a wide range of query tasks including frequency estimation, entropy estimation and heavy hitter detection. With the same memory usage, a generic sketch may not outperform a dedicated sketch in terms of accuracy on the specific task. However, for a general-purpose data processing engine that aims to support multiple query tasks, it should employ either one generic sketch or multiple dedicated sketches designed for different tasks. If the total memory consumption is the same, the generic sketch can achieve higher accuracy. Additionally, the generic sketch is also faster because multiple sketches require multiple passes of the data.

The universal sketch [20], [21] is widely recognized as one of the most generic sketches. It supports queries for the aggregated sum of statistics over all items, where each statistic is obtained by applying a function to the frequency of one item. One of the significant advantages of the universal sketch is its *fidelity* in providing accuracy guarantee for a broad range of query tasks. In contrast, many other generic sketches do not offer accuracy guarantee for all the query tasks they support. However, the performance of the universal sketch in both accuracy and processing speed is often unsatisfactory in real world datasets [10], [22], which limits its practical application. Ideally, a generic sketch should offer both high fidelity in theory and high performance in practice.

In this paper, we propose LETFramework (short for Lossless ExTraction Framework), a framework which optimizes the universal sketch, achieving practically high performance and theoretically high fidelity. The key technique of the LETFramework is named *lossless extraction*. By employing lossless extraction, LETFramework losslessly records the main body of the frequent items, while the remaining information, including the small portion of frequent items and the infrequent items, is recorded with the universal sketch with error bounds. We propose a unified methodology to implement lossless extraction with existing top-k algorithms, leveraging their capability in identifying frequent items.

LETFramework consists of two parts: the Top-k part and the USketch part. The purpose of the Top-k part is to extract frequent items, and the USketch part is a universal sketch for infrequent items. The lossless extraction technique is to provide fully accurate estimation for those items inserted in the Top-k part but not passed to the USketch part, in spite of the substitution process. Logically, the whole data stream is divided into two parts: items in the Top-k part, which are accurately recorded, and items in the USketch part, which are bounded with the universal sketch. It is noticeable that,

The corresponding author: Tong Yang (yangtong@pku.edu.cn).

Ruijie Miao, Xiangwei Deng, Ziyun Zhang and Tong Yang are with School of Computer Science, Peking University, Beijing, China.

while we wish to extract all frequent items into the Top-k part, it is infeasible due to limited memory. During the data stream processing, when the data structure reaches its capacity and needs to replace recorded items, a one-pass algorithm cannot distinguish between the infrequent items and the frequent items whose frequency counts is currently low. Therefore, the Top-k part stores the main body of the frequency counts for each frequent item, excluding the first δ counts. The extraction of frequent items is beneficial to the accuracy, considering the imbalanced data distribution in the real world data [23], [24], [25], [26], [27]. One main reason is that it enables more accurate estimation for frequent items, which form the majority of the data stream. Additionally, extracting frequent items results in a more balanced frequency distribution, reducing the estimation variance of the universal sketch which applies its sampling technique.

To efficiently extract the main body of the frequent items, we leverage the advanced design of the substitution strategies in state-of-the-art top-k algorithms. However, directly applying most top-k algorithms as the Top-k part is not feasible due to their dual-purpose counters, which are used for both counting and substitution. This dual-purpose nature prevents achieving lossless extraction. Besides, the designs of the most top-k algorithms do not consider passing information about infrequent items. To address these challenges, we propose a unified methodology to transform the top-k algorithms into the Top-k part. The idea is to introduce a separate pure counter (if not exist) to provide fully accurate estimations, while the original counter solely serves the purpose of substitution. To reduce the memory overhead caused by the increased number of counters, we further propose a technique of substitution counter sharing, which let multiple (ID, pure counter) pairs to share a single substitution counter. We believe our methodology enables the transformation of most top-k algorithms into the Top-k part. We provide case studies with four top-k algorithms: SpaceSaving [28], frequent [29], HeavyGuardian [9] and Elastic Sketch [10].

Prior works [30], [31] have pointed out a wide range of query tasks that are supported by the universal sketch, which are naturally supported by LETFramework. Besides, we show that some query tasks on multiple data streams can be supported, including the estimation of inner-production, cosine similarity, and Jaccard similarity. We further show the high fidelity of LETFramework by theoretical analysis, which proves that LETFramework still provides accuracy guarantee for a wide range of queries. We conduct rich experiments on the performance of LETFramework. The results show that, LETFramework outperforms the universal sketch, achieving accuracy improvements 1 to 3 orders of magnitude on most query tasks and up to 15.73 times higher throughput. Our source code is publicly available at Github [32].

II. Background and Related Work

A. Sketches

Sketches are compact data structures that support approximate queries over large-scale data. Sketches are highly efficient in processing speed, as they only require single pass of the data stream with simple operations. Moreover, sketches provide fast query capabilities with accuracy guarantee. Therefore, sketches are considered promising solutions for approximate query processing. Typical sketches include CM sketch [1], Count sketch (Fast-AGMS) [2], [5], CU sketch [3] and more [7], [12], [18].

Sketches can be classified into two categories: dedicated sketches for specific tasks and generic sketches for multiple tasks. The example of dedicated sketches for specific tasks include CU sketches [3] for frequency estimation, HyperLogLog [7] for cardinality estimation, Unbiased SpaceSaving [12] for subset sum estimation, and more [1], [4], [5], [18], [19]. These sketches achieve high performance for their respective supported tasks due to extensive optimization efforts. However, when using a general-purpose data processing engine, multiple dedicated sketches need to be constructed to support different query tasks. This approach leads to reduced memory efficiency and increased processing overhead.

For generic sketches designed for multiple tasks, they achieve good performance for a broader range of query tasks. However, most of them show the feasibility on their supported tasks through experiments but fail to provide accuracy guarantee, leading to low fidelity. Elastic sketch [10] and DHS [11] support query tasks including frequency estimation, heavy hitter detection, heavy change detection and entropy estimation, but they only provide error bound analysis for the frequency estimation task. On the other hand, the universal sketch [20], [21] achieves high fidelity by providing accuracy guarantee for a wide range of query tasks, but its accuracy is not satisfactory on real world dataset [10], [22]. The main focus of this paper is to develop a generic sketch that achieves both high fidelity and high performance simultaneously.

B. Top-k Algorithms

Top-k items refer to the k most frequent items in the data stream. Approximate algorithms for finding top-k items have been widely studied. Existing top-k algorithms share a similar algorithm design: they use a limited number of buckets, with each bucket typically consisting of an ID field and a counter. When a new item arrives, the algorithm first tries to update the bucket that matches the ID of the incoming item. If such a bucket does not exist, it then tries to insert the item into an empty bucket. In the case where no empty bucket exists, it tries to update the bucket with the smallest counter value. While the main difference among different top-k algorithms lies in their strategies to update the buckets and substitute items.

Popular top-k algorithms include SpaceSaving [28], frequent [29], HeavyGuardian [9], and Elastic sketch [10]. SpaceSaving replaces the ID field by the incoming item, and increments the counter by 1. Frequent decreases the counter by 1. If the counter reaches 0, the ID is replaced by the incoming item and the counter is set to 1. HeavyGuardian decreases the counter by 1 with a probability of b^{-C}, where b is a constant and C is the value of the minimum counter. If the counter reaches 0, the ID is replaced by the incoming item and the counter is set to 1. In the Elastic sketch, each bucket includes an additional field for negative votes. When

```
Algorithm 1: Insertion of the universal sketch
Input: An incoming item $e$
1  for $i = 1$ to $L$ do
2      for $j = 1$ to $d_i$ do
3          $M_i[j][h_i^j(e)] \leftarrow M_i[j][h_i^j(e)] + s_i^j(e)$;
4      $Freq \leftarrow median\left(\{M_i[j][h_i^j(e)] \cdot s_i^j(e)\}_{j=1,\cdots,d}\right)$
5      if $e$ appears in $T_i$ then
6          Update the corresponding frequency to $Freq$
7      else if $T_i$ has empty slot(s) then
8          Insert $(e, Freq)$ to one empty slot
9      else
10         $idx_{min} \leftarrow get\_slot\_index\_with\_minfreq(T_i)$
11         if $idx_{min} < Freq$ then
12             $T_i[idx_{min}].ID \leftarrow e$
13             $T_i[idx_{min}].freq \leftarrow Freq$
14     if $H_i(e) == 0$ then
15         return
```

```
Algorithm 2: Query of the universal sketch
Input: The queried item $e$; the G-sum function $g(\cdot)$
Output: Estimated G-sum $\sum_x g(f_x)$
1  G-sum$_L \leftarrow 0$
2  for $j = 1$ to $k$ do
3      if $T_L[j]$ is not empty then
4          G-sum$_L \leftarrow$ G-sum$_L + g(T_L[j].freq)$
5  for $i = k - 1$ to $1$ do
6      G-sum$_i = 2 \cdot$ G-sum$_{i+1}$
7      for $j = 1$ to $k$ do
8          G-sum$_i += (1 - 2H_i(T_i[j].ID)) \cdot g(T_i[j].freq)$
9  return G-sum$_1$
```

the incoming item updates the bucket with smallest counter, it increases the negative vote field by 1. If the negative votes exceed $\lambda \cdot C$, where λ is a constant and the C is the value of the counter, the ID is replaced by the incoming item, the counter is set to 1 and the negative vote field is reset. In summary, various top-k algorithms use different substitution strategies, which can be incorporated into the design of the Top-k part in the LETFramework. Since there is no theoretical analysis indicating the superiority of one strategy over another, we believe different substitution strategies have different advantages. Ideally, the LETFramework should support the use of different substitution strategies from top-k algorithms.

III. The Universal Sketch

A. Overview

Suppose the item x has frequency f_x in the data stream. Given a function $g(\cdot)$, we define the *G-sum* as the sum $\sum_x g(f_x)$ over all items. The universal sketch uses one sketch to address a broad range of G-sums with an accuracy guarantee. Specifically, it supports arbitrary G-sum functions that are monotonic, tractable, and upper bounded by $O(f^2)$.

We first present a high-level overview of the universal sketch. It consists of multiple layers, and incoming items are inserted layer by layer. Initially, each item is inserted into the top layer. As an item progresses to the i^{th} layer, it has a 50% chance of passing through to the $(i+1)^{th}$ layer, or the insertion process terminates otherwise. In each layer, the universal sketch employs a Count sketch to identify top-k items from the sub-stream of the inserted data. When querying for the G-sum $\sum_x g(f_x)$, the universal sketch utilizes a recursive estimation approach. At a high level, the universal sketch estimates the G-sum for the sub-stream inserted in each layer, starting from the bottom layer and progressing upwards. The final estimation is obtained from the G-sum calculated in the topmost layer, since all items are inserted in the top layer.

B. The Data Structure

The universal sketch consists of $L = \log n$ layers, where each layer is composed of a Count sketch [2] and a top-k array. In the i^{th} layer, the Count sketch is a $d_i \times w_i$ counter matrix M_i, and the top-k array is denoted as T_i. For the Count sketch M_i, the j^{th} row is associated with a hash function h_i^j that maps the ID of the item to one of the columns and a hash function s_i^j that maps the ID to either -1 or 1. Besides, all layers except the bottom layer employs a hash function H_i that maps the ID of the item to either 0 or 1, which is used to decide whether an item can pass to the next layer.

C. Operations

Insertion: The pseudo code of the insertion is shown in Algorithm 1. The incoming item e is initially inserted into the Count sketch M_1 of the top layer and its frequency is estimated as $\hat{f}_e$ (line 2 - 4). The pair $(e, \hat{f}_e)$ is then used to update the top-k array T_1, which maintains the k most frequent items (line 5 - 13). Next, we compute $H_1(e)$ to determine whether the item will be passed to the next layer (line 14 - 15). If $H_1(e)$ equals 1, indicating that e is sampled, it is passed to the next layer and inserted following the same procedure. Otherwise, the insertion of e terminates immediately. The insertion process continues until e is not sampled for the next layer or it reaches the bottom layer.

Query: The pseudo code of query is shown in Algorithm 2. The first step is to estimate the G-sum on the bottom layer by applying the G-sum function to the top-k items in the bottom layer and calculating the sum (line 1 - 4). Then, to estimate the G-sum on the i^{th} layer, the universal sketch multiplies the estimated G-sum of the $(i+1)^{th}$ layer by 2, and adjusts estimation based on top-k items in the i^{th} layer (line 5 - 8). For the top-k items that pass through to the $(i+1)^{th}$ layer, their contribution is subtracted from the estimated G-sum. For the top-k items that are not sampled to the $(i+1)^{th}$ layer, their contribution is added to the estimated G-sum. Finally, the G-sum estimation on the top layer is computed and returned.

D. Advantages and Limitations

The universal sketch can use one sketch to support a broad range of G-sum functions that are monotonic, tractable and bounded by $O(x^2)$. Prior works [30], [31] have highlighted its genericness in supporting many important approximate query tasks. With $g(x) = x^0$, it supports cardinality estimation [6], [7]. With $g(x) = x^2$, it supports second frequency moment estimation [33], [34]. With $g(x) = x \log x$, it supports entropy estimation [11], [35]. Additionally, since the universal sketch uses a Count sketch and a top-k array in the top layer where all

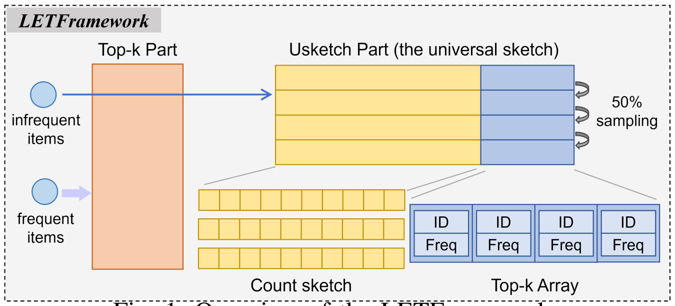


Fig. 1: Overview of the LETFramework.

items are inserted, it naturally supports frequency estimation [1], [3] and heavy hitter detection [2], [10], [9].

However, the experiment results of prior works [10], [22] have shown that the accuracy of the universal sketch is not satisfactory on real world workloads. The key reason is that, the design of the universal sketch is agnostic of the data distribution. Prior research [23], [24], [25], [26], [27] has highlighted the imbalanced nature of real world data, which provides the opportunity to improve the accuracy of the universal sketch. Since frequent items comprise the main body of the data, achieving more accurate estimation for these items will enhance the overall accuracy of the G-sum estimation. Besides, extracting out frequent items leads to a more balanced distribution of data being inserted into the universal sketch, which helps to reduce the estimation variance, as the estimation is based on a recursive sampling.

IV. The LETFramework

In this section we will introduce the data structure and the interfaces. The main interfaces include:

- `Initialize(Sketch)`
- `Insert(Sketch, Item)`
- `SingleStreamQuery(Sketch, Task)`
- `MultiStreamQuery(SketchA, SketchB, Task)`

The interface of initializing is to simply reset all counters and set IDs to null. We will introduce the data structure and other interfaces in the follows.

A. The Data Structure

Data structure: As shown in Figure 1, LETFramework comprises two components: the Top-k part and the USketch part. The Top-k part is used to provide lossless extraction for the main body of the frequent items. The USketch part is a universal sketch that provides G-sum estimation for the remaining data, each layer consisting of a Count sketch and a top-k array. The construction of the Top-k part using existing top-k algorithms will be discussed in §V. In this section the Top-k part can be regarded as a black box that takes an incoming item as input and optionally transfers information about infrequent items to the USketch part. When the insertion process is finished, the Top-k part provides a list of (ID, frequency) pairs for the frequent items.

B. Basic Operations

Insertion: For an incoming item e, it is first inserted to the Top-k part. The Top-k part decides whether the incoming item should be maintained as a candidate of frequent items. If the item e is identified as an infrequent item, the information of $(e, 1)$ is transferred to the USketch part. Otherwise, the item e is decided as a candidate of frequent items and updates the recorded items in the Top-k part. In the updating process it may substitute an item that is already stored in the Top-k part, and in such case the information of the substituted item (e', frequency) will be transferred to the USketch part. For the information transferred to the USketch part, the insertion follows the same process described in Algorithm 1. Initially the item is inserted in the top layer. On each layer, the item has a 50% chance to pass through to the next layer, or otherwise the insertion terminates immediately. In each layer, the item updates the Count sketch and uses the estimated frequency from the Count sketch to update the top-k array.

Basic Query for G-sum: LETFramework supports query for the G-sum $\sum_x g(f_x)$ where f_x is the frequency of the item x and g is the G-sum function. Note that the original universal sketch requires the G-sum function to be tractable, monotonic and upper bounded by $O(x^2)$. LETFramework further requires the G-sum function $g(\cdot)$ to be differentiable, and $g'(x)$ should be bounded by rx for a constant r (see §VI for details). The LETFramework narrows the range of supported G-sum functions. Nonetheless, the supported G-sum functions in the LETFramework still cover a wide range of query tasks encountered in real world scenarios, including all tasks discussed in prior works on the universal sketch [31], [30]. The G-sum functions of $x^0, x\log x, x^2$ are all supported.

Intuitively, in LETFramework, the frequent and infrequent items are divided into the Top-k part and USketch part respectively. Therefore, we calculate the G-sum separately for items in the Top-k part and the USketch part, and combine the results to obtain the final estimation. For items in the Top-k part, the G-sum is estimated by applying the G-sum function to all (ID, frequency) pairs. For the USketch part, the G-sum is computed by following the query of the universal sketch, as described in Algorithm 2. It will recursively estimate the G-sum for the sub-stream of data inserted in each layer, and output the estimation on the top layer as the final estimation.

C. Query on Multiple Data Streams

By supporting a wide range of G-sum functions, LETFramework addresses various query tasks on a single data stream. We extend the capabilities of LETFramework to support query tasks on multiple data streams. In this section we discuss three common query tasks: inner-production estimation, cosine similarity estimation, and Jaccard similarity estimation.

Inner-production estimation: Suppose $f_A(x)$ is the frequency of item x in the data stream A, and $f_B(x)$ is the frequency of item x in the data stream B. The inner-production is defined as $\sum_x f_A(x) \cdot f_B(x)$, which has wide applications [4], [36], [37].

We aim to provide inner-production estimation with two sketches, each built for one data stream. Note that,

$$\sum_x f_A(x) \cdot f_B(x) = \frac{1}{2}\Big(\sum_x (f_A(x) + f_B(x))^2 - \sum_x f_A^2(x) - \sum_x f_B^2(x)\Big) \tag{1}$$

Since LETFramework supports the function $g(x) = x^2$, with two sketches built for data stream A and B we can calculate $\sum_x f_A^2(x)$ and $\sum_x f_B^2(x)$. The remaining problem is how to estimate $\sum_x (f_A(x)+f_B(x))^2$. We enforce the two sketches to use the same hash functions and parameters for the USketch part. Then we merge the two sketches together to create a new sketch for the merged data steam $A+B$, and perform G-sum queries on the new sketch. Merging the USketch parts is simple: as the USketch part provides a list of (ID, frequency) pairs, we can merge the two lists together. If an ID appears in both lists, we sum up their frequencies. For the USketch parts, we merge them layer by layer. Since we use the same hash functions and the same size of Count sketch, we can merge the Count sketches by summing up each corresponding counters. Then we can extract the set of items from both top-k arrays, query the frequencies of these items again in the merged Count sketch and select top-k items in the set.

Cosine similarity estimation: Suppose $f_A(x), f_B(x)$ are the frequencies of the item x in the data stream A and B, respectively. The cosine similarity is defined as:

$$cos(A, B) = \frac{\sum_x f_A(x) \cdot f_B(x)}{\sqrt{(\sum_x f_A^2(x)) \cdot (\sum_x f_B^2(x))}}$$

We have discussed how to estimate $\sum_x f_A(x) \cdot f_B(x)$ above, and estimating $\sum_x f_A^2(x)$, $\sum_x f_B^2(x)$ is already supported by LETFramework. Therefore, cosine similarity estimation can also be supported.

Jaccard similarity estimation: Let S_A be the set of distinct items in the data stream A, and S_B be the set of distinct items in the data stream B. The Jaccard similarity is defined as: $J(A, B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|}$. $S_A \cup S_B$ is the same as the set of distinct items in the merged data stream of A and B. We have already discussed how to build a sketch for the merged data stream. As LETFramework supports the G-sum function $g(x) = x^0$, the cardinality $|S_A \cup S_B|$ can be estimated. $|S_A \cap S_B|$ can be estimated by the formula $|S_A| + |S_B| - |S_A \cup S_B|$. Therefore, Jaccard similarity estimation can be supported.

V. Lossless extraction

In this section, we discuss how to construct the Top-k part using existing top-k algorithms while ensuring lossless extraction of the main body of frequent items. We propose a unified methodology that can transform existing top-k algorithms into Top-k part structures. The key insight is that in top-k algorithms, each ID is usually associated with a single counter, which serves the dual purpose of counting and substitution of top-k items. This dual-purpose nature makes them fail to provide fully accurate estimation. Lossless extraction associate one ID with two counters: a pure counter and a substitution counter. The pure counter provides fully accurate estimation for the corresponding ID, while the substitution counter serves the purpose of replacing for frequent items. To address the low memory efficiency caused by increased number of counters, we let multiple pairs of (ID, pure counter) share a single substitution counter. Our solution achieves lossless extraction at the cost of a little higher computation and memory access overhead, as the insertion should access and update two counters instead of one.

A. Methodology

Our methodology is as follows. We first transform the data structure. As shown in Figure 2, the data structure consists of a bucket array T with length L, and each bucket contains a shared substitution counter and C cells of the (ID, pure counter) pair. Then, we transform the insertion. We apply the technique in the original top-k algorithm to locate the cell. If the ID in the cell matches the incoming item, we only update the pure counter. Otherwise, we update the shared substitution counter and do substitution following the top-k algorithm's strategy. For example, the Frequent algorithm decreases the counter for each unmatched item and replaces the ID when the counter becomes zero. In the transformed version, we increase the substitution counter, and do substitution when the substitution counter is no less than the pure counter. We believe our methodology works on most top-k algorithms.

We provide case studies on four well-known top-k algorithms and construct four versions of the LETFramework: S-LETSketch with SpaceSaving [28], F-LETSketch with Frequent [29], H-LETSketch with HeavyGuardian [9], and E-LETSketch with Elastic sketch. For the first three versions, we need to apply the above methodology to transform them accordingly. For the E-LETSketch, as the Elastic sketch already uses two types of counters, it can be applied to the Top-k part with minimal modifications.

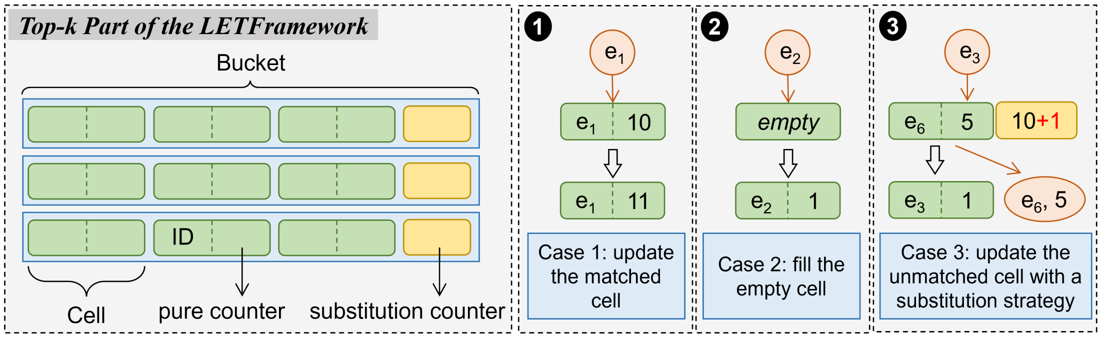
B. S-LETSketch

The Top-k part of the S-LETSketch is based on the SpaceSaving algorithm [28]. In the original SpaceSaving algorithm, the counter represents the sum of frequencies of all the items that are mapped to the bucket. When an incoming item has no match in any bucket and there are no empty buckets, the bucket with the minimum counter is updated. The ID is substituted by the incoming item and the counter is incremented by 1. With the transformation methodology, we use the pure counter to record the fully accurate frequency of an item, and use the shared sum counter as the substitution counter to represent the frequency sum for the substitution purpose.

For an incoming item, if one cell has the matched ID, we increment both the pure counter in the cell and the substitution counter in the corresponding bucket by 1. If no cell has the matching ID but an empty cell is found, we update the empty cell with the ID of the incoming item, set the pure counter to 1, and increment the corresponding substitution counter by 1. Otherwise, we find the bucket with the minimum sum counter, and select the cell with the minimum pure counter to perform substitution. The recorded ID and the pure counter in the bucket are passed to the USketch part. Then the ID is replaced by the incoming item, the pure counter is reset to 1, and the substitution counter is incremented by 1.

C. F-LETSketch

The Top-k part of the F-LETSketch is based on the Frequent algorithm [29]. In the original Frequent algorithm, an incoming item uses a hash function to locate a bucket, and it only updates the cells in the hashed bucket. When there



Top-k Part of the LETFramework
Bucket
ID
Cell
pure counter
substitution counter
1
e1
e1 10
e1 11
Case 1: update the matched cell
2
e2
empty
e2 1
Case 2: fill the empty cell
3
e3
e6 5
10+1
e3 1
e6, 5
Case 3: update the unmatched cell with a substitution strategy

Fig. 2: Overview of the Top-k part with substitution strategies from existing top-k algorithms. The shown example uses the substitution strategies from the SpaceSaving algorithm.

is neither a matching bucket nor an empty bucket, the cell with the minimum counter in the hashed bucket is selected, and its counter is decreased by 1. If the counter reaches 0, the ID is replaced by the incoming item, and the counter is set to 1. With the transformation methodology, we use a pure counter to record the fully accurate frequency of an item, and let the substitution counter to track the decreased value. When the pure counter is smaller than the substitution counter, it is equivalent to the situation where the counter reaches a value lower than zero, and the ID should be replaced.

For an incoming item, if one cell in the hashed bucket has the matched ID, then we increment the pure counter in the cell by 1. If no cell has matched ID but one cell is empty, we update the empty cell with the ID of the incoming item, and set the pure counter to 1. Otherwise, we increment the substitution counter by 1. If the minimum pure counter in the hashed bucket is smaller than the substitution counter, the corresponding cell is replaced by the ID of the incoming item, and the substitution counter is reset. The previously recorded pair of the ID and the pure counter is passed to the USketch part. Otherwise, the incoming item is passed to the USketch part.

D. H-LETSketch

The Top-k part of the H-LETSketch is based on the HeavyGuardian algorithm [29]. The original HeavyGuardian algorithm is similar to the Frequent algorithm. The only difference is that, for Frequent, the counter is decreased by 1 each time the ID of the cell does not match the incoming item. In contrast, for HeavyGuardian, the counter is decreased by 1 with a probability b^{-C}, where b is a constant and C is the value of the counter. Therefore, when there is neither a matching cell nor an empty cell in the hashed bucket, H-LETSketch calculates the value of C as the difference between the minimum pure counter and the substitution counter. With a probability of b^{-C}, the substitution counter is incremented by 1. If the substitution counter exceeds the pure counter, the cell should be replaced by the incoming item, following the same procedure as in the F-LETSketch.

E. E-LETSketch

The Top-k part of the E-LETSketch is based on the Elastic algorithm [29]. In the original Elastic sketch, each bucket already contains two counters: one for pure frequency counting, and the other for substitution counting. The updating strategy in the Elastic sketch is that, if the ratio of the substitution counter to the pure counter exceeds a pre-defined threshold λ, the bucket will be replaced by the incoming item. The use of two counters in the Elastic sketch aligns naturally with our framework, and the design of flag bits in the Elastic sketch can be discarded in the E-LETSketch since it is not relevant to the goal of lossless extraction.

When there is neither a matching cell nor an empty cell in the hashed bucket, we increases the substitution counter by 1. If the ratio of the substitution counter to the minimum pure counter in the hashed bucket exceeds λ, the corresponding cell is replaced by the incoming item, and the previously recorded pair of (ID, pure counter) is sent to the USketch part. Otherwise, the incoming item is passed to the USketch part.

F. Discussion

We provide principles for choosing among the above-mentioned four versions the suitable one. The S-LETSketch has the strongest guarantee for frequent items extraction but is the slowest. For a data stream with A distinct items and N total items, with $min(|A|, \frac{N}{\epsilon F})$ counters in the Top-k part, the S-LETSketch ensures to monitor any items whose frequency is above F [28]. The other three versions, due to their probabilistic design, cannot provide fully guarantee.

The F-LETSketch is hardware-friendly. The H-LETSketch and the E-LETSketch requires complex operations like multiplication, division, and exponentiation. The F-LETSketch uses only the addition and comparison, and therefore is more suitable for hardware platforms such as programmable switches [38] and FPGA.

For the H-LETSketch and the E-LETSketch, they both requires parameter fine-tuning to achieve good performance. In the H-LETSketch, when the items come in the bursty patterns, the substitution counters are harder to increase due to the negative exponential growth probability, while in the E-LETSketch they can increase normally. Therefore, the H-LETSketch is more suitable for bursty data stream, while the E-LETSketch is more suitable for steady data stream.

For those situations that require switching between different versions, the time-window-based setting is a common solution. When it is needed to switch to a new version, we prepare the new version of sketch for the next time window, while the current time window runs on the old version. Once the preparation finished, we wait until the current time window ends and switch to the new version.

VI. Mathematical Analysis

In this section, we first analyze the error bound of G-sum estimation in §VI-A. Then in §VI-B we give the error bounds for query tasks on multiple data streams, including the estimation of inner-production, cosine similarity, and Jaccard similarity.

A. Error Bounds for G-sum

We divide the data stream f into two sub-streams, namely f_H, f_L. f_H contains the items recorded in the Top-k part, while f_L comprises of the remaining items. We use T to denote the set of IDs which are recorded in the Top-k part. Then f_H is a sub-stream of the items whose IDs are in T.

The USketch part provides accuracy guarantee for G-sum estimation, and the top-k algorithms provide accuracy guarantee for frequent items detection. Suppose USketch part provides $(1 \pm \epsilon_1)$-approximations for the G-sum w.p. more than $1-\delta_1$, and Top-k part provides $(1 \pm \epsilon_2)$-approximations for $f(x)$ where $x \in T$ w.p. more than $1-\delta_2$. We consider $\sum_{x\in T} g(f(x))$ and $\sum_{x\notin T} g(f(x))$ separately. For $\sum_{x\notin T} g(f(x))$, we use the query result in USketch part of $g(\cdot)$. The USketch part provide $(1 \pm \epsilon_1)$-approximation of $\sum_{x\notin T} g(f(x)) + \sum_{x\in T} g(f_L(x))$ w.p. more than $1-\delta_1$. As for $\sum_{x\in T} g(f(x))$, we use Top-k part for estimation, which is $\sum_{x\in T} g(f_H(x))$.

Theorem 1. *Suppose $g(\cdot)$ is a tractable, differentiable, and non-decreasing function s.t. $g'(t)$ is bounded by rt for any t and constant r. Then w.p. at least $1-\delta_1$, LETFramework provides estimation for $\sum_x g(f(x))$ with a relative error less than*

$$\epsilon_1+\delta_2+\max\{(1-\delta_2)(r\epsilon_2\frac{\sum_{x\in T} f^2(x)}{\sum_x g(f(x))}), \frac{(1-\delta_2)\epsilon_2^3\sum_{x\in T} f^3(x)}{\sum_x g(f(x))}\}$$

Proof. According to the analysis above, the estimation of $\sum_x g(f(x))$ of LETFramework is

$$(1+\eta)(\sum_{x\notin T} g(f(x)) + \sum_{x\in T} g(f_L(x))) + \sum_{x\in T} g(f_H(x))$$

where $|\eta| \le \epsilon_1$ w.p. at least $1-\delta_1$.
Then the relative error is less than

$$\begin{aligned} &\frac{|\eta(\sum_{x\notin T} g(f(x)) + \sum_{x\in T} g(f_L(x)))|}{\sum_x g(f(x))} \\ &+\frac{|\sum_{x\in T}(g(f_L(x)) + g(f_H(x)) - g(f(x)))|}{\sum_x g(f(x)))} \end{aligned} \tag{1}$$

Obviously, the first term of the expression above is less than $|\eta|$. For the second term, we have $g(f_L(x)) \le g(f(x))$ and $g(f_H(x)) \le g(f(x))$ since $g(\cdot)$ is a non-decreasing function. Note that Top-k part provides $(1 \pm \epsilon_2)$-approximations for $f(x)$ where $x \in T$ w.p. more than $1-\delta_2$, and $f(x) = f_L(x)+f_H(x)$. Thus, $\frac{f(x)-f_H(x)}{f(x)} = \frac{f_L(x)}{f(x)} \le \epsilon_2$ for $x \in T$ w.p. at least $1-\delta_2$. When $\frac{f_L(x)}{f(x)} \le \epsilon_2$, $0 \le g(f(x)) - g(f_H(x)) = g'(\xi f_H(x) + (1-\xi)f(x)) \cdot (f(x) - f_H(x)) \le r\epsilon_2 \cdot f^2(x)$, i.e. $-g(f_L(x)) \le g(f(x)) - g(f_L(x)) - g(f_H(x)) \le r\epsilon_2 f^2(x) - g(f_L(x))$. When $\frac{f_L(x)}{f(x)} > \epsilon_2$, $-g(f_L(x)) \le g(f(x)) - g(f_L(x)) - g(f_H(x)) \le g(f(x))$. Thus,

$$\begin{aligned} E(g(f(x))-&g(f_L(x)) - g(f_H(x))) \le \\ &(1-\delta_2)(r\epsilon_2 f^2(x) - g(f_L(x))) + \delta_2 g(f(x)) \end{aligned}$$

Due to the Lyapunov Central Limit Theorem, w.p. approximately to 1,

$$\begin{aligned} &\frac{\sum_{x\in T}(g(f(x)) - g(f_L(x)) - g(f_H(x)))}{\sum_x g(f(x))} \\ \approx\ &\frac{E(\sum_{x\in T}(g(f(x)) - g(f_L(x)) - g(f_H(x))))}{\sum_x g(f(x))} \\ \le\ &\frac{(1-\delta_2)r\epsilon_2\sum_{x\in T} f^2(x) - (1-\delta_2)\sum_{x\in T} g(f_L(x))}{\sum_x g(f(x))} \\ &+\frac{\delta_2\sum_{x\in T} g(f(x))}{\sum_x g(f(x))} \\ \le\ &\delta_2 + (1-\delta_2)(r\epsilon_2\frac{\sum_{x\in T} f^2(x)}{\sum_x g(f(x))}) \end{aligned}$$

And w.p. approximately to 1,

$$\begin{aligned} &\frac{\sum_{x\in T}(g(f(x)) - g(f_L(x)) - g(f_H(x)))}{\sum_x g(f(x))} \\ \approx\ &\frac{E(\sum_{x\in T}(g(f(x)) - g(f_L(x)) - g(f_H(x))))}{\sum_x g(f(x))} \\ \ge\ &-\frac{(1-\delta_2)\sum_{x\in T} g(\epsilon_2 f(x)) + \delta_2\sum_{x\in T} g(f(x))}{\sum_x g(f(x))} \\ \ge\ &-\frac{(1-\delta_2)\sum_{x\in T}(\epsilon_2 f(x))^3}{\sum_x g(f(x))} - \delta_2 \end{aligned}$$

Above all, w.p. at least $1-\delta_1$, LETFramework provides estimation for $\sum_x g(f(x))$ with a relative error less than

$$\epsilon_1+\delta_2+\max\{(1-\delta_2)r\epsilon_2\frac{\sum_{x\in T} f^2(x)}{\sum_x g(f(x))}, \frac{(1-\delta_2)\epsilon_2^3\sum_{x\in T} f^3(x)}{\sum_x g(f(x))}\}$$

□

B. Error Bound for queries on Multiple Data Streams

For the estimation of join size of two stream A, B, we suppose that we provide $(1 \pm \epsilon)$-approximations for $\sum_x f_A(x)^2, \sum_x f_B(x)^2$ and $\sum_x (f_A(x) + f_B(x))^2$, and errs w.p. less than δ. We have the following theorem.

Theorem 2. *Let $S = \sum_x (f_A(x)+f_B(x))^2, T = \sum_x (f_A(x)-f_B(x))^2, k = S/T$, the relative error is less than $2\epsilon(3k+1)/(k-1)$, and errs w.p. less than 3δ.*

Proof. The probability that LETFramework fails to estimate one of $\sum_x f_A(x)^2, \sum_x f_B(x)^2$ or $\sum_x (f_A(x) + f_B(x))^2$ with relative error smaller than ϵ is less than 3δ.
Consider the occasion that LETFramework provides $(1 \pm \epsilon)$-approximations for $\sum_x f_A(x)^2, \sum_x f_B(x)^2$ and $\sum_x (f_A(x) + f_B(x))^2$. The estimation of $\sum_x f_A(x) \cdot f_B(x)$ is no more than

$$(1+\epsilon)\sum_x (f_A(x) + f_B(x))^2 - (1-\epsilon)\sum_x (f_A(x)^2 + f_B(x)^2)$$

In this case, the relative error is

$$\frac{2\epsilon\sum_x(f_A(x)^2+f_B(x)^2+f_A(x)\cdot f_B(x))}{\sum_x f_A(x)\cdot f_B(x)}$$
$$=\frac{2\epsilon(\frac{3}{4}S+\frac{1}{4}T)}{\frac{1}{4}S+\frac{1}{4}T}=\frac{2\epsilon(3k+1)}{k-1}$$

Note that k reflects the similarity between A and B. The more similar A and B is, the larger k is.

Similar analysis can be applied to the lower bound of the estimation. Above all, we have that, our estimation provides $(1\pm\frac{2\epsilon(3k+1)}{k-1})$-approximation of $\sum_x f_A(x)\cdot f_B(x)$, and errs w.p. less than 3δ. □

The cosine similarity estimation is closely related to the the inner-production estimation. We can obtain the bound of relative error $\frac{\epsilon+\frac{2\epsilon(3k+1)}{k-1}}{1-\epsilon}$, which also have a probability less than 3δ to fail.

Finally, we discuss the error bound of Jaccard similarity.

Theorem 3. *Suppose that we provide $(1\pm\epsilon)$-approximations for $|S_A|,|S_B|$ and $|S_A\cup S_B|$, and errs w.p. less than δ. The absolute error of the estimation for Jaccard similarity is less than $\frac{4\epsilon}{1-\epsilon}$, and errs w.p. at most 3δ.*

Proof. The probability that LETFramework fail to estimate one of $|S_A|,|S_B|$ and $|S_A\cup S_B|$ with relative error smaller than ϵ is less than 3δ.

Consider the occasion that LETFramework provide $(1\pm\epsilon)$-approximations for $|S_A|$, $|S_B|$ and $|S_A\cup S_B|$. The upper bound of our estimation for $\frac{|S_A\cap S_B|}{|S_A\cup S_B|}$ is

$$\frac{1+\epsilon}{1-\epsilon}\cdot\frac{|S_A|+|S_B|}{|S_A\cup S_B|}-1$$

In this case, the absolute error is

$$\frac{2\epsilon}{1-\epsilon}\cdot\frac{|S_A|+|S_B|}{|S_A\cup S_B|}<\frac{4\epsilon}{1-\epsilon}$$

As for the lower bound of our estimation, the absolute error is less than $\frac{4\epsilon}{1+\epsilon}$.

Above all, the absolute error of the estimation for Jaccard similarity is less than $\frac{4\epsilon}{1-\epsilon}$, and errs w.p. less than 3δ. □

C. Discussion

Although Theorem 1 provides a general bound for a large group of the function $g(\cdot)$, some additional discussion and comparison are desirable for specific functions with great importance and typical features. Let us move back to the initial expression of the relative error, Expression 1, in our proof. It splits the error into two terms caused by the USketch and Top-k parts, respectively. Note that the Top-k part only needs to record relatively few frequent items. Thus, in a highly skewed real-world data stream, the error of the Top-k part can be nearly ignored compared with that of the USketch part. Therefore, we delved into analyzing of the error caused by the USketch part, which corresponds to the first term of Expression 1.

We bounded the term by $|\eta|$ to make the error bound compatible for tasks such as cardinality estimation, which uses function $g(x)=x^0$ in calculating G-sum. In this case, $\sum_{x\notin T}g(f(x))$ dominates $\sum_x g(f(x))$ for the reason that the Top-K part only keeps a few frequent items. Therefore, the error bound could not be better than just using a USketch part. The experiment results also shows that LETFramework performs worse than the USketch part in cardinality estimation and Jaccard similarity estimation based on the former task. However, when using other function $g(\cdot)$ with an order higher than 0, such as $x\log x$ and x^2, the error is much smaller than proved in real-world data stream processing. In these cases, we could suppose that $\xi=\max\{\frac{g(\epsilon t)}{g(t)}\}$ for a constant small amount ϵ is also a small amount. Moreover, in real-world data streams, a few frequent items make up a relatively large proportion of the total frequency. Considering this, if we suppose that $\frac{\sum_{x\notin T}g(f(x))}{\sum_x g(f(x))}=\tau$, then we have

$$\begin{aligned}&\frac{|\eta(\sum_{x\notin T}g(f(x))+\sum_{x\in T}g(f_L(x)))|}{\sum_x g(f(x))}\\
&\le|\eta|\cdot(\frac{\xi(1-\delta_2)\sum_{x\in T}g(f(x))+\delta_2\sum_{x\in T}g(f(x))}{\sum_x g(f(x))}\\
&\quad+\frac{\sum_{x\notin T}g(f(x))}{\sum_x g(f(x))})\\
&\le|\eta|\cdot(\xi(1-\delta_2)+\delta_2+\tau)\\
&\approx|\eta|\cdot\tau\\
&\le\epsilon_1\cdot\tau\end{aligned}$$

Thus, the error bound is multiplied by τ, which is smaller for $g(\cdot)$ with a higher change rate and data stream with higher skewness. The experiment results in §VII also verify this point, which show a relatively more remarkable accuracy improvement in second frequency moment estimation than in entropy estimation.

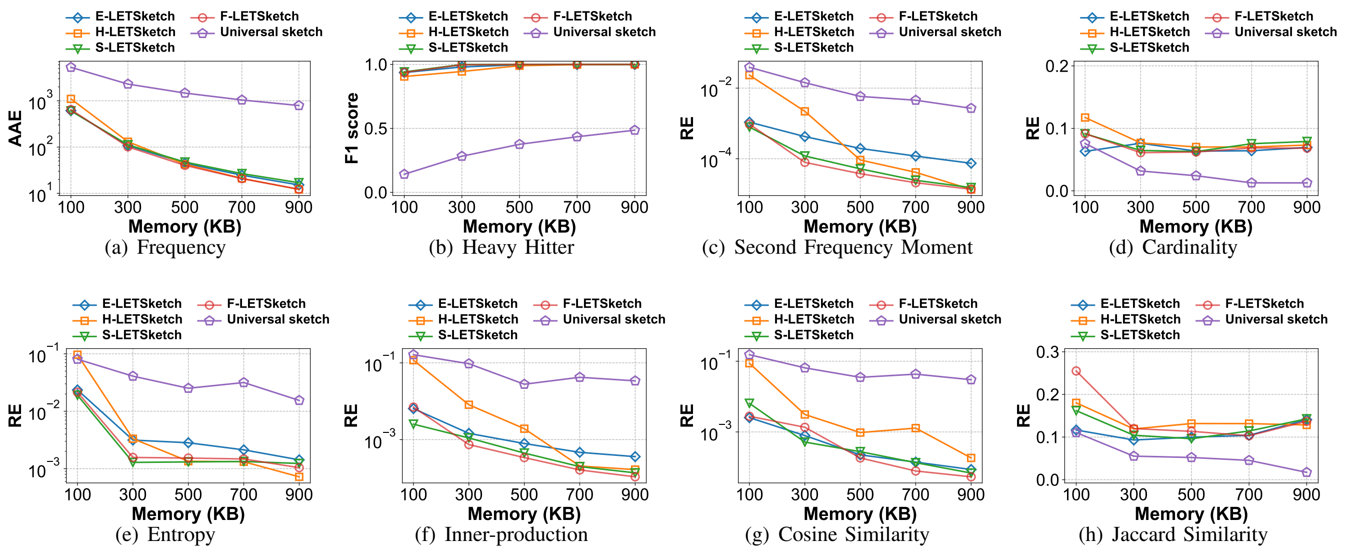
VII. Evaluation

We conduct experiments to answer the following questions:

- How does the LETFramework improve the accuracy and processing speed of the universal sketch? (§VII-B, §VII-C)
- How does the LETFramework compare to the baseline solution of using a collection of sketches to support multiple query tasks? (§VII-D)
- How does the parameter setting influence the performance of the LETFramework? (§VII-E)

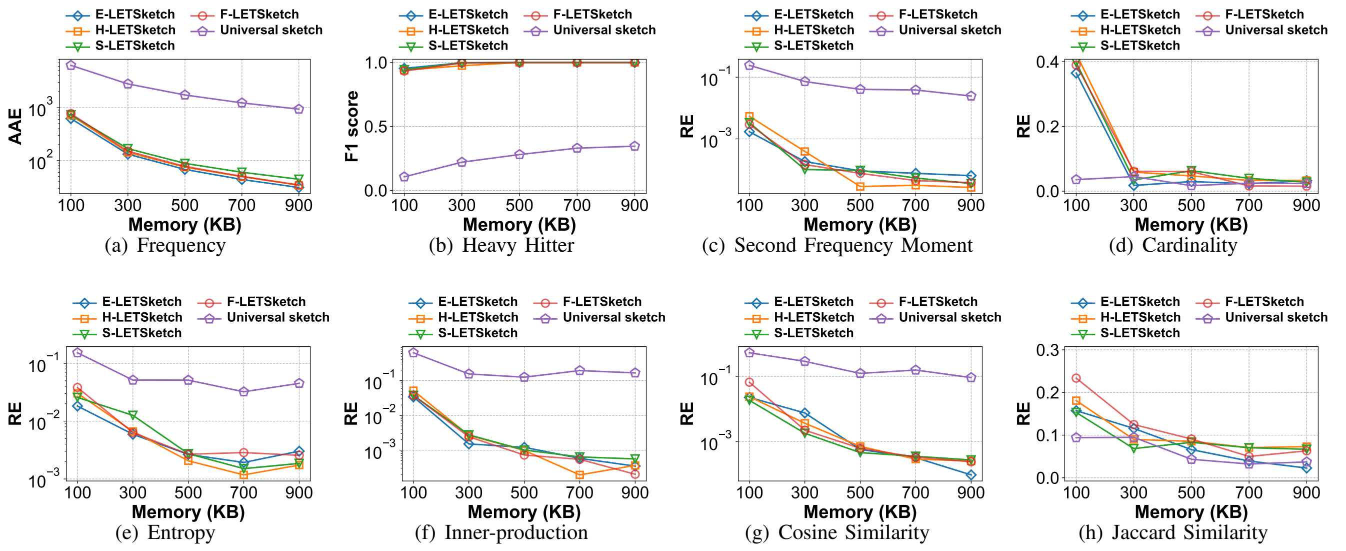
A. Experimental Setup

Implementation: We implement in C++ the four versions of the LETFramework. For the S-LETSketch, we use an optimized implementation to improve the insertion speed. The implementation uses an additional hash table to look up the cells and a double-linked list to sort the buckets and quickly locate the bucket with minimum value. We also deploy our sketches on Spark [39]. Integrating our solution to existing system is simple, which only requires to implement interfaces listed in §IV. All the experiments are conducted on a machine



(a) Frequency (b) Heavy Hitter (c) Second Frequency Moment (d) Cardinality (e) Entropy (f) Inner-production (g) Cosine Similarity (h) Jaccard Similarity

Fig. 3: Accuracy comparison between LETFramework and the universal sketch on the IP Trace dataset.



(a) Frequency (b) Heavy Hitter (c) Second Frequency Moment (d) Cardinality (e) Entropy (f) Inner-production (g) Cosine Similarity (h) Jaccard Similarity

Fig. 4: Accuracy comparison between LETFramework and the universal sketch on the WebDocs dataset.

with Intel Core i9-10980XE CPU @3.00GHz and 125GB DRAM memory.

Datasets:

(1) IP Trace Dataset (CAIDA) [40]. The dataset consists of one-hour-long IP traces collected anonymously in 2018. Each item is identified by a 4-byte source IP. In our experiments we use a one-minute part of the dataset which contains around 27M items and 85K distinct items.

(2) WebDocs Dataset [41]. The dataset is built from a collection of web documents and uses the terms as the items. It contains around 64M items and 33K distinct items, and each item is 4 bytes long.

(3) Synthetic Dataset. The dataset is generated following the Zipf distribution [42] with the skewness parameter 1.0. It contains 32M items, and each item is 4 bytes long.

Metrics:

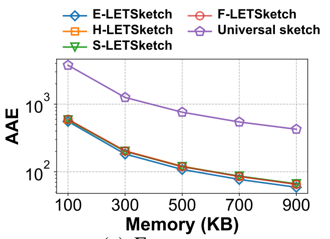
(1) Average Absolute Error (AAE): $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}|f_i-\hat{f}_i|$, where f_i is the true frequency of item e_i and $\hat{f}_i$ is the estimated frequency. We use AAE to quantify the error on the frequency estimation task.

(2) Relative Error (RE): $\frac{|g-\hat{g}|}{g}$, where g is the true value of the query task and the $\hat{g}$ is the estimated value. We use RE to quantify the error on query tasks that estimate aggregated statistics, such as entropy estimation and second frequency moment estimation.

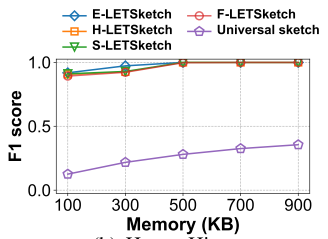
(3) F1 Score: $\frac{2\cdot PR\cdot RR}{PR+RR}$, where PR (precision rate) represents the ratio of the correctly reported items to the ground truth, and RR (Recall rate) represents the proportion of the correctly reported items among all the ground truth. We use F1 score to quantify the error on the heavy hitter detection task.

(4) Throughput: The number of inserted items in million per second (Mips).

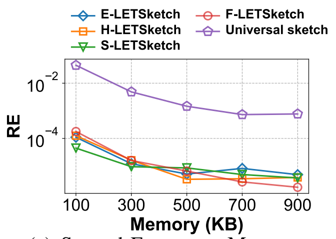
Settings: For query tasks on multiple data streams (inner-production, cosine similarity, Jaccard similarity), we divide the



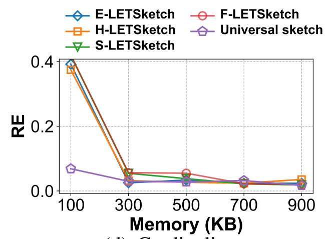
(a) Frequency



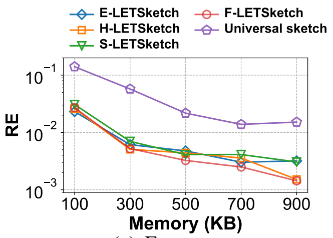
(b) Heavy Hitter



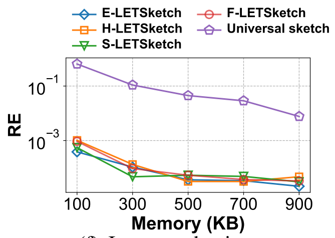
(c) Second Frequency Moment



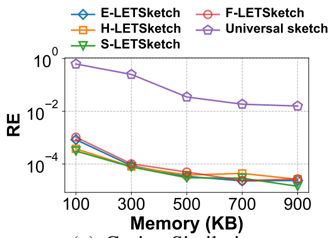
(d) Cardinality



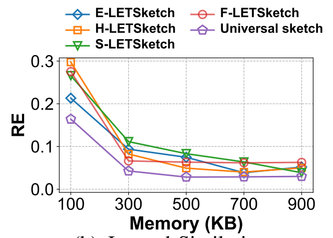
(e) Entropy



(f) Inner-production



(g) Cosine Similarity



(h) Jaccard Similarity

Fig. 5: Accuracy comparison between LETFramework and the universal sketch on the Synthetic dataset.

dataset into two parts with equal size as two data streams. By default, for the LETFramework we set the memory size ratio of the Top-k part to the USketch part as $6:4$. In the USketch part we set the memory size ratio of the Count sketches to the top-k arrays as $9:1$. We set the number of arrays d in the Count sketches of the USketch part to be 1, which achieves both good accuracy and throughput (see §VII-E). For the original universal sketch, we set $d=5$, because smaller d may lead to severe hash collision and result in low accuracy. Besides, $d=5$ is also the parameter choice in prior work [10].

B. *Accuracy Gain of the LETFramework*

In this section, we evaluate how the LETFramework improves the accuracy of the universal sketch. We compare the accuracy between the original universal sketch and the four versions of the LETFramework. We vary the memory consumption from 100KB to 900KB and conduct comparison experiments on eight different query tasks and three datasets.

Figure 3, 4, 5 show the experiment results on the IP Trace dataset, the WebDocs dataset, and the Synthetic dataset, respectively. For frequency estimation, LETFramework reduces the AAE by up to $65.22\times$. For heavy hitter detection, with 300KB memory, LETFramework achieves F1 score higher than 0.9976, while the F1 score of the universal sketch is lower than 0.2205. For second frequency moment estimation, LETFramework reduces the relative error by up to $1418.45\times$. For entropy estimation, LETFramework improves the accuracy by up to $31.70\times$. For inner-production estimation, LETFramework reduces the relative error by up to $2330.39\times$. For cosine similarity estimation, LETFramework improves the accuracy by up to $3216.02\times$. For the query tasks of the cardinality estimation and Jaccard similarity estimation, LETFramework achieves slightly lower but comparable accuracy. With 300KB memory, on the IP Trace dataset, the difference of the relative error between LETFramework and the universal sketch is less than 0.04 for cardinality estimation, and less than 0.06 for Jaccard estimation. The reason is that, LETFramework provides accurate estimation for frequent items. For G-sum functions such as $x^2, x\log x$, the estimation results are mainly contributed by the frequent items, and LETFramework can achieve higher accuracy. However, for query tasks such as cardinality estimation and Jaccard similarity estimation, the estimation is based on the G-sum function $g(x)=x^0$. Using additional memory to record the ID and the frequency of the frequent items is not efficient for these tasks. However, the accuracy gap is small. We also find that, four versions of the LETFramework basically achieves comparable accuracy of different tasks on all three datasets.

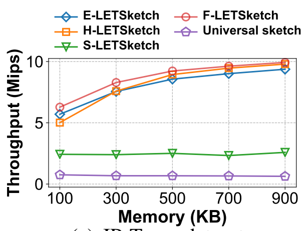
In summary, LETFramework achieves comparable accuracy on cardinality estimation and Jaccard similarity estimation, and much higher accuracy on all other tasks.

C. *Throughput Gain of the LETFramework*

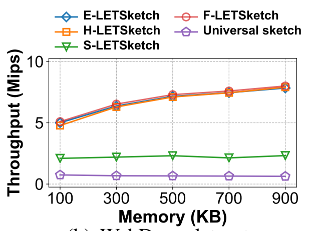
In this section, we evaluate how the LETFramework improves the insertion throughput of the universal sketch. We vary the memory from 100KB to 900KB and compare the insertion throughput on three datasets.

The experiment results are shown in Figure 6(a)-6(c). On the IP Trace dataset, with 900KB, LETFramework achieves insertion throughput up to 9.93Mips, while that of the universal sketch is 0.63Mips. On the WebDocs dataset, with 900KB, LETFramework achieves insertion throughput up to 8.00Mips, while that of the universal sketch is 0.63Mips. On the Synthetic dataset, with 900KB, LETFramework achieves insertion throughput up to 4.76Mips, while that of the universal sketch is 0.61Mips. The LETFramework achieves up to $15.73\times$ higher throughput compared with the universal sketch.

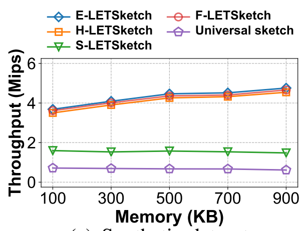
The insertion throughput is improved, because in the universal sketch, an item may go through multiple layers, which means multiple times of hash computation and memory accesses and can slow down the insertion. For LETFramework, it holds in the Top-k part the frequent items that usually comprise the main body of the data. Therefore, most items only reach the Top-k part and are quickly inserted, speeding up



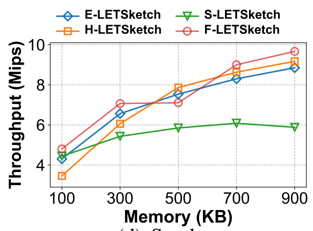
(a) IP Trace dataset



(b) WebDocs dataset

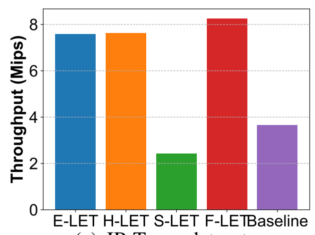


(c) Synthetic dataset

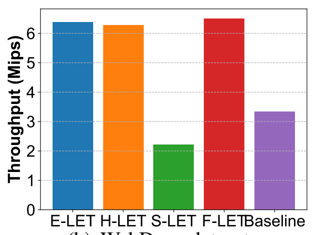


(d) Spark

Fig. 6: Throughput experiments on LETFramework.



(a) IP Trace dataset



(b) WebDocs dataset

Fig. 7: Throughput comparison with the baseline solution.

the insertion. We also find that the S-LETSketch is the slowest among four versions. This is because the S-LETSketch should update among all cells, while other three versions only update among the cells in the hashed bucket. Besides, we notice that when the memory size grows, the insertion throughput of the LETFramework increases. This is because we fix the memory ratio of the Top-k part to the USketch part. When the total memory size increases, the memory allocated to the Top-k part increases, and thus more frequent items are hold in the Top-k part, which is beneficial to the insertion throughput.

In summary, LETFramework achieves up to $15.73\times$ higher throughput compared with the universal sketch. Among the four versions, S-LETSketch is the slowest, while the throughput of the other three versions are close.

D. *Comparison with the Baseline*

For the eight query tasks that LETFramework supports, a baseline solution is to use a collection of several existing state-of-the-art (SOTA) sketches to address all these tasks. In this section, we use the collection of the following three sketches as the baseline to address the eight query tasks:

- Elastic sketch [10]: frequency estimation, heavy hitter detection, cardinality estimation, and entropy estimation.
- JoinSketch [4]: second frequency moment estimation, inner-production estimation, and cosine similarity estimation.
- MaxLogOPH [43]: Jaccard similarity estimation.

We compare the four versions of the LETFramework with the baseline solution. We fix the total memory to be the 300KB, and for the baseline solution each sketch in the collection uses 100KB of the memory. We conduct experiments on the two real world datasets, and compare the accuracy and the insertion throughput.

Table I and Table II show the accuracy comparison on the IP Trace dataset and the WebDocs dataset, respectively. On the IP Trace dataset, the LETFramework achieves comparable accuracy on the frequency estimation and the Jaccard similarity estimation and higher accuracy on all other tasks. For the heavy hitter detection, LETFramework can achieve the F1 score of 0.9987, while that of the baseline solution is 0.8277. Compared with the baseline solution, LETFramework achieves up to $95.19\times$, $41.89\times$, $47.86\times$, $2.83\times$, $3.41\times$ lower RE on the estimation of second frequency moment, cardinality, entropy, inner-production and cosine similarity. On the WebDocs dataset, the LETFramework achieves comparable accuracy on the frequency estimation and higher accuracy on all other tasks. For the heavy hitter detection, LETFramework can achieve the F1 score of 0.9986, while that of the baseline solution is 0.8454. Compared with the baseline solution, LETFramework achieves up to $431.77\times$, $38.00\times$, $66.98\times$, $2.43\times$, $3.54\times$, $1.43\times$ lower RE on the estimation of second frequency moment, cardinality, entropy, inner-production, cosine similarity and Jaccard similarity.

Figure 7 shows the comparison of the insertion throughput. The experimental results show that, except for the S-LETSketch, all the other three versions of the LETFramework have faster processing speed. The F-LETSketch is the fastest, which is $2.26\times$ faster on the IP Trace dataset and $1.94\times$ faster on the WebDocs dataset.

In summary, compared with a collection of SOTA dedicated sketches, LETFramework achieves better accuracy on most tasks.

E. *Parameter Setting*

In this section, we analyze the impact of two parameters: the memory ratio of the Top-k part, and the number of arrays d of the Count sketches in the USketch part. We conduct experiments on the IP Trace dataset with 1MB memory. Due to the space constraints, we only present the influence of accuracy on two query tasks: cardinality estimation and second frequency moment estimation. We believe the cardinality estimation and the second frequency moment estimation are representative tasks because they rely on the two extreme cases of the G-sum function (x^0 and x^2). We believe the findings on these two tasks provide valuable insights to the overall behavior of the LETFramework for other query tasks as well.

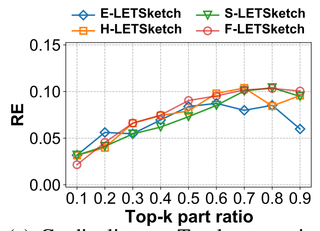
Figure 8(a) - 8(c) show the effect of the memory ratio of the Top-k part. As the memory ratio of the Top-k part increases from 0.1 to 0.9, the RE of the cardinality estimation increases, and the RE of the second frequency estimation decreases. This shows that, on the IP Trace dataset, the cardinality estimation mainly relies on the USketch part, while the second frequency moment estimation mainly relies on the Top-k part.

TABLE I: Accuracy comparison with the baseline solution on the IP Trace dataset.

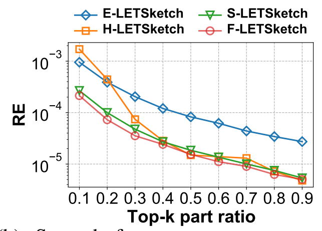
Query Tasks (Metric)	Frequency Estimation (AAE)	Heavy Hitter Detection (F1 score)	Second Frequency Moment Estimation (RE)	Cardinality Estimation (RE)	Entropy Estimation (RE)	Inner-production Estimation (RE)	Cosine Similarity (RE)	Jaccard Similarity (RE)
Baseline	**75.43**	0.8277	7.84×10^{-3}	0.71	0.133	2.12×10^{-3}	7.10×10^{-4}	**0.028**
S-LETSketch	112.68	**0.9987**	8.65×10^{-5}	**0.017**	$\mathbf{2.79 \times 10^{-3}}$	9.57×10^{-4}	5.09×10^{-4}	0.082
F-LETSketch	104.16	0.9986	$\mathbf{8.24 \times 10^{-5}}$	0.061	2.91×10^{-3}	$\mathbf{7.53 \times 10^{-4}}$	$\mathbf{2.08 \times 10^{-4}}$	0.043
H-LETSketch	128.61	0.9441	3.85×10^{-4}	0.037	3.85×10^{-3}	3.42×10^{-3}	5.19×10^{-3}	0.091
E-LETSketch	109.15	0.9784	4.14×10^{-4}	0.020	4.24×10^{-3}	1.80×10^{-3}	6.12×10^{-4}	0.070

TABLE II: Accuracy comparison with the baseline solution on the WebDocs dataset.

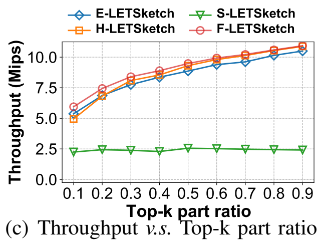
Query Tasks (Metric)	Frequency Estimation (AAE)	Heavy Hitter Detection (F1 score)	Second Frequency Moment Estimation (RE)	Cardinality Estimation (RE)	Entropy Estimation (RE)	Inner-production Estimation (RE)	Cosine Similarity (RE)	Jaccard Similarity (RE)
Baseline	**94.50**	0.8454	0.0176	0.92	0.114	3.51×10^{-3}	5.19×10^{-3}	0.063
S-LETSketch	167.82	**0.9986**	1.81×10^{-4}	**0.024**	4.84×10^{-3}	1.52×10^{-3}	$\mathbf{1.47 \times 10^{-3}}$	0.071
F-LETSketch	151.34	0.9837	1.06×10^{-4}	0.029	$\mathbf{1.70 \times 10^{-3}}$	2.92×10^{-3}	2.21×10^{-3}	0.047
H-LETSketch	143.79	0.9792	$\mathbf{4.09 \times 10^{-5}}$	0.025	6.70×10^{-3}	$\mathbf{1.44 \times 10^{-3}}$	2.79×10^{-3}	0.095
E-LETSketch	133.15	0.9939	4.11×10^{-4}	0.062	5.07×10^{-3}	1.84×10^{-3}	2.09×10^{-3}	**0.044**



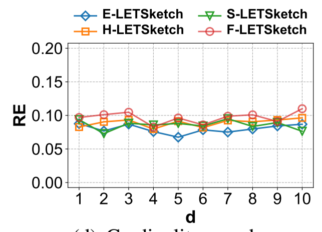
(a) Cardinality *v.s.* Top-k part ratio



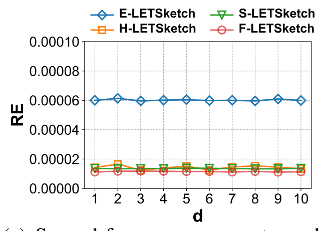
(b) Second frequency moment *v.s.* Top-k part ratio



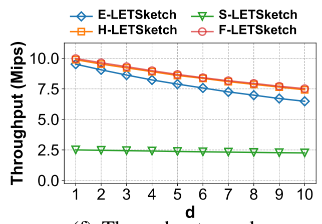
(c) Throughput *v.s.* Top-k part ratio



(d) Cardinality *v.s.* d



(e) Second frequency moment *v.s.* d



(f) Throughput *v.s.* d

Fig. 8: Parameter settings in the LETFramework.

The throughput of the S-LETSketch remains consistently low for different memory ratios. For the other three versions of the LETFramework, the throughput increases as the memory of the Top-k part grows. This indicates that for the three versions of the LETFramework, the Top-k part has higher processing speed than the USketch part. We choose the memory ratio of the Top-k part to 0.6, which strikes a good balance for accuracy over different query tasks and throughput.

Figure 8(d) - 8(f) show the effect of the number of arrays d of the Count sketches in the USketch part. The results show that, the choice of d has little impact on the accuracy of the LETFramework. This is because the frequent items have been extracted out, and the Count sketches can achieve high accuracy with small d. The throughput decreases when the d grows. Therefore, we choose $d = 1$ for LETFramework.

In summary, the impact of the top-k part ratio on the algorithm efficiency on different tasks is relatively complex, and we select 0.6 from an experimental perspective. The parameter d has low impact on accuracy, and higher d lead to lower throughput, therefore we choose $d = 1$.

F. Spark

In this section, we evaluate the performance of LETFramework on Spark with IP Trace dataset. Figure 6(d) shows the throughput of four versions of LETSketch. Basically, LETSketch achieves higher throughput when more memory is allocated, following the similar trend on CPU. With 900KB, LETSketch achieves up to 9.67 MIPS, showing that LETSketch can be efficiently implemented on the Spark platform.

VIII. Conclusion

In this paper, we propose the LETFramework to optimize the universal sketch. With the key technique of lossless extraction, LETFramework achieves high performance in practice while maintaining high fidelity. We introduce a unified methodology to transform existing top-k algorithms into the Top-k part of the LETFramework, and provide case studies on four top-k algorithms. We also extend the supported query tasks of the LETFramework by discussing how to support queries on multiple data streams, including inner-production estimation, cosine similarity estimation, and Jaccard similarity estimation. We conduct rich experiments, and the results show LETFramework outperforms both the universal sketch and a baseline solution that uses a collection of SOTA sketches to address multiple query tasks.

Acknowledgments

We thank all anonymous reviewers for their help in improving this paper. This work was supported in part by the National Key R&D Program of China (No. 2024YFB2906603), and in part by the National Natural Science Foundation of China (NSFC) (No. U20A20179, 62372009), research grant No. SH-2024JK29, and High Performance Computing Platform of Peking University.

References

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

[2] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*. Springer, 2002.

[3] 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.

[4] Feiyu Wang, Qizhi Chen, Yuanpeng Li, Tong Yang, Yaofeng Tu, Lian Yu, and Bin Cui. Joinsketch: A sketch algorithm for accurate and unbiased inner-product estimation. *Proceedings of the ACM on Management of Data*, 1(1):1–26, 2023.

[5] Graham Cormode and Minos Garofalakis. Sketching streams through the net: Distributed approximate query tracking. In *Proceedings of the 31st international conference on Very large data bases*, pages 13–24, 2005.

[6] Hazar Harmouch and Felix Naumann. Cardinality estimation: An experimental survey. *Proceedings of the VLDB Endowment*, 11(4):499–512, 2017.

[7] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[8] Renzhi Wu, Bolin Ding, Xu Chu, Zhewei Wei, Xiening Dai, Tao Guan, and Jingren Zhou. Learning to be a statistician: learned estimator for number of distinct values. *arXiv preprint arXiv:2202.02800*, 2022.

[9] Tong Yang, Junzhi Gong, Haowei Zhang, and etal. Heavyguardian: Separate and guard hot items in data streams. In *SIGKDD*, 2018.

[10] 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 *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[11] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2285–2293, 2021.

[12] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1129–1140, 2018.

[13] Chi Wang and Bailu Ding. Fast approximation of empirical entropy via subsampling. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 658–667, 2019.

[14] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin J Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *Proc. VLDB*, pages 454–465. VLDB Endowment, 2002.

[15] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 286–296, 2004.

[16] Rana Shahout, Roy Friedman, and Ran Ben Basat. Together is better: Heavy hitters quantile estimation. *Proceedings of the ACM on Management of Data*, 1(1):1–25, 2023.

[17] Graham Cormode, Minos Garofalakis, Shanmugavelayutham Muthukrishnan, and Rajeev Rastogi. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 25–36, 2005.

[18] Charles Masson, Jee E Rim, and Homin K. Lee. Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment*, 12(12):2195–2205, 2019.

[19] Nicholas JA Harvey, Jelani Nelson, and Krzysztof Onak. Sketching and streaming entropy via approximation theory. In *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 489–498. IEEE, 2008.

[20] Vladimir Braverman and Rafail Ostrovsky. Zero-one frequency laws. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 281–290, 2010.

[21] Vladimir Braverman and Rafail Ostrovsky. Generalizing the layering method of indyk and woodruff: Recursive sketches for frequency-based vectors on streams. In *International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 58–70. Springer, 2013.

[22] Tong Yang, Yang Zhou, Hao Jin, , and etal. Pyramid sketch: A sketch framework for frequency estimation of data streams. *VLDB Endowment*, 2017.

[23] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1449–1463, 2016.

[24] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[25] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In *Proceedings of the 11th international conference on World Wide Web*, pages 293–304, 2002.

[26] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2014.

[27] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[28] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, 2005.

[29] G. Lukasz, D. David, D. Erik D, L. Alejandro, and M. J Ian. Identifying frequent items in sliding windows over on-line packet streams. In *IMC*, 2003.

[30] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

[31] Antonis Manousis, Zhuo Cheng, Ran Ben Basat, Zaoxing Liu, and Vyas Sekar. Enabling efficient and general subpopulation analytics in multidimensional data streams. *Proceedings of the VLDB Endowment*, 15(11):3249–3262, 2022.

[32] Source code related to letframework. https://github.com/LETFramework/LETFramework/, 2023.

[33] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *Proc. ACM symposium on Theory of computing*, 1996.

[34] Piotr Indyk and David Woodruff. Optimal approximations of the frequency moments of data streams. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 202–208, 2005.

[35] Dina Thomas, Rajesh Bordawekar, Charu C Aggarwal, and S Yu Philip. On efficient query processing of stream counts on the cell processor. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 748–759. IEEE, 2009.

[36] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 63–78, 2015.

[37] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[38] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[39] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, pages 15–28, 2012.

[40] Anonymized internet traces 2018. https://catalog.caida.org/details/dataset/passive_2018_pcap. Accessed: 2022-6-29.
[41] Real-life transactional dataset. http://fimi.ua.ac.be/data/, 2004.
[42] David MW Powers. Applications and explanations of zipf's law. In *New methods in language processing and computational natural language learning*, 1998.
[43] Pinghui Wang, Yiyan Qi, Yuanming Zhang, and etal. A memory-efficient sketch method for estimating high similarities in streaming sets. In *SIGKDD*, 2019.