

SandwichSketch: A More Accurate Sketch for Frequent Object Mining in Data Streams

Zhuochen Fan, *Member, IEEE*, Ruixin Wang, Zihan Jiang, Ruwen Zhang, Tong Yang, *Member, IEEE*, Sha Wang, Yuhan Wu, Ruijie Miao, Kaicheng Yang, and Bui Cui, *Fellow, IEEE*

Abstract—Frequent object mining has gained considerable interest in the research community and can be split into frequent item mining and frequent set mining depending on the type of object. While existing sketch-based algorithms have made significant progress in addressing these two tasks concurrently, they also possess notable limitations. They either support only software platforms with low throughput or compromise accuracy for faster processing speed and better hardware compatibility. In this paper, we make a substantial stride towards supporting frequent object mining by designing SandwichSketch, which draws inspiration from sandwich making and proposes two techniques including the double fidelity enhancement and hierarchical hot locking to guarantee high fidelity on both two tasks. We implement SandwichSketch on three platforms (CPU, Redis, and FPGA) and show that it enhances accuracy by $38.4\times$ and $5\times$ for two tasks on three real-world datasets, respectively. Additionally, it supports a distributed measurement scenario with less than a 0.01% decrease in Average Relative Error (ARE) when the number of nodes increases from 1 to 16.

Index Terms—Frequent Item Mining; Frequent Set Mining; Data Streams; Sketch

I. INTRODUCTION

Frequent object mining in data streams is a fundamental but challenging problem in numerous areas, including databases and data mining [1]–[3], network measurement [4]–[6], machine learning [7]–[9], and network security [10]–[12], *etc.* It refers to finding objects whose frequencies/sizes are large or exceed a threshold and reporting their frequencies. In the scenario of large-scale data streams, sketch algorithms [13]–[15] have made significant advancements in terms of accuracy and processing speed while using a small amount of resources.

Typically, the object comprises the item and the set, so frequent object mining can be categorized into frequent item mining and frequent set mining depending on the type of object. For frequent item mining, there is no denying the

significance of discovering frequent items in data streams. The existing sketch solutions [16]–[20] have made considerable strides in this respect. As for frequent set mining, a set is made up of multiple items, and its frequency corresponds to the aggregate of item frequencies within that set according to subset sum estimation [21], which is the theoretical foundation for frequent set mining. It has applications across a range of areas including machine learning [22], hierarchical aggregation [23], Distributed Denial-of-Service (DDoS) attack detection [24], arbitrary partial key query [25] and database query optimization and join size estimation [26]. Moreover, it facilitates merging operations, which makes it appropriate for distributed scenarios. Given the growing importance of frequent item and set mining in recent years, we aim to concentrate on sketch algorithms capable of effectively managing both tasks concurrently.

An ideal system for frequent object mining should meet three requirements: [R1] versatility (support frequent item mining and frequent set mining simultaneously), [R2] fidelity (provide high accuracy guarantee for the above two tasks) and [R3] compatibility (have strong compatibility on software and hardware platforms, *e.g.*, CPU, Redis [27], FPGA [28]). These requirements are motivated by practical needs in real-world applications. First, versatility is essential because many applications (*e.g.*, network monitoring, database query optimization, *etc.*) do require both tasks at the same time. Second, fidelity is crucial to ensure accurate results, as even small errors can be accumulated and lead to wrong decisions in applications. Third, compatibility is necessary to deploy these algorithms in diverse environments, where efficient resource utilization and scalability are paramount.

Regrettably, as highlighted in Table I, while state-of-the-art solutions such as Unbiased Space-Saving (USS) [29] and CocoSketch [25] are competent in addressing both tasks at the same time, they each have their own inherent limitations. USS proposes variance minimization and presents the theoretical optimum for frequent set mining. However, its actual implementation uses a double-linked list, which cannot achieve the theoretical optimum due to the large amount of memory waste caused by pointers. Additionally, it suffers from a long update delay that hampers its ability to operate at high speed, making it challenging to deploy on high-speed hardware platforms like FPGA. CocoSketch proposes stochastic variance minimization and circular dependency removal in an attempt to improve the update speed and hardware compatibility of USS, but this comes at the expense of some accuracy. These limitations highlight the need for a solution that simultaneously achieves

Zhuochen Fan is with the Department of Strategic and Advanced Interdisciplinary Research, Pengcheng Laboratory, Shenzhen, Guangdong 518055, China (email: fanzhch@pcl.ac.cn).

Zhuochen Fan, Ruixin Wang, Zihan Jiang, Ruwen Zhang, Tong Yang, Yuhan Wu, Ruijie Miao, and Kaicheng Yang are with the State Key Laboratory of Multimedia Information Processing, School of Computer Science, Peking University, Beijing 100871, China. Zhuochen Fan, Ruixin Wang, and Zihan Jiang are co-first authors with equal contributions, and Tong Yang is the corresponding author. (e-mail: {fanzc, zrw, yangtong, yuhan.wu, miaoruijie, ykc}@pku.edu.cn, wang.ruixin@alumni.pku.edu.cn, jumbo0715@stu.pku.edu.cn).

Sha Wang is with the College of Computer Science and Technology, National University of Defense Technology, Changsha 410073, China (e-mail: ws0623zz@163.com).

Bin Cui is with the Key Laboratory of High Confidence Software Technologies (MOE) & School of Computer Science, Peking University, Beijing 100871, China. (e-mail: bin.cui@pku.edu.cn).

versatility, fidelity, and compatibility.

TABLE I
OUR WORK V.S. PRIOR SOLUTIONS.

Solution	Versatility	Fidelity	Compatibility
USS	✓	✗	✗
CocoSketch	✓	✗	✓
SandwichSketch	✓	✓	✓

In this work, we present **SandwichSketch**, a sketch-based measurement system for frequent object mining. SandwichSketch conceptually resembles sandwich making. On the one hand, similar to a sandwich that comprises two bread slices with a layer of filling in between, SandwichSketch features a comparison replacement layer (Stage 2) flanked by two probability replacement layers (Stage 1 and Stage 3). On the other hand, just like the process of making a sandwich where the bottom bread layer is prepared first, then the fillings are added, and finally the top bread layer is placed, SandwichSketch processes each inserted item in a similar sequential manner. Based on the sandwich making philosophy, we propose two main techniques as follows aimed at fulfilling our design requirements.

1) Double Fidelity Enhancement. Current solutions like USS and CocoSketch utilize an unbiased probability replacement strategy to avoid substantial error accumulation in frequent set mining, aiming for unbiased estimation. We inherit this strategy and equip SandwichSketch with an unbiased part as Stage 3. Despite this, we find that distinguishing between frequent and infrequent items can further enhance estimation fidelity. Specifically, we introduce a heavy part (Stage 1 and Stage 2) before the unbiased part (Stage 3) to better maintain frequent items. When a new item inserts, it will first be processed in the heavy part and frequent items will remain here as much as possible. Other infrequent items will be replaced in the heavy part and kicked to the unbiased part. Moreover, we propose the pure replacement method to handle item replacements in the heavy part to increase fidelity while preserving unbiasedness. As the heavy part is applied to both frequent item mining and frequent set mining, the fidelity of both tasks is simultaneously improved. 2) Hierarchical Hot Locking. Common methods for maintaining frequent items in the heavy part are the probability replacement strategy and the comparison replacement strategy. However, using only one strategy has drawbacks. The probability strategy may let frequent items be kicked out, and the comparison strategy is sensitive to the order of item appearance. Therefore, in the heavy part, we use the probability strategy in Stage 1 to buffer frequent items and the comparison strategy in Stage 2 to precisely retain them. In this way, frequent items are better maintained in the heavy part, ensuring accurate mining results.

In conclusion, combining double fidelity enhancement with hierarchical hot locking satisfies requirements [R1] and [R2]. Furthermore, operating in a pipeline format enables SandwichSketch to conveniently meet the requirement [R3].

We implement SandwichSketch prototypes on various software (*e.g.*, CPU and Redis) and hardware (*e.g.*, FPGA) platforms. Our evaluation is under three tasks (heavy hitter detection, heavy change detection, and hierarchical heavy

hitter detection). For frequent item mining, the ARE of SandwichSketch achieves 31.7 and 38.4 improvements on CAIDA and Web Page datasets, respectively. For frequent set mining, the ARE of SandwichSketch is 5 and 4.5 better than those of CocoSketch and USS, respectively, and is just 1.2 worse than the theoretical optimum. Additionally, SandwichSketch achieves 0.4 Mops and 230 Mops on Redis and FPGA, respectively. In addition, our experiments also verify that SandwichSketch is suitable for distributed scenarios. When the number of nodes increases from 1 to 16, the decreases in F1 Score and ARE of SandwichSketch are less than 0.37% and 0.01%, respectively. We have open-sourced code of SandwichSketch and other baseline algorithms on GitHub [30].

II. BACKGROUND AND RELATED WORK

A. Problem Definitions

Many data analysis problems involve the following SQL statement including some filters and group by conditions. Frequent object mining including frequent item mining and frequent set mining can also be transformed into this schema depending on whether there is group by clause.

```
SELECT di mensi ons, sum(metri c)
FROM table
WHERE filters
GROUP BY di mensi ons
```

Before the sketch measurement starts, we first define unit dimensions of measurement. For frequent item mining, there is no group by clause and we can identify frequent items or heavy hitters over unit dimensions through filters. For frequent set mining, there are arbitrary group by dimensions called set dimensions except for a small restriction that set dimensions cannot be finer granularity than unit dimensions. We can aggregate a number of items to build a set by utilizing the group by clause, use sum function to acquire the total metric of each set and define filters to find frequent sets.

B. Applications

It has many applications of frequent set mining including machine learning [22], hierarchical aggregation [23], DDoS attack detection [24], arbitrary partial key query [25], database query optimization and join size estimation [26]. In addition, it supports merge operations, making it ideal for distributed scenarios. In such case, we can initially measure the data subset in each node using a sketch, and subsequently merge all sketches to respond to queries over all data.

To illustrate, let's consider the problem of hierarchical aggregation [23] in the context of network traffic data. IP addresses are organized hierarchically to differentiate between various subnets. Each subnet (*e.g.*, 172.168.10.) is a set of many IP addresses (*e.g.*, 172.168.10.0, . . . , 172.168.10.255). We can monitor the flow size of IP addresses and perform the group by clause to acquire the size of arbitrary subnet by summing all sizes of IP addresses sharing the same IP prefix.

It can also be readily applied to multidimensional measurement, where it's necessary to measure and query the metric

across various dimensions. For example, in DDoS attack detection, it can be challenging to identify a few dimensions which need to be measured in advance [24], [31]–[37]. Thus, it becomes necessary to track numerous potential flow dimensions such as 5-tuple, SrcIP/DstIP, and their arbitrary prefixes [24]. By applying frequent set mining, we can predefine the broadest dimension range called the unit dimension (*e.g.*, 5-tuple), prior to measurement, and measure item frequency over this unit dimension. Post the measurement window, we can query over any set dimension (*e.g.*, SrcIP/DstIP) which is a part of unit dimension by aggregating all related items.

C. Existing Solutions and Limitations

Although recent efforts including Unbiased Space-Saving (USS) [29] and CocoSketch [25] have taken a significant step toward frequent object mining, they still have some unacceptable limitations. Apart from these two, there are almost no outstanding works on mining frequent items and frequent sets simultaneously: They are basically studied separately, with the former dominating. Next, we will introduce the USS and CocoSketch in detail, and briefly introduce other works.

Unbiased Space-Saving: Unbiased Space-Saving (USS) [29] firstly proposes the technique of variance minimization to achieve unbiased estimation with minimum estimation variance. It consists of an array with (*key, value*) buckets. For an incoming item with key e and value w , if e is already recorded in a bucket, USS will increment the counter of this bucket by w and the variance will not increase. Otherwise, USS will find the bucket with minimum size C_{min} , increase it by w , and replace the key in this bucket with e with the probability $\frac{w}{C_{min}+w}$. Due to the variance minimization, USS achieves the theoretical optimum for frequent set mining. However, it is obvious that USS must scan all buckets to check whether there is a matched bucket or find the bucket with minimum size during each update process. The time complexity of update process is $O(n)$ where n is the number of buckets (*e.g.*, the scale of 10^4) in the USS. Such update strategy makes it hard to run with high throughput and deploy on the high-speed hardware such as FPGA. In practice, the implementation of USS often uses a double-linked list to maintain the minimum bucket, which wastes lots of memory on pointers and results in lower accuracy.

CocoSketch: CocoSketch consists of d arrays with same number of (*key, value*) buckets. In order to overcome the problem of high update overhead of USS, CocoSketch [25] proposes the stochastic variance minimization. Instead of scanning all buckets, it randomly selects d (*e.g.*, 1, ..., 4) hashed buckets. For an incoming item with key e and value w , if e is already recorded in one of d hashed buckets, CocoSketch will increment the counter of this bucket by w . Otherwise, CocoSketch will find the one with minimum size C_{min} among d hashed buckets, increase it by w , and replace the key in this bucket with e with the probability $\frac{w}{C_{min}+w}$. In addition, for improving the hardware compatibility, CocoSketch introduces circular dependency removal technique, which independently runs stochastic variance minimization on each array and separates the update of key and value into two stages for each

array. Finally, it reports the median value of d hashed buckets as the estimated value. Although CocoSketch is the state-of-the-art (SOTA) algorithm for frequent set mining, it still has some drawbacks. Compared with variance minimization, stochastic variance minimization sacrifices some accuracy to some extent because it looks for the local minimum instead of the global minimum in the insertion process. In addition, circular dependency removal will further weaken the accuracy guarantee. Because each array is updated independently, it loses the step of finding minimum among d hashed buckets which results in larger estimation variance.

Others: 1) Frequent item mining. The research community has generally adopted sketch-based solutions for finding frequent items (and related heavy hitters and heavy changes), but even the SOTA ones [38]–[42] do not consider frequent sets. 2) Frequent set mining. Hyper-USS [43] is the SOTA sketch scheme for this problem. Notably, it targets multi-attribute subset queries where items carry multiple numerical values. Ours, as well as USS and CocoSketch, focuses on single-attribute frequent object mining.

III. OVERVIEW

A. Problem Scope

Requirements: SandwichSketch has three design requirements:

- [R1] Versatility: SandwichSketch should support frequent item mining and frequent set mining.
- [R2] Fidelity: SandwichSketch should provide high accuracy guarantee for above two tasks.
- [R3] Compatibility: SandwichSketch should have strong compatibility on both software (*e.g.*, CPU and Redis [27]) and hardware (*e.g.*, FPGA [28]) platforms with high throughput.

SandwichSketch Architecture: Prior to initiating the measurement, it's imperative to first define unit dimensions, which can cover arbitrary set dimensions we aim to query in the future. Figure 1 illustrates the SandwichSketch architecture, and the detailed processing steps are as follows:

Step 1: In the Data plane, we deploy a sketch to measure unit metric over unit dimensions. Whenever an item is coming, the sketch will be updated based on the insertion algorithm (see details in §IV-B).

Step 2: At the end of each measurement window, SandwichSketch's control plane will scan the whole structure only once, aggregate estimated results of the sketch and build both Item_Query_Table and Set_Query_Table (see details in §IV-C).

Step 3: Utilizing the query table from the control plane, operators can employ SQL statements to query the estimated metric of any dimension in the query front-end (see details in §IV-C).

B. Sandwich Making Philosophy

The key philosophy of SandwichSketch is called *Sandwich Making*. The reason why we call it is that it has the following two figurative characteristics:

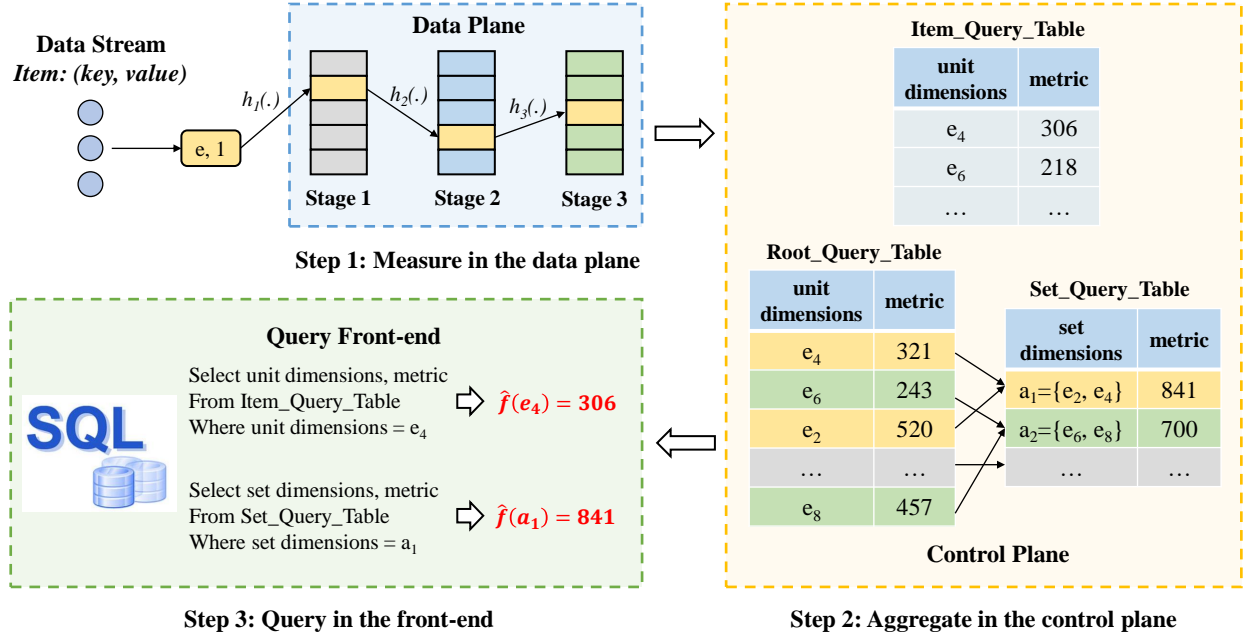


Fig. 1. Overview of the SandwichSketch architecture, which composes of data plan, control plane and query front-end.

Analogous to the structure of a sandwich with a filling layer between two slices of bread, the SandwichSketch structure also comprises two probability replacement layers (Stage 1 and Stage 3) and a comparison replacement layer (Stage 2) between them.

Just like the sandwich making process, first making the bottom layer of bread, then placing the filling ingredients and finally covering the top layer of bread, SandwichSketch also processes the inserted item in three parts in sequence.

In line with the sandwich making philosophy, we propose two primary techniques to better align with our design requirements.

1) Technique I: Double Fidelity Enhancement:

The subset sum estimation theory forms the theoretical foundation for frequent set mining. This theory necessitates that the estimated unit metric of unit dimensions is unbiased to avoid significant error accumulation during the group by process. Generally, existing solutions like USS and CocoSketch employ an unbiased probability replacement strategy to ensure unbiased estimation, and we continue to apply this strategy. Specifically, we designate one array as the unbiased part (Stage 3). When an item (e, w) is inserted to the bucket $(ID, count)$ of the unbiased part, it will first increment the counter by w and then replace ID with e with the probability $\frac{w}{count+w}$.

However, we believe that there is potential for further enhancement of estimation fidelity. We find that separating frequent and infrequent items is an effective technique to boost the accuracy for frequent object mining. Therefore, we consider adding a heavy part (Stage 1 and Stage 2) which is composed of a number of $(ID, count)$ bucket arrays before the unbiased part to better maintain frequent items. Specifically, during the insertion process, a new item will first be processed in the heavy part, in which frequent items will stay here with the greatest probability and infrequent items will be replaced and kicked to the unbiased part.

To ensure unbiasedness of the combination of the heavy part and the unbiased part, we introduce the pure replacement method to handle item replacement in the heavy part. Specifically, when item replacement occurs, if replacement condition holds, the new item will occupy the whole bucket and the ID and count of original item will be evicted to next array. Otherwise, the new item will be directly inserted to the next array. Essentially, each bucket in the heavy part is a pure bucket as the count of this bucket entirely belongs to the item identified by ID in this bucket. Given that the buckets in the heavy part are pure buckets and the unbiased part achieves an unbiased estimation, we can ensure that the estimated value across two parts is unbiased, as proven in §V-B.

For frequent item mining, we depend on the heavy part to report the estimation results because the majority of memory is allocated here, ensuring that nearly all frequent items are stored here. Furthermore, given that the buckets in the heavy part are pure buckets, the estimated value of each item is very close to the real value. As for frequent set mining, we employ both the heavy part and the unbiased part to achieve an unbiased estimation and answer the query. Compared with CocoSketch and USS, which provide an entirely unbiased value, the majority of our unbiased value is pure value, and a small fraction is an unbiased value. Therefore, our estimated value is expected to be more accurate. By introducing the heavy part and the pure replacement method, we can augment the fidelity of both tasks simultaneously.

2) Technique II: Hierarchical Hot Locking:

To maintain frequent items in the heavy part, while ensuring hardware compatibility, common approaches include (1) the probability replacement strategy (items with higher frequency have a higher chance of staying), or (2) the comparison replacement strategy (items with higher frequency are certain to stay). Unfortunately, the use of only one of these strategies is not effective (as proven in §VI-B). If only the probability

replacement strategy is used, it is possible for larger items to be kicked to the unbiased part, and the goal of retaining frequent items may not be fully achieved. If only the comparison replacement strategy is used, the strategy is sensitive to the chronological order in which frequent items appear. The frequency of each item is small when it first occurs, thus frequent items that appear later cannot be retained effectively through frequency comparison. As a result, in the heavy part, we initially use the probability replacement strategy (in Stage 1) to buffer frequent items, followed by the comparison replacement strategy (in Stage 2) to accurately lock frequent items.

Specifically, we come up with a pure probability replacement strategy in Stage 1 and a comparison replacement strategy in Stage 2. In Stage 1, any inserted item whose key does not match the key in the bucket will occupy this bucket with the probability $\frac{w}{count + w}$. The value of inserted item is higher, the probability of occupying this bucket is higher. In Stage 2, the replacement will occur when the key does not match and $w > count$, which means that the larger item will definitely occupy the bucket. For a new item, there is a certain probability that it will first stay in Stage 1. If it ends up being a frequent item, it will stay in Stage 1 with high probability. Even if a frequent item is kicked to Stage 2, it will be maintained in Stage 2 with high probability due to higher frequency. If it turns out to be an infrequent item, it will probably not stay in the heavy part and will eventually be inserted into the unbiased part.

In summary, double density enhancement and hierarchical hot locking achieve the goal of [R1] and [R2] well.

C. Hardware Compatibility

Strong hardware compatibility is one of our major design goals. Using FPGA as a hardware deployment example, we analyze factors that influence its performance as follows:

First, the ability of an algorithm to run in a pipeline is an essential factor for improving processing speed. On one hand, if the update algorithm follows a unidirectional workflow, it can more easily facilitate the program to run in a pipeline. On the other hand, if the steps of the algorithm are highly coupled, it becomes difficult to decouple them into different modules. Strong independence between different modules also aids in running algorithms in a pipeline.

Second, computational complexity also has significant influence on processing throughput. If the combination or timing logic to be completed in one clock cycle is too complex, the processing delay will increase, resulting in a lower clock frequency. Therefore, the update complexity of the algorithm should be as simple as possible.

The sandwich making philosophy helps SandwichSketch facilitate the hardware performance. First of all, similar to the process of sandwich making, the update of SandwichSketch follows a unidirectional workflow. In addition, just like the structure of sandwich, SandwichSketch consists of three independent modules, which can be decoupled easily. Moreover, update operations including assignment, counter increment, numerical comparison and probability calculation are not sophisticated so that the computational complexity is low. More

TABLE II
SYMBOLS AND NOTATIONS.

Symbol	Description
e	the item
$f(e)$	the real value of item e
$\hat{f}(e)$	the estimated value of item e
d	the number of arrays in SandwichSketch
l	the number of buckets in one array
r	the memory ratio of Stage 1 and Stage 2
$h_i(\cdot)$	the hash function corresponding to the i th array
$A_i[j]$	the j th bucket in the i th array
$A_i[j]:ID$	the key held in $A_i[j]$
$A_i[j]:count$	the value held in $A_i[j]$

importantly, the hardware version of SandwichSketch is the same as the software, which has no accuracy loss for hardware compatibility. In a word, SandwichSketch can be implemented in a pipeline on the hardware platform (e.g., FPGA) with high throughput and accuracy (see details in §VI-F), which satisfy the design requirements [R3].

IV. DETAILED DESIGN

A. Data Structure

We first list the frequently used symbols in Table II. As shown in Figure 2, SandwichSketch consists of the heavy part (including Stage 1 and Stage 2) and the unbiased part (Stage 3). Each stage contains d_1 , d_2 and 1 arrays respectively ($d = d_1 + d_2 + 1$; $r = \frac{d_1 + d_2}{d}$). Each array maintains l buckets. Each bucket records a particular item key and its estimated value. Let $A_i[j]$ ($1 \leq i \leq d$; $1 \leq j \leq l$) be the j th bucket of the i th array, and $A_i[j]:ID$ and $A_i[j]:count$ be its key and value. The d arrays are associated with d independent hash functions $h_1(\cdot); \dots; h_d(\cdot)$.

B. Insertion Operation

We regard each inserting item in each array as a pair of $(e; w)$, where e is the item key and w is the item value. When an item inserts into SandwichSketch, it will be processed in the Stage 1, Stage 2 and Stage 3 in sequence. In the array A_i ($1 \leq i \leq d$), according to the information of $A_i[h_i(e)]$, there are three cases for these three parts as follows:

Case 1: If e matches $A_i[h_i(e):ID]$, we will increment $A_i[h_i(e):count]$ by w and return.

Case 2: If $A_i[h_i(e)]$ is empty, we will insert the item into the empty bucket and return, which sets $A_i[h_i(e):ID]$ to e and sets $A_i[h_i(e):count]$ to w .

Case 3: Otherwise, these three parts will use different strategies to process the insertion. There are three sub-cases:

(a) For Stage 1: We use pure probability replacement strategy to keep larger item in the bucket. With the probability $\frac{w}{A_i[h_i(e):count + w]}$, we replace $(A_i[h_i(e):ID; A_i[h_i(e):count])$ with $(e; w)$ and the item $(A_i[h_i(e):ID; A_i[h_i(e):count])$ will be evicted to the next array. Otherwise, we directly insert item $(e; w)$

into the next array if the probability condition is not satisfied.

Fig. 2. Insertion example in SandwichSketch (with $d_2 = 1$).

- 2) For Stage 2: We use comparison replacement of bucket correspond to unit dimensions and unit metric strategy to store larger item in the bucket. Where respectively, w is larger than $A_i[h_i(e)]:count$, we replace $(A_i[h_i(e)]:ID; A_i[h_i(e)]:count)$ with $(e; w)$ and the item $(A_i[h_i(e)]:ID; A_i[h_i(e)]:count)$ will be evicted to the next array. Otherwise, we directly insert it into the next array.
- 3) For Stage 3: We use unbiased probability replacement strategy to achieve unbiased estimation. Specifically, we first increase $A_i[h_i(e)]:count$ by w directly. And then with probability $\frac{w}{A_i[h_i(e)]:count + w}$, we replace $A_i[h_i(e)]:ID$ with e if the probability condition is satisfied.

When the item is inserted into the next array, we process it according to the above strategies. In this way, an inserted item will be processed in these three parts in sequence so that the update process can be pipelined, which is friendly to pipelined hardware platforms.

Example (Figure 2): We set $d_1; d_2 = 1$ as an example. To insert item $(e_3; 1)$, we first use hash function to map the item to the bucket with content $(e_4; 8)$ in the 1st array, which belongs to Stage 1. Because $e_3 \notin e_4$, we try to replace item with the probability $\frac{w}{A_i[h_i(e)]:count + w} = \frac{1}{9} = 11.1\%$. Assumed that the probability condition is satisfied, we replace $(e_4; 8)$ with $(e_3; 1)$ and $(e_4; 8)$ will be evicted to 2nd array, which belongs to Stage 2. And then, we map $(e_3; 1)$ to the $(e_7; 6)$ bucket and find that $e_3 \notin e_7$. Therefore, we replace $(e_7; 6)$ with $(e_4; 8)$ due to $8 > 6$ and $(e_7; 6)$ will be evicted to 3rd array, which belongs to Stage 3. In this part, we also map $(e_3; 1)$ to the $(e_9; 3)$ bucket and find the item key is not matched. So we first increase the value by 6 (from 3 to 9) and finally replace e_9 with e_7 with the probability $\frac{w}{A_i[h_i(e)]:count + w} = \frac{6}{9} = 66.7\%$. To insert item $(e_2; 1)$, we find $e_2 \notin e_8$ in the mapped bucket so that we try to replace $(e_8; 2)$ with the probability $\frac{1}{3} = 33.3\%$. Assumed that replacement fails, $(e_2; 1)$ will be directly inserted into 2nd array. In the mapped bucket in this array, we find that the item key is matched and we increment the value by 1 (from 4 to 5). Up to this point, the whole update process is finished.

C. Query Operation

In the query process, we can regard each bucket in the sketch as one row in the sketch table. The key field and value hierarchical aggregation problem as an example. Suppose that

Query for frequent item mining: According to the algorithm design, the query for frequent item mining mainly relies on Stage 1 and Stage 2. After a measurement period, we will scan Stage 1 and Stage 2 only once, aggregate unit metrics of same items over unit dimensions and build a query table named Item_Query_Table. If a query item does not exist in the query table, we regard its estimated metric as 0. We provide a front-end and the user can directly use SQL statement to acquire the estimated metric of each item over unit dimensions. Through setting filters, we can find frequent items or heavy hitters. The SQL statement is as follow:

```
SELECT unit_dimensions,metric
FROM (SELECT unit_dimensions,
             sum(unit_metric) as metric
FROM sketch_table
GROUP BY unit_dimensions
) Item_Query_Table
WHERE filters
```

Query for frequent set mining: The query for frequent set mining need information of all three parts. Firstly, like the above process, we will scan all arrays only once, aggregate unit metrics of same items over unit dimensions and build the Root_Query_Table. Subsequently, according to arbitrary group by conditions, we can aggregate results in the Root_Query_Table to build arbitrary set query table named Set_Query_Table. It is important to notice that set dimensions must be a finer granularity than unit dimensions. Through setting filters, we can find frequent sets or heavy hitters. The SQL statement is as follow:

```
SELECT set_dimensions,sum(metric)
FROM (SELECT unit_dimensions,
             sum(unit_metric) as metric
FROM sketch_table
GROUP BY unit_dimensions
) Root_Query_Table
WHERE filters
GROUP BY set_dimensions
```

Examples of frequent set mining (Figure 3): We take the hierarchical aggregation problem as an example. Suppose that

the unit dimension is SrcIP, the set dimension we want to query does not reach Stage 3, the increment $\delta(e)$ is w if $e = e_t$ is 24-bit pre x for SrcIP and the metric is row size. We first and 0 otherwise. If the insertion process reaches Stage 3, we assume $e = e_t^0$ based on the 24-bit pre x for SrcIP to get the Query Table is inserted into Stage 3. (right). There are two rows which has the same 24-bit pre x. If $e = e_t$ there are two cases: Case 1: if $e \in e_t^0$, it means that e is recorded in the first two parts and $f_P(e) + f_C(e)$ increase by w . Consider Stage 3. If e is not recorded in $h_d(e) \in h_d(e_t^0)$, the estimated value does not change. Otherwise, the expected estimated value $f_U(e) = A_d[h_d(e)]:count$ becomes

$$\frac{A_d[h_d(e):count]}{A_d[h_d(e):count + w^0]} (A_d[h_d(e):count + w^0]) = A_d[h_d(e):count]$$

Fig. 3. Example of frequent set mining.

V. ANALYSIS

In this section, we provide a rigorous mathematical analysis for SandwichSketch. First, we prove that the algorithm achieves unbiasedness for frequent set mining in §V-B, then we present an error bound in §V-C.

A. Preliminary

For an arbitrary item e in SandwichSketch, let $f_P(e)$, $f_C(e)$ and $f_U(e)$ denote the inserted value in three parts, respectively. According to the insertion process, we have

$$f(e) = f_P(e) + f_C(e) + f_U(e)$$

Let $f_P(e)$, $f_C(e)$ and $f_U(e)$ denote the estimated value in three parts respectively, since the first two parts are pure buckets and don't have hash collisions, which means when we try to insert an item $(e_a; m)$, and it collides with $(e_b; n)$ at the first part or the second part, we wouldn't delete $(e_a; m)$ nor $(e_b; n)$, but we just put $(e_a; m)$ or $(e_b; n)$ to the next part, which means we wouldn't drop any count of items in the first or second part. Therefore, we can conclude by that, in the query process, we can deduce $f_P(e) = f_P(e)$ and $f_C(e) = f_C(e)$.

B. Unbiasedness

Theorem 1. For any sets of any set dimension, in SandwichSketch,

$$E \hat{f}(s) = f(s)$$

Proof. We first prove that, for items of unit dimension, in SandwichSketch $E \hat{f}(e) = f(e)$.

Let $\hat{f}(e)$ be the estimated size of before t^{th} insertion. Suppose that the incoming item is denoted $(e; w)$ for the t^{th} insertion. Here we prove the unbiasedness by showing that the expected increment $\delta(e)$ is w if $e = e_t$ and 0 otherwise.

Before the insertion, we have $\hat{f}(e) = f_P(e) + f_C(e) + f_U(e)$. According to the insertion operation, each time an item arrives, SandwichSketch will process arrays in sequence, and an item could be evicted from an array and inserted into the next array.

Since Stage 1 and Stage 2 are consisted of pure buckets and those arrays record accurate values. If the insertion process

Therefore, the expected increment $\delta(e)$ is 0 and the expected increment $\delta(e)$ is w .

Case 2: $e = e_t^0$, then based on the insertion process, we can deduce that e is not recorded in the first two parts and $w^0 = w$. Consider Stage 3. If e is recorded, the estimated value increase by w . Otherwise, the expected increment to $f_U(e)$ can be calculated as:

$$\frac{w}{A_d[h_d(e):count + w]} (A_d[h_d(e):count + w]) = w$$

Therefore, the expected increment $\delta(e)$ is w .

If $e \in e_t$ there are two cases:

Case 1: $e \in e_t^0$, then the change to $f_P(e) + f_C(e)$ is 0. Similar to above analysis, the expected increment $\delta(e)$ is 0, so the theorem holds.

Case 2: $e = e_t^0$, then according to the insertion operation, the recorded value of e is evicted into Stage 3, therefore $f_P(e) + f_C(e)$ decrease by w^0 . Similar to above analysis, the expected increment to $f_U(e)$ is w^0 , so the expected increment $\delta(e)$ is $w^0 + w^0 = 0$.

Based on the above results, we can deduce that SandwichSketch achieves unbiasedness for any item of unit dimension. Therefore, for any set of any set dimension, we have

$$E \hat{f}(s) = E \sum_{e \in s} \hat{f}(e) = \sum_{e \in s} f(e) = f(s)$$

□

C. Error Bound

We first introduce Chebyshev's Inequality. Chebyshev's inequality is a fundamental result in probability theory, which provides an upper bound on the probability that a random variable deviates from its mean by a specified amount. The only prerequisite for applying Chebyshev's inequality is that the random variable must have a finite mean and finite, non-zero variance². Formally, for any random variable X with mean $E[X] = \mu$ and variance $\text{Var}(X) = \sigma^2$, and for any $\epsilon > 0$, the inequality states that

$$\Pr(|X - \mu| \geq \epsilon) \leq \frac{\sigma^2}{\epsilon^2}$$

As our estimator $\hat{f}(e)$ satisfies these prerequisites. This result does not make any assumptions about the distribution

of X (such as normality), making it broadly applicable. In essence, Chebyshev's inequality quantifies how the variance of a random variable controls the likelihood of large deviations from the mean.

Theorem 2. For any item of unit dimension, in SandwichSketch,

$$P[|f_U(e) - f(e)| > \frac{1}{2}] \leq \frac{6}{\text{count}}$$

where $\text{count} = \frac{1}{2}$

Proof. In Stage 3, the probability of an item occupying the bucket is proportional to its size, suppose $f_U(e) = j$, we have:

$$P[A_{d[j]}:ID = e] = \frac{f_U(e)}{A_{d[j]}:\text{count}}$$

Based on above probability, the expectation of the estimated size in Stage 3 is:

$$E[f_U(e)] = \frac{f_U(e)}{A_{d[j]}:\text{count}} \cdot A_{d[j]}:\text{count} = f_U(e)$$

The variance can be calculated by:

$$\text{Var}[f_U(e)] = E[(f_U(e) - E[f_U(e)])^2]$$

In our context, there are two cases:

Case A: The estimated value is $A_{d[j]}:\text{count}$ with probability $p_1 = \frac{f_U(e)}{A_{d[j]}:\text{count}}$. Thus, $x_1 = A_{d[j]}:\text{count}$.

Case B: The estimate is 0 with probability $p_2 = 1 - p_1$. Thus, $x_2 = 0$.

The mean is $f_U(e)$. Therefore, substitute into the variance formula: Let $f_U(e) = \sum_{e_i \in e} f_U(e_i)$ in Stage 3. Then, we can get the variance for Stage 3 is:

$$\begin{aligned} \text{Var}[f_U(e)] &= E\left[\frac{f_U(e)}{A_{d[j]}:\text{count}} (A_{d[j]}:\text{count} - f_U(e))^2\right] \\ &\quad + (1 - \frac{f_U(e)}{A_{d[j]}:\text{count}}) (0 - f_U(e))^2 \\ &= f_U(e) E[A_{d[j]}:\text{count} - f_U(e)] \\ &= \frac{f_U(e) f_U(e)}{1} \end{aligned}$$

According to the insertion process, the first two parts don't introduce error, we have $f_U(e) - f(e) = |f_U(e) - f_U(e)|$, therefore, based on the Chebyshev's inequality, we have:

$$\begin{aligned} P[|f_U(e) - f(e)| > \frac{1}{2}] &\leq \frac{6}{\text{count}} \\ &= P[|f_U(e) - f_U(e)| > \frac{1}{2}] \\ &= \frac{\text{Var}[f_U(e)]}{2f_U(e) f_U(e)} = \frac{1}{2} \end{aligned}$$

VI. EVALUATION

A. Experimental Setup

Datasets: We use three real-world datasets in our experiments.

1) CAIDA Dataset: It is streams of anonymized IP packets collected in the Equinix-Chicago monitor by CAIDA in 2018 [44]. We use the trace with a monitoring interval of memory is larger than

60s. Each item is a 5-tuple (13 bytes). There are around 27M items and 1.3M distinct items in this dataset.

2) Network Dataset: It contains users' posting history on the stack exchange website [45]. Each item (4 bytes) represents the ID of each user. There are around 10M items and 0.7M distinct items in this dataset.

3) Web Page Dataset: It is collected from HTML documents [46]. Each item (8 bytes) represents the number of distinct terms in a web page. There are around 32M items and 0.9M distinct items.

Tasks: We evaluate our algorithm on the following tasks.

- 1) Heavy Hitter Detection: It reports objects whose frequencies are larger than a predefined threshold.
- 2) Heavy Change Detection: It is aimed to find objects whose frequencies increase/decrease beyond a predefined threshold in the adjacent time windows.
- 3) Hierarchical Heavy Hitter Detection (HHH): It is a variant of the hierarchical aggregation problem. In this section, we define it as finding all IP prefixes whose frequencies are larger than a given threshold.

Metrics: We evaluate our algorithm using the following metrics.

- 1) Recall Rate: The ratio of true positive items to all actual items.
- 2) Precision Rate: The ratio of true positive items to all reported items.
- 3) F1 Score: $\frac{2 \cdot RR \cdot PR}{RR + PR}$.
- 4) Average Relative Error (ARE): $\frac{1}{j} \sum_{e \in Q} \frac{|f(e) - \hat{f}(e)|}{f(e)}$, where $f(e)$ is the real size, $\hat{f}(e)$ is the estimated size, and Q is the query set.
- 5) Throughput: Million operations (insertions) per second (Mops).

CPU implementation: We implement SandwichSketch and baseline algorithms in C++. The hash functions are implemented using 32-bit Bob Hash [47]. We implement and evaluate them on a 18-core CPU Server (Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz) with 128 GB memory and 24.75 MB L3 Cache.

B. Experiments on Parameter Settings

In this section, we compare the performance of the algorithm and user-set parameter settings on the CAIDA dataset.

1) Algorithm Parameters: The following experiments are conducted on the heavy hitter detection, and we set the threshold to 10^{-4} of the total size of the dataset.

Firstly, we mainly introduce the effect of d_1 and d_2 . We set the ratio $r = 0.75$ as the default value and compare 6 parameter combinations. The figure legend $d_1 = 2$ and $d_2 = 2$.

Varying d_1 and d_2 for frequent item mining (Figures 4(a)-4(b)): For $3 : 0$ and $0 : 3$, we can clearly find that their not only recall rate but also ARE are less than other combinations, which means that using only Stage 1 or Stage 2 will not work accurately. For $2 : 2$, $3 : 3$ and $4 : 4$, they have real close performance and all achieve high accuracy. Specially, when $d_1 = 2$ and $d_2 = 2$, their recall rate is larger than

(a) Recall Rate vs. Memory (b) ARE vs. Memory (c) PR vs. Num of Dimensions (d) RR vs. Num of Dimensions (e) ARE vs. Num of Dimensions

Fig. 4. Varying d_1 and d_2 , where $A : B$ in the legend means $d_1 = A$ and $d_2 = B$.

(a) Recall Rate vs. Memory (b) ARE vs. Memory (c) PR vs. Num of Dimensions (d) RR vs. Num of Dimensions (e) ARE vs. Num of Dimensions

Fig. 5. Varying the ratio r .

(a) Heavy Hitter (I) (b) Heavy Change (I) (c) Heavy Hitter (S) (d) Heavy Change (S) (e) 1-d HHH (f) 2-d HHH

Fig. 6. Varying the threshold, where I and S represent frequent item mining and frequent set mining, respectively.

98.4% and ARE is less than 0.7%. In order to achieve high throughput, we usually choose $r = 2$ as the optimal parameter. Varying d_1 and d_2 for frequent set mining (Figures 4(c)-4(e)): What's obvious is that $8 : 0$ and $0 : 3$ are two with the worst performance. For $2 : 2$, $3 : 3$ and $4 : 4$, the differences in the precision rate, recall rate, and ARE are less than 0.09 and 0.2%, respectively. When querying 6 set dimensions for these three parameter combinations, the precision rate and recall rate reach at least 92.0% and 96.5%, respectively, and the ARE is less than 4.7%. As a result, we choose $r = 2$ as the optimal parameter considering the insertion speed.

In addition, we set $d_1 = 2$, $d_2 = 2$ and measure the effect of the ratio r . We can dynamically adjust to frequent item mining or frequent set mining. Varying the ratio r for frequent item mining (Figures 5(a)-5(b)): As the ratio increases, the recall rate increases at the same time. The reason is that frequent item mining totally depends on Stage 1 and Stage 2. The larger the memory, the higher the accuracy. Under 200KB and the ratio = 0.9, the recall rate and ARE are 95.4% and 1.2%, respectively. Therefore, we choose the ratio = 0.9 as the optimal value for frequent item mining. Without considering frequent set mining at all, we can set $r = 1$ to maximize accuracy.

Varying the threshold for frequent set mining (Figures 6(a)-6(f)): We can observe that it has high accuracy for both frequent item mining and frequent set mining query when the ratio = 0.7. Specially, for frequent item mining, the recall rate of = 0.7 is just 0.8% lower than that of the optimal parameter on average, and the ARE of the optimal parameter is just 0.2% better than that of $r = 0.7$. For frequent set mining, when querying 6 set dimensions, the precision rate and recall rate are only about 0.9% and 0.2% lower than those of the optimal parameter, respectively, and the ARE of the optimal parameter is just 1.05% better than that of $r = 0.7$.

The User-Set Parameter We conduct the following experiments on all tasks to evaluate the effect of threshold varying and set the algorithm parameters based on the previous winners.

Varying the threshold for all tasks (Figures 6(a)-6(f)): We find that when the memory or the number of dimensions is fixed, the change in throughput induced by varying the

threshold is small, with only the one on the Heavy Change (5) dimensions (5-tuple, SrcIP, DstIP, (SrcIP, SrcPort) pair, (DstIP, DstPort) pair and (SrcIP, DstIP)), set the threshold to 10^4 of the total size of dataset and report average metrics on these dimensions. For HHH detection, we set the threshold to be 5×10^5 .

C. Experiments on Frequent Item Mining

In this section, we compare SandwichSketch ("Ours" in the figures) with 8 other sketches including Count-Min sketch (CM) [13] with a d-left hashing table (CM-Hash), Count sketch (C) [14] with a d-left hashing table (C-Hash), Unbiased Space-Saving (USS) [29], CocoSketch [25], Space-Saving (SS), Augmented sketch (AS) [17], Salsa [19], LogLog Iter (LLF) [20]. We perform experiments on CAIDA, Network and Web Page datasets and use 10M items. For heavy hitter and heavy change detection, we set the threshold to 2×10^5 of the total size of the dataset.

Implementation: We set $d_1 = 2$, $d_2 = 2$ and $r = 0.9$ in SandwichSketch. For an item inserted into CM-Hash/C-Hash, it will first be inserted to CM/C and then return a query value for this item. If the query value exceeds a threshold, it will be inserted into the d-left hashing table. For CM-Hash/C-Hash, we set the memory ratio of CM and C to 0.95 and 0.95, respectively, and the threshold to 256. USS and SS uses an optimized implementation enhanced by a hash table and a double-linked list (Naive implementation of USS has a very high update delay). The hash table is used to check whether an item is already in the USS. The double-linked list is used to rank the buckets by their counters so that we can find the minimal bucket as quickly as possible. For AS, the sketch consists of 32 buckets, and the rest of the memory is for CM. Each bucket has ID, newcount, oldcount and oldcount. For Salsa, we use the CM version, set $s = 4$ and pick $s = 8$ bit counters. For LLF, we allocate 75% memory for CU sketch [15]. We give 4 bits for each register, set the number of hash functions to 3 and threshold to 5.

Heavy hitter detection with different memory (Figure 7): Under 400KB memory, the F1 Score of SandwichSketch is 92.8% and 96.9% on two datasets, respectively, and other baseline algorithms are still on a low level. On average, under 200KB memory, SandwichSketch achieves 53.1% and 45.4% improvements on the F1 Score and is 3.7 and 38.4 better on the ARE on two datasets, respectively.

Heavy change detection with different memory (Figure 8): Under 400KB memory, the F1 Score of SandwichSketch is above 95.6% and 91% on two datasets, respectively, and other baseline algorithms are still on the low level. On average, under 200KB memory, SandwichSketch achieves 49.5% and 54.2% improvements on the F1 Score and is 3 and 12.5 better on the ARE on two datasets, respectively.

D. Experiments on Frequent Set Mining

In this section, we compare SandwichSketch ("Ours" in the figures) with 3 other baseline algorithms including theoretical optimal accuracy of frequent set mining (Ideal), Unbiased SpaceSaving (USS) [29] and CocoSketch [25]. We take the network measurement scenario to perform all kinds of experiments on CAIDA dataset. For heavy hitter and heavy change detection, we simultaneously querying 6 different set

Implementation: We set $d_1 = 2$, $d_2 = 2$, $r = 0.5$ and the total memory is 600KB in SandwichSketch. The implementation of USS is the same as the version for frequent item mining. Unfortunately, USS depends on a double-linked list, which occupies lots of memory on pointers. In order to reach theoretical optimal accuracy of frequent set mining, we use min-heap with (ID; count) buckets and only consider the memory usage of all buckets.

Heavy hitter detection with different numbers of dimensions (Figures 9(a)-9(b)): The F1 Score of SandwichSketch is above 95.6% regardless of the number of tracked set dimensions while this value of CocoSketch and USS is less than 83.4% and 42.6%. On average, the F1 Score of SandwichSketch is 3.9% and 57.3% higher than those of CocoSketch and USS, respectively, while the ARE is 3.6 better. Compared with the theoretical optimum, the difference in the F1 Score is just 1.7% and the ARE of SandwichSketch is just 1.9 worse.

Heavy hitter detection with different memory (Figures 9(c)-9(d)): The F1 Score of SandwichSketch is above 90.4% except with 200KB memory while this value of CocoSketch and USS is always lower than 80% and 88.5%. For the ARE, SandwichSketch is 3 and 3.8 better than those of CocoSketch and USS, respectively. Besides, the difference in accuracy between SandwichSketch and the theoretical optimum is quite small.

Heavy change detection with different numbers of dimensions (Figures 10(a)-10(b)): When querying 6 set dimensions at the same time, the F1 Score of SandwichSketch achieves 17.5% and 29% improvements while the ARE is 5 and 4.5 better than those of CocoSketch and USS, respectively. Under the different number of set dimensions, the F1 Score of SandwichSketch is above 95.7%. As for the difference between SandwichSketch and the theoretical optimum, the difference in the F1 Score is just 1.7% and the ARE of the theoretical optimum is just 1.2 better than that of SandwichSketch.

Heavy change detection with different memory (Figures 10(c)-10(d)): Under 200KB memory, the F1 Score of SandwichSketch achieves 36.7% and 50.7% improvements while the ARE of SandwichSketch is 2.8 and 5.1 better than those of CocoSketch and USS, respectively. The F1 Score of SandwichSketch is larger than 91.9% above 400KB memory but this value of CocoSketch is always less than 85.6%. The ARE of the theoretical optimum is just 5 better than that of SandwichSketch, which is just a tiny gap.

HHH detection with different memory (Figure 11): We consider the source IP hierarchy of IPv4 in bit granularity (32 (32 indexes + 1 empty IP = 33 set dimensions) in HHH detection (HHH detection). Under 200KB memory and compared with CocoSketch and USS, the F1 Score of SandwichSketch achieves 9% and 29.7% improvements while the ARE is 3.8 and 15 better, respectively. When memory is larger than

(a) CAIDA: F1

(b) CAIDA: ARE

(c) Web Page: F1

(d) Web Page: ARE

Fig. 7. Heavy hitter detection of frequent item mining with different memory.

(a) CAIDA: F1

(b) CAIDA: ARE

(c) Network: F1

(d) Network: ARE

Fig. 8. Heavy change detection of frequent item mining with different memory.

(a) F1 vs. Num of Dimensions (b) ARE vs. Num of Dimensions

(c) F1 vs. Memory

(d) ARE vs. Memory

Fig. 9. Heavy hitter detection of frequent set mining.

(a) F1 vs. Num of Dimensions (b) ARE vs. Num of Dimensions

(c) F1 vs. Memory

(d) ARE vs. Memory

Fig. 10. Heavy change detection of frequent set mining.

(a) F1 (b) ARE

Fig. 11. 1-d HHH detection with different memory.

(a) F1 (b) ARE

Fig. 12. 2-d HHH detection with different memory.

or equal to 400KB, the difference in the F1 Score between SandwichSketch and the theoretical optimum is less than 0.9%.

2-d HHH detection with different memory (Figure 12): We consider the source IP and destination IP hierarchies of IPv4 in bit granularity (32 bits = 1089 set dimensions) in HHH detection (2-d HHH detection). Under 0.5MB memory, the F1 Score of SandwichSketch achieves 16.5% and 40% improvements while the ARE is 3.9 and 7.4 better than those of CocoSketch and USS, respectively. Above 1MB, the difference in the F1 Score between SandwichSketch and the theoretical optimum is less than 0.7%, which means that the performance of SandwichSketch is very close to the theoretical optimal value.

E. Experiments on Software Platform

In this section, we compare the throughput of SandwichSketch with CocoSketch on Redis platform.

Redis implementation: We implement the SandwichSketch and CocoSketch as a module in the Redis, where users can use the provided API to create SandwichSketch/CocoSketch, insert items and execute queries. MurmurHash [48] is used as the hash function.

Throughput on Redis platform (Figure 13): The throughput of the SandwichSketch/CocoSketch in Redis will not be affected with the increase of memory configuration. The experimental results show that SandwichSketch and CocoSketch in Redis achieve nearly 0.4 Mops throughput.

Fig. 13. Throughput on the Redis platform.

(a) Throughput (b) Resource Usage

Fig. 14. Throughput and resource usage on the FPGA platform.

F. Experiments on Hardware Platform

In this section, we compare the performance (throughput and resource usage) of SandwichSketch with the hardware version of CocoSketch on an FPGA platform.

FPGA implementation: We implement SandwichSketch on an experimental FPGA platform (Virtex-7 VC709 XC7VX690T) with 433200 Slice LUTs, 866400 Slice Registers, and 1470 Block RAM Tiles. The FPGA-based SandwichSketch is fully pipelined, which can input an item in every clock, and each item needs 30 clocks if it is inserted into the bucket of Stage 3 module.

Throughput on the FPGA platform (Figure 14(a)): The difference in throughput between SandwichSketch and CocoSketch is not significant. Under four different memory configurations, the throughput of these two algorithms is larger than 200 Mops. Specifically, with 0.25MB memory, SandwichSketch is expected to achieve 230 Mops while CocoSketch reaches 236 Mops.

Resource usage on the FPGA platform (Figure 14(b)): The main resources of FPGA consist of Slice LUTs, Slice Registers and Block RAM Tiles. Slice LUTs are lookup tables, which are mainly used for combinational logic. Slice Registers are cache resources and Block RAM Tiles are main storage resources. Slice LUTs usage of SandwichSketch is 0.5% while CocoSketch needs 0.08%. Meanwhile, SandwichSketch and CocoSketch need 0.12% and 0.04% Slice Registers, respectively. The main resource usage for these two algorithms is Block RAM Tiles, which achieves 5% and 1% for them, respectively.

G. Experiments on Distributed Measurement

In a distributed scenario, there are data streams S_1 ; S_N ($S = [S_1, S_2, \dots, S_N]$) and each data stream S_i contains m_i items. At the server node, we deploy a sketch with the same

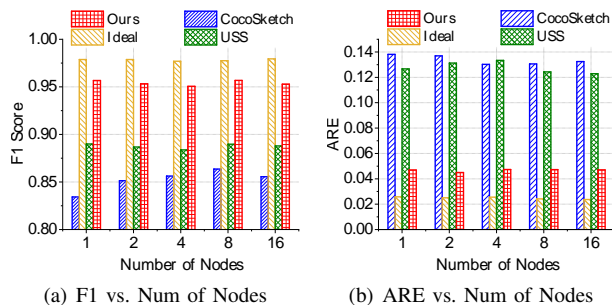


Fig. 15. Evaluations in the distributed scenario.

memory configuration to measure the data stream S_j . For each item of the CAIDA dataset, we map it to a node by its hashed value. In this experiment, we set the default memory to 600KB and focus on the frequent set mining while querying 6 set dimensions. Specifically, we will first merge results of sketches from all nodes to acquire global results and then answer the query over global results.

Heavy hitter detection with different number of nodes (Figures 15(a)-15(b)): For SandwichSketch, the F1 Score is at least 95% and the ARE is at most 4.4%. When the number of nodes increases from 1 to 16, the decrease in the F1 Score and ARE of SandwichSketch is less than 0.37% and 0.01%, respectively, which means that the increase in the number of nodes has little impact on frequent set mining. In addition, it also proves that SandwichSketch is suitable for sketch merging task. With 16 nodes, SandwichSketch achieves 10% improvement over CocoSketch on the F1 Score and is 2.81 better on the ARE.

VII. CONCLUSIONS

This paper takes an important step in supporting frequent object mining, including frequent item mining and frequent set mining, which has wide applications in data mining, network and security fields. We draw on the philosophy of sandwich making and propose a sketch-based measurement framework named SandwichSketch. Thanks to the proposed double fidelity enhancement and hierarchical hot locking techniques, it can handle two key tasks with great fidelity. We implement SandwichSketch on three popular platforms (CPU, Redis and FPGA) and demonstrate that it outperforms the SOTA solutions under three real-world datasets and supports a distributed measurement scenario.

ACKNOWLEDGMENT

The authors sincerely thank Dayu Wang from Peking University for improving the mathematical analysis of our paper. This work was supported in part by the National Key R&D Program of China (No. 2024YFB2906603), in part by the Project of the Department of Strategic and Advanced Interdisciplinary Research of Pengcheng Laboratory (No. 2025QYB004), and in part by the National Natural Science Foundation of China (NSFC) (No. 62402012, 62372009, 624B2005).

REFERENCES

- [1] L. Wang, R. Cheng, S. D. Lee, and D. Cheung, "Accelerating probabilistic frequent itemset mining: a model-based approach," in *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM)*, 2010, pp. 429–438.
- [2] L. Wang, D. W.-L. Cheung, R. Cheng, S. D. Lee, and X. S. Yang, "Efficient mining of frequent item sets on large uncertain databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, no. 12, pp. 2170–2183, 2011.
- [3] Z. Halim, O. Ali, and M. G. Khan, "On the efficient representation of datasets as graphs to mine maximal frequent itemsets," *IEEE Transactions on Knowledge and Data Engineering*, vol. 33, no. 4, pp. 1674–1691, 2019.
- [4] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 101–114.
- [5] Q. Xiao, S. Chen, Y. Zhou, M. Chen, J. Luo, T. Li, and Y. Ling, "Cardinality estimation for elephant flows: A compact solution based on virtual register sharing," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3738–3752, 2017.
- [6] L. Tang, Q. Huang, and P. P. Lee, "Mv-sketch: A fast and compact invertible sketch for heavy flow detection in network data streams," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, 2019, pp. 2026–2034.
- [7] D. Nguyen, T. D. Nguyen, W. Luo, and S. Venkatesh, "Trans2vec: learning transaction embedding via items and frequent itemsets," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, 2018, pp. 361–372.
- [8] C.-Y. Hsu, P. Indyk, D. Katabi, and A. Vakilian, "Learning-based frequency estimation algorithms," in *International Conference on Learning Representations (ICLR)*, 2019.
- [9] H. Hu, X. He, J. Gao, and Z.-L. Zhang, "Modeling personalized item frequency information for next-basket recommendation," in *Proceedings of the 43rd international ACM SIGIR Conference on Research and Development in Information Retrieval*, 2020, pp. 1071–1080.
- [10] S. Qiu, B. Wang, M. Li, J. Liu, and Y. Shi, "Toward practical privacy-preserving frequent itemset mining on encrypted cloud data," *IEEE Transactions on Cloud Computing*, vol. 8, no. 1, pp. 312–323, 2017.
- [11] L. Tang, Q. Huang, and P. P. Lee, "Spreadsketch: Toward invertible and network-wide detection of superspreaders," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, 2020, pp. 1608–1617.
- [12] P. Jia, P. Wang, Y. Zhang, X. Zhang, J. Tao, J. Ding, X. Guan, and D. Towsley, "Accurately estimating user cardinalities and detecting super spreaders over time," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 92–106, 2020.
- [13] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
- [14] M. Charikar, K. C. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoretical Computer Science*, vol. 312, no. 1, pp. 3–15, 2004.
- [15] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems*, vol. 21, no. 3, pp. 270–313, 2003.
- [16] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *International Conference on Database Theory (ICDT)*, 2005, pp. 398–412.
- [17] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, 2016, pp. 1449–1463.
- [18] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, "Cold filter: A meta-framework for faster and more accurate stream processing," in *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, 2018, pp. 741–756.
- [19] R. B. Basat, G. Einziger, M. Mitzenmacher, and S. Vargaftik, "Salsa: self-adjusting lean streaming analytics," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 864–875.
- [20] P. Jia, P. Wang, J. Zhao, Y. Yuan, J. Tao, and X. Guan, "Loglog filter: Filtering cold items within a large range over high speed data streams," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 804–815.
- [21] N. Duffield, C. Lund, and M. Thorup, "Priority sampling for estimation of arbitrary subset sums," *Journal of the ACM*, vol. 54, no. 6, pp. 32–es, 2007.

- [22] A. Shrivastava, A. C. Konig, and M. Bilenko, "Time adaptive sketches (ada-sketches) for summarizing data streams," in *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, 2016, pp. 1417–1432.
- [23] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017, pp. 127–140.
- [24] Y. Zhang, S. Singh, S. Sen, N. Duffield, and C. Lund, "Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications," in *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2004, pp. 101–114.
- [25] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang, "Cocosketch: high-performance sketch-based measurement over arbitrary partial key query," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 207–222.
- [26] D. Vengerov, A. C. Menck, M. Zait, and S. P. Chakkappen, "Join size estimation subject to filter conditions," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1530–1541, 2015.
- [27] J. Carlson, *Redis in action*. Simon and Schuster, 2013.
- [28] "Alveo U280 Data Center Accelerator Card." 2022. [Online]. Available: <https://www.xilinx.com/products/boards-and-kits/alveo/u280.html>
- [29] D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," in *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, 2018, pp. 1129–1140.
- [30] "Source code related to SandwichSketch," 2023. [Online]. Available: <https://github.com/pkufzc/SandwichSketch>
- [31] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE Security & Privacy*, vol. 1, no. 4, pp. 33–39, 2003.
- [32] V. Sekar, N. G. Duffield, O. Spatscheck, J. E. van der Merwe, and H. Zhang, "Lads: Large-scale automated ddos detection system," in *USENIX Annual Technical Conference, General Track*, 2006, pp. 171–184.
- [33] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred, "'007: Democratically finding the cause of packet drops," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018, pp. 419–435.
- [34] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *Proceedings of the ACM SIGCOMM 2011 conference*, 2011, pp. 350–361.
- [35] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014, pp. 71–85.
- [36] P. Richter, R. Padmanabhan, N. Spring, A. Berger, and D. Clark, "Advancing the art of internet edge outage detection," in *Proceedings of the Internet Measurement Conference (IMC)*, 2018, pp. 350–363.
- [37] T. Holterbach, E. Aben, C. Pelsser, R. Bush, and L. Vanbever, "Measurement vantage point selection using a similarity metric," in *Proceedings of the 2017 Applied Networking Research Workshop*, 2017, pp. 1–3.
- [38] Z. Fan, R. Wang, Y. Cai, R. Zhang, T. Yang, Y. Wu, B. Cui, and S. Uhlig, "Onesketch: A generic and accurate sketch for data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 12 887–12 901, 2023.
- [39] Y. Zhao, W. Han, Z. Zhong, Y. Zhang, T. Yang, and B. Cui, "Double-anonymous sketch: Achieving top-k-fairness for finding global top-k frequent items," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 1, no. 1, pp. 1–26, 2023.
- [40] Y. Li, F. Wang, X. Yu, Y. Yang, K. Yang, T. Yang, Z. Ma, B. Cui, and S. Uhlig, "Ladderfilter: Filtering infrequent items with small memory and time overhead," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 1, no. 1, pp. 1–21, 2023.
- [41] L. Zheng, Q. Xiao, and X. Cai, "A universal sketch for estimating heavy hitters and per-element frequency moments in data streams with bounded deletions," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 2, no. 6, pp. 1–28, 2024.
- [42] Q. Shi, X. Li, H. Zheng, T. Yang, Y. Wang, and M. Xu, "Heavylocker: Lock heavy hitters in distributed data streams," in *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*, 2025, pp. 1244–1255.
- [43] R. Miao, Y. Zhang, G. Qu, K. Yang, T. Yang, and B. Cui, "Hyper-uss: Answering subset query over multi-attribute data stream," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 1698–1709.
- [44] "The CAIDA anonymized 2016 internet traces." 2016. [Online]. Available: <http://www.caida.org/data/overview/>
- [45] "The Network dataset Internet Traces," 2014. [Online]. Available: <http://snap.stanford.edu/data/>
- [46] "Real-life Transactional Dataset." 2004. [Online]. Available: <http://fimi.uantwerpen.be/data/>
- [47] "The source code of Bob Hash," 1997. [Online]. Available: <http://burtleburtle.net/bob/hash/evahash.html>
- [48] "The source code of MurmurHash," 2015. [Online]. Available: <https://github.com/aappleby/smhasher>



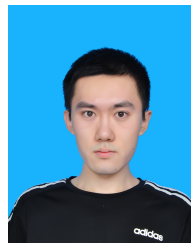
Zhuochen Fan (Member, IEEE) received his Ph.D. degree in computer science from Peking University in 2023, advised by Professor Tong Yang. He is currently an Associate Researcher with Pengcheng Laboratory. Before joining Pengcheng Laboratory, he was a Boya Post-Doctoral Research Fellow at Peking University. His research interests include approximation algorithms in computer networks and databases. He had articles published by IEEE/ACM TRANSACTIONS ON NETWORKING, IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, THE VLDB JOURNAL, ICDE, RTSS, ICPP, ICNP, USENIX ATC, etc:



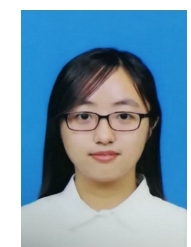
Ruixin Wang received the M.E. degree in Software Engineering from Peking University in 2024. His research interests include network measurements, sketches and machine learning.



Zihan Jiang received the B.E. degree in Information and Computing Science from Peking University in 2024. He is currently working toward the M.E. degree in Electronic Information with Peking University, advised by professor Tong Yang. His research interests include data structures and algorithms in networking.



Ruwen Zhang received the M.E. degree in Software Engineering from Peking University in 2024, advised by professor Tong Yang. He is interested in networks and data stream processing.



Sha Wang received the M.E. degree in Cyberspace Security with the College of Computer Science and Technology, National University of Defense Technology. Her main research interests include Time Sensitive Network.

