Answering Subset Query over Multi-Attribute Data Streams using Hyper-USS

Ce Zheng, Zhouran Shi, Ruijie Miao, Wenpu Liu, Tong Yang, *Member, IEEE*, Bin Cui, *Fellow, IEEE*, and Steve Uhlig

***Abstract*—Approximate queries offer an efficient means of analyzing massive data streams under acceptable errors. Among these, subset queries over multiple attributes are common in many real-world applications. While sketches offer promising approximate solutions for massive data streams, efficiently supporting subset queries over multiple statistical attributes remains a significant challenge. To address this, we propose Hyper-USS, a novel sketching solution that accurately and efficiently supports subset queries over data streams involving multiple statistical attributes. With Joint Variance Optimization, Hyper-USS provides unbiased estimation and optimizes estimation variance jointly, addressing the challenge of accurately estimating multiple statistical attributes in the sketch design. The algorithm records the information of keys and all attributes in one sketch, ensuring high insertion efficiency. Furthermore, its three speed-optimized versions are introduced to handle the growing number of statistical attributes in data streams. Experimental results show that Hyper-USS and its three speed-optimized versions consistently surpass state-of-the-art methods that support subset queries in both estimation accuracy and insertion throughput. Specifically, Hyper-USS improves accuracy by at least 38%, while the algorithm and its three speed-optimized versions achieve throughput improvements of up to $31.90\times$, $45.31\times$, $49.21\times$, and $58.03\times$, respectively. The code is open-sourced on GitHub[1].**

***Index Terms*—Sketch, multi-attribute data streams, subset query, unbiased estimation**

I. Introduction

A. Background and Motivation

Approximate queries for massive data streams have wide applications in data analysis [2]–[12], especially when processing efficiency is a high priority and a certain level of error is acceptable. In practice, data streams often involve multiple attributes, and users may simultaneously estimate several attributes of interest to support decision-making and system monitoring [13], [14]. That is, each item in streams can be denoted as $(e, v_1, v_2, \cdots, v_n)$, where e is the key and $v_1, v_2, \cdots, v_n$ are the statistical attributes used for aggregation analysis. Users can filter data by setting value or range constraints on predicate combinations [5]–[7], and then aggregate statistical attributes to obtain query results over the subset of items whose keys satisfy these conditions. These queries are referred to as subset queries over multiple attributes.

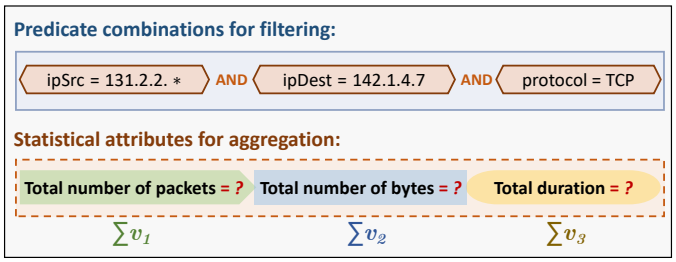


Fig. 1: A typical query use case in network measurement.

Many real-world problems can be abstracted as subset queries over multiple attributes. For example, in network measurement [15], [16] (shown in Fig. 1), users need to query a subset of items where the source IP falls within the `131.2.2.*` subnet, the destination IP is `142.1.4.7`, and the protocol is `TCP`. Under such filtering conditions, users often focus on multiple statistical attributes simultaneously, such as total packets, total bytes, and total duration. These can be used for subset sum as well as ratio and weighted sum queries across attributes. When treating each data item as a table row, let i and j denote the indices of the target attributes, and let w represent the weights. Formulated as:

```
SELECT SUM (v_i), SUM (v_j),
SUM(v_i) / SUM(v_j) AS RATIO,
SUM(v_i) * w_i + SUM(v_j) * w_j AS WEIGHTED_SUM
FROM table
WHERE Key in Subset
```

B. Prior Art and Limitations

To answer approximate queries for massive data streams, many approximation techniques have been developed, and sketches are one of the most popular algorithms among them. Sketches can be classified into two types. Most sketches [17]–[26] are designed for point queries[2], and existing works [3],

This work was supported by the National Key R&D Program of China (No. 2024YFB2906602).

The preliminary version of this paper titled "Hyper-USS: Answering Subset Query Over Multi-Attribute Data Stream" [1] has been published in the proceedings of the 29^{th} ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD).

Ce Zheng is with the School of Cyber Science and Technology, Beihang University, Beijing 100191, China. E-mail: *zhengce*@buaa.edu.cn.

Zhouran Shi is with the School of Mathematical Sciences, Peking University, Beijing 100871, China. E-mail: *shizhouran*@gmail.com.

Ruijie Miao, Wenpu Liu, and Bin Cui are with the State Key Laboratory of Multimedia Information Processing, School of Computer Science, Peking University, Beijing 100871, China. E-mail: {*miaoruijie*, *bin.cui*}@pku.edu.cn, *liuwp*@stu.pku.edu.cn.

Corresponding author Tong Yang is with the State Key Laboratory of Multimedia Information Processing, School of Computer Science, Peking University, Beijing 100871, China. Email: *yangtong*@pku.edu.cn.

Steve Uhlig is with the School of Electronic Engineering and Computer Science, Queen Mary University of London, London E1 4NS, U.K. Email: *steve*@eecs.qmul.ac.uk.

[1]https://github.com/moxiaoshao/Fast_Hyper-USS

[2]The point query refers to querying the results of a single key.

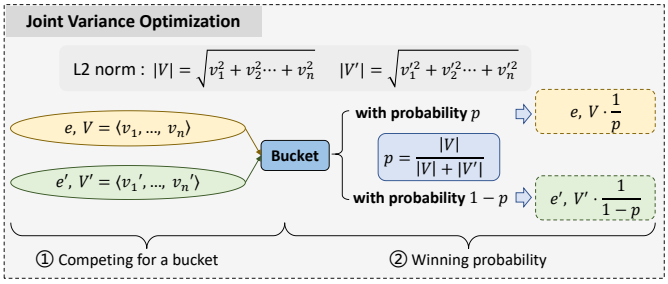


Fig. 2: The Joint Variance Optimization in Hyper-USS.

[4], [15] show that such algorithms are inefficient in supporting subset queries. The other type of sketches [3]–[5], [15] achieves high accuracy and high processing speed in supporting subset queries. However, they are often designed for data stream models with a single statistical attribute. When extended to multiple attributes, a straightforward solution is to build a separate sketch for each attribute to be aggregated. This method wastes memory by repeatedly recording the information of keys in all sketches, and inserting one item requires updating all sketches. Therefore, it suffers from limited performance in both accuracy and insertion throughput.

Furthermore, as the number of statistical attributes to be analyzed in data streams continues to grow, the computational overhead and complexity of existing sketch algorithms increase accordingly, limiting their practical usability. Consequently, designing sketch algorithms that can support multi-dimensional and even high-dimensional statistical attributes while maintaining both efficiency and accuracy remains a critical and challenging research problem.

C. Our Solution

We propose Hyper-USS, a novel sketching solution designed to efficiently answer subset queries over data streams involving multiple statistical attributes. Hyper-USS records the information of keys and all attributes in one sketch. And the method provides for all statistical attributes unbiased estimation and optimized estimation variance, which is well acknowledged as the golden principle for accurate subset query [3], [4]. Therefore, Hyper-USS achieves high accuracy and high insertion throughput for the subset query over multiple statistical attributes.

The challenge in sketch design lies in providing unbiased estimation while optimizing variance for all statistical attributes. To address the problem, we propose our key technique, namely, Joint Variance Optimization. In order to achieve high accuracy for all statistical attributes, we set the optimization goal as minimizing the sum of the estimation variance of all statistical attributes. As a result, optimizing the sum of variances will jointly improve the accuracy of all statistical attributes. Specifically, we generalize the idea of probability proportional to size sampling to scenarios involving multiple statistical attributes (see Fig. 2). When two items compete for a bucket, each item has a probability of winning the competition. The winning probability of two items is proportional to the L2 norm of the attributes. If one item successfully stays in the bucket, all its attributes are divided by its winning probability. This vector-based evaluation mechanism enhances query accuracy while ensuring the unbiasedness of the estimated values. We present a theoretical analysis of Hyper-USS, showing that it provides unbiased estimation, minimizes the total variance across all statistical attributes, and offers a formal error bound.

To address the growing number of attributes in data streams, we design three speed-optimized versions of Hyper-USS to further improve insertion efficiency. Each version targets a specific performance bottleneck: (1) the high computational overhead of L2 norm calculation, (2) the inefficiency of item positioning due to multiple hash computations, and (3) the insufficient use of the inherent parallelism of the algorithm. First, the Precomputed L2-Norm Optimization introduces a 1-bit memory overhead to distinguish between frequent and infrequent items. The L2 norm is precomputed only for infrequent items, which reduces the number of computations. Second, the Hash Simplification Structure Optimization uses a single hash function for item positioning, eliminating the speed bottleneck caused by multiple hash computations. Finally, the SIMD Parallel Acceleration Optimization fully utilizes the support for contiguous memory access provided by the algorithm and employs SIMD techniques [27] to achieve parallel acceleration. Experimental results show that Hyper-USS and its speed-optimized versions consistently outperform SOTA methods in both accuracy and throughput.

In addition, our approach extends support for subset queries by enabling arbitrary combinations of predicates as filtering conditions, where individual conditions can be specified using exact values, wildcards, or ranges. All related codes of Hyper-USS are provided open-source and available at GitHub [28].

Our main contribution can be summarized as follows.

❶ We propose Hyper-USS, which supports subset queries over data streams with multiple statistical attributes efficiently (§ III).

❷ We provide formal theoretical proofs, including unbiasedness, variance minimization, and the error bound (§ IV).

❸ To further improve insertion efficiency while maintaining accuracy, we propose three speed-optimized versions (§ V).

❹ Extensive experiments validate the efficiency and accuracy of Hyper-USS. Compared to SOTA sketches for subset queries involving multiple statistical attributes, it achieves a $31.90\times$ speedup in insertion, reduces estimation error by at least 38%, and, with three speed-optimized versions, boosts throughput by up to $45.31\times$, $49.21\times$, and $58.03\times$, respectively (§ VI).

II. Related Work

A. Sketching Algorithms

Although the traditional hashing technique could provide exact statistics, its memory usage grows rapidly with data scale. In contrast, sketching algorithms [17]–[26], [29]–[35] are compact data structures designed to answer approximate queries using limited memory. They offer high throughput and provable error bounds, making them well-suited for large-scale data stream processing. Most existing sketches typically support only point queries and focus on a single statistical attribute. Although recent works have made progress in subset queries, they still face limitations when dealing with subset queries involving multiple statistical attributes.

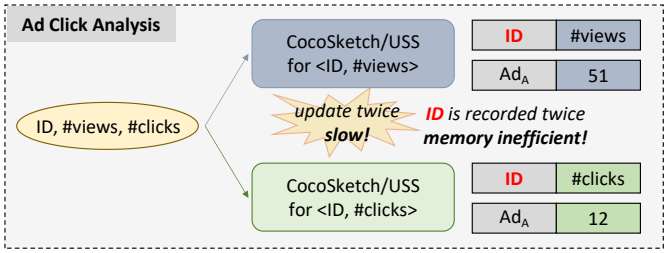


Fig. 3: The limitations of SOTA sketches (CocoSketch/USS) on subset query over multiple statistical attributes.

Sketches for Point Query. In data stream processing, a point query refers to retrieving the result associated with a single key. The research community has defined a lot of fundamental tasks and designed corresponding sketches in data stream processing, such as heavy hitter detection [19], [29]–[31], pattern mining [32], [33], and more [23], [34], [35]. Most tasks are defined over the data stream model involving a single statistical attribute, and the corresponding sketches should only support point queries. For example, in heavy hitter detection of single keys, users are only interested in those single keys with a large aggregated sum of value. As pointed out by prior work [3], [4], sketches for point query suffer from inaccuracy and low throughput when applied to subset query, since these sketches support subset query by building one sketch for each queried subset. As a result, the memory of one specific sketch is reduced, causing a drop in accuracy, while the insertion of one item requires updating multiple sketches and limits insertion throughput.

Sketches for Subset Query. Recently, researchers have noticed an increased demand for subset queries. Unbiased SpaceSaving (USS) [3] and CocoSketch [4] are the two representative sketch solutions. The key design of both is similar. An incoming item will choose a bucket and compete with the recorded item in the bucket to decide which item stays in the bucket. The settings of winning probability in both algorithms follow the idea of probability proportional to size sampling [36]. Suppose the incoming item (A, V_A) competes with the recorded item (B, V_B). With probability $\frac{V_A}{V_A+V_B}$, the incoming item wins and the bucket is updated to $(A, V_A + V_B)$. With probability $\frac{V_B}{V_A+V_B}$, the recorded item wins and the bucket is updated to (B, V_A+V_B). The probabilistic substitution ensures that one bucket provides unbiased estimation and minimized variance for two items. To locate the bucket for the incoming item, USS selects the bucket with the minimum value among all buckets, while CocoSketch selects the bucket with the minimum value among the hashed buckets. CocoSketch improves USS throughput while maintaining high accuracy, and is regarded as the SOTA sketch solution for subset query.

However, directly applying CocoSketch for the subset query over multiple statistical attributes may suffer from inefficiency in terms of both memory and throughput. To support multiple attributes, we have to build one CocoSketch for each attribute. From the aspect of accuracy, as CocoSketch records keys in the bucket, multiple CocoSketches will keep multiple copies of keys, which drags down memory efficiency. From the aspect of throughput, each insertion should update multiple CocoSketches, leading to low throughput. Fig. 3 shows an example of ad click analysis with two attributes: the number of views and the number of clicks. Building one sketch for each attribute requires updating two sketches for each incoming item, causing a halving of throughput. Moreover, both sketches record the ID of ads, and the duplicated ID records lead to inefficient memory usage.

B. Multi-Attribute Model

The multi-attribute model naturally arises in many real-world scenarios and has thus been extensively studied [37]–[40]. Related research spans various domains, including multi-attribute databases [41], [42], business application systems [43], and other areas [44], [45].

A data stream can be modeled as multi-attribute, meaning that each incoming item contains several different attributes. In the study of multi-attribute data stream models, Hydra [7] by Manousis *et al.* and OmniSketch [5], [6] by Punter *et al.* are two of the most representative works. Hydra addresses multi-dimensional data stream processing under different predicate combinations and emphasizes some statistical metrics, such as α-heavy hitters, entropy, and cardinality. And all these metrics target a statistical attribute, which is the frequency of keys. Meanwhile, OmniSketch, the successor to Hydra, represents the SOTA in this area and provides an effective solution for analyzing multi-dimensional data streams under arbitrary predicates. It supports filtering over arbitrary combinations of attributes, with conditions specified as ranges rather than limited to exact values or wildcards. It currently only supports queries involving a single statistical attribute: the count of records that satisfy the filtering conditions.

However, in approximate query research, the "multi-attribute" nature of data streams when processing is reflected not only in multi-attribute filtering predicates, but also in aggregated results containing multiple statistical attributes (as shown in Fig. 1). Neither of the above solutions considers queries involving multiple statistical attributes, which represents a key focus of this paper on multi-attribute data stream models.

III. The Hyper-USS

This section begins with a formal definition of the subset query problem over multi-attribute data streams, followed by a detailed description of our algorithmic design to answer the problem. Symbols frequently used are listed in Table I.

TABLE I: Symbols frequently used in this paper.

Notation	Meaning
e	A distinct item in the data stream.
$h(.)$	Hash function.
d	The number of candidate positions when performing item competition and replacement.
w	The number of hash buckets for each hash function.
ID	The key value of the item.
v_i	The i^{th} statistical attribute of the item.
n	The number of item statistical attributes.
(e, V)	The item e with attribute V. Among them, $V = \langle v_1, \ldots, v_n \rangle$.
$\mathcal{B}_i[j]$	The j^{th} bucket in the i^{th} array.
$C_i(B[j])$	The i^{th} cell in the j^{th} bucket.

A. Problem Definition

We formally define the subset query problem over multi-attribute data streams, as it presents the key question we aim to answer and constitutes the main focus of this paper. The problem is broadly applicable, as many real-world problems can be abstracted as instances of this query model.

Problem 1. *Suppose the item in multi-attribute data streams can be denoted as* $(e, v_1, v_2, \cdots, v_n)$*, where* e *is the key and* $v_1, v_2, \cdots, v_n$ *are the statistical attributes. Given a target attribute* v_i*, a subset of keys* $\mathcal{S}_k$ *and an operator* f*, we apply the operator on the attribute* v_i *for any key* $k \in \mathcal{S}_k$*. The problem is to design a sketching solution that supports accurate and efficient estimation of subset queries over massive data streams, where each item carries multiple statistical attributes, under limited memory.*

To answer this problem, we introduce the Hyper-USS and describe how it supports three classic query operators: *sum*, *ratio*, and *weighted sum*, each involving multiple attributes.

- For f=[*sum*] on i^{th} attribute over subset $\mathcal{S}_k$,

$$\sum_{(e,v_1,\cdots,v_n),e\in\mathcal{S}_k} v_i$$

- For f=[*ratio*] on the i^{th} and j^{th} attributes over subset $\mathcal{S}_k$,

$$\frac{\sum_{(e,v_1,\ldots v_n),e\in S_k} v_i}{\sum_{(e,v_1,\ldots v_n),e\in S_k} v_j}$$

- For f=[*weighted sum*] on the i^{th} and j^{th} attributes over subset $\mathcal{S}_k$, with corresponding weights w_i and w_j,

$$w_i * \sum_{(e,v_1,\ldots,v_n),e\in S_k} v_i + w_j * \sum_{(e,v_1,\ldots v_n),e\in S_k} v_j$$

B. Basic Design

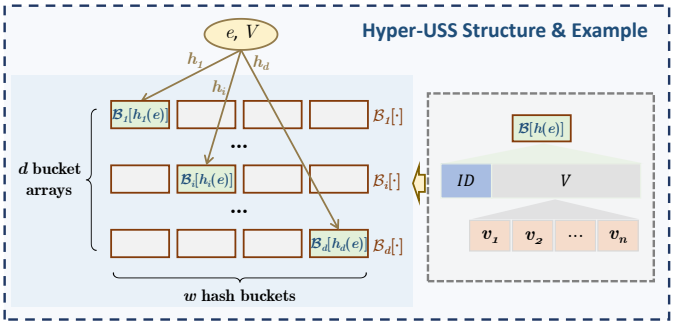


Fig. 4: Data structure & Example of Hyper-USS.

(1) Data Structure (Fig. 4). The hash table serves merely as a basic implementation structure. Its structure consists of d bucket arrays, each consisting of w buckets. Each bucket records a key and n statistical attributes. The attributes are represented as a vector $V = \langle v_1, v_2, \cdots, v_n \rangle$. Let $\mathcal{B}_i[j](1 \leqslant i \leqslant d, 0 \leqslant j \leqslant w-1)$ be the j^{th} bucket in the i^{th} array. Let $\mathcal{B}_i[j].ID$ be the recorded key in the bucket, and $\mathcal{B}_i[j].V[t](1 \leqslant t \leqslant n)$ be the t^{th} recorded attribute. Each bucket array is associated with one hash function, respectively, and we denote the corresponding hash function for the i^{th} array as $h_i(\cdot)$.

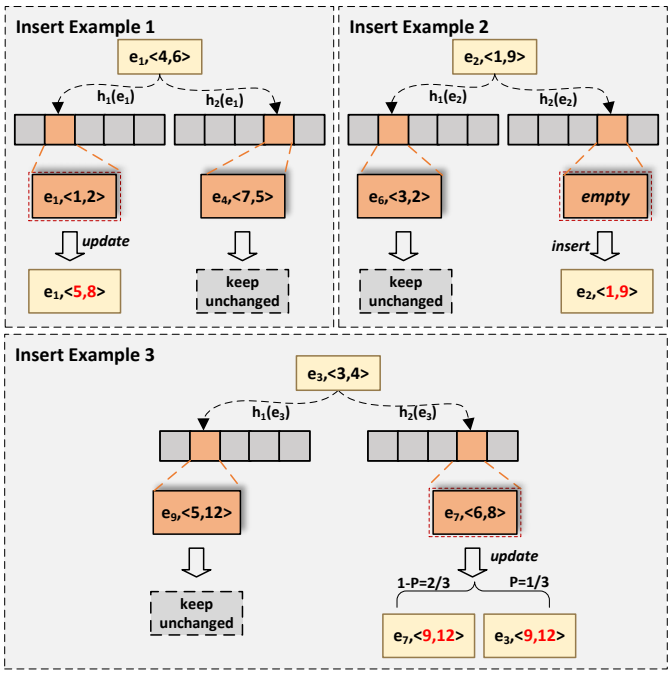


Fig. 5: Insertion examples for Hyper-USS ($n = 2, d = 2$).

(2) Insertion. Insertion strategy is the core of Hyper-USS. To insert the item (e, V), we first use d hash functions to hash the key e to d buckets $(\mathcal{B}_i[h_i(e)], 1 \leqslant i \leqslant d)$ in d arrays. In matching the key of the incoming item with existing keys in the buckets, there are three possible cases:

`Case 1:` e is recorded in $\mathcal{B}_k[h_k(e)]$, one of the d buckets. In this case, we add the attributes $v_1, \cdots, v_n$ to the bucket. To estimate the frequency of an item, which can be treated as a statistical attribute, we introduce a virtual attribute v_{n+1}. For each item, we simply set $v_{n+1} = 1$. For each dimension t, the t^{th} attribute $\mathcal{B}_k[h_k(e)].V[t]$ is increased by v_t.

`Case 2:` e is not recorded in any one of the d buckets, and at least one bucket is still empty. In such a case, we select the first empty bucket to record the incoming item.

`Case 3:` e is not recorded in any of the d buckets, and all buckets are occupied. We look for the bucket with the smallest L2 norm of values. The incoming item e and the item in the bucket are treated as competing, and we apply the Joint Variance Optimization technique (Fig. 2). Winning probabilities are set according to the probability proportional to size sampling principle [36], where items with greater statistical importance are more likely to be retained. Suppose the hashed bucket in the k^{th} array has the smallest L2 norm, we set the winning probability of the incoming item $\mathcal{P}$ as follows:

$$\mathcal{P} = \frac{\sqrt{\sum_t v_t^2}}{\sqrt{\sum_t v_t^2} + \sqrt{\sum_t \mathcal{B}_k[h_k(e)].V[t]^2}} \tag{1}$$

With probability of $\mathcal{P}$, the recorded item in $\mathcal{B}_k[h_k(e)]$ is replaced by (e, V), and then each attribute is divided by $\mathcal{P}$. Otherwise, the item in the bucket wins the competition, and each recorded attribute is divided by $1 - \mathcal{P}$.

①**Insertion Examples (Fig. 5):** As shown in the first example, we aim to insert the item $(e_1, 4, 6)$, and we use two associated

hash function to locate two buckets, respectively. The bucket in the first array matches the key, so the recorded attributes are updated. The hashed bucket in the second array is not changed. In the second example, we insert the item $(e_2, 1, 9)$. Neither of the two hashed buckets match e_2, and there exists an empty bucket, so the item is inserted to the empty bucket. In the third example, we insert the item $(e_3, 3, 4)$. Neither of the two hashed buckets match e_3, and the L2 norm of the second bucket is smaller, so we update the second bucket. The replacement probability $\mathcal{P}$ is set to $\sqrt{3^2+4^2}/(\sqrt{3^2+4^2}+\sqrt{6^2+8^2}) = 1/3$. With probability $1/3$, the key is replaced by e_3, and each attribute is divided by $1/3$. Otherwise, the key is unchanged, and all attributes are divided by $2/3$.

TABLE II: Complexity analysis of Hyper-USS for (e, V).

Operation	Time Complexity	Space Complexity
Insertion	$O(dn)$	$O(dwn)$
Query	$O(d)$	$O(d)$

②**Analysis on Computational Cost for Insertion:** We analyze the time and space complexity (shown in Table II).
Time Complexity Analysis. 1. Hashing: The e is hashed into d positions using d independent hash functions. The time complexity of hashing is $O(d)$. 2. Key Matching: The algorithm searches the d hashed buckets to check if the key of e already exists. The time complexity of this operation is also $O(d)$. 3. Handling Different Cases: If the key is found in one of the buckets, the attributes are updated in $O(n)$ time (Case 1). If an empty bucket is available, the new item and its attributes are inserted in $O(n)$ time (Case 2). If d buckets are occupied, the L2 norm of each bucket must be computed, taking $O(dn)$ time (Case 3). The probability calculation for replacement is performed in $O(n)$, and attribute updates take $O(n)$ time.

Given that Case 3 incurs the highest computational cost, the worst-case time complexity per insertion is $O(dn)$.
Space Complexity Analysis. The space complexity is determined by the data structure, which consists of d bucket arrays, each containing w buckets. Each bucket stores a key and n attributes: The total number of buckets is $d \times w$. Each bucket stores one key and n attributes, contributing to a space complexity of $O(d \cdot w \cdot (1+n))$, which simplifies to $O(dwn)$. Notably, the number of hash functions d is typically small, and w is determined by the memory size allocated before algorithm execution. The number of item attributes n significantly impacts performance, so our optimization versions (§ V) extend the approach to handle larger n.
③**Dealing with Imbalanced Attributes:** The winning probability depends on the L2 norm of n equal-weighted attributes, and all attributes contribute equally. However, in real scenarios, the values of different attributes may have different orders of magnitude. Such attributes with large values will dominate the L2 norm, thus dominating the update process and hurting the sketch performance on other attributes.

To deal with imbalanced attributes, we introduce the technique named Fair Evolution. The key design is to *normalize n attributes* before calculating the L2 norm for the winning probability. During the insertion, we compute the average of all past items on all n attributes, $A_i, 1 \leqslant i \leqslant n$. The i^{th} attribute is divided by A_i before calculating the L2 norm. Therefore,

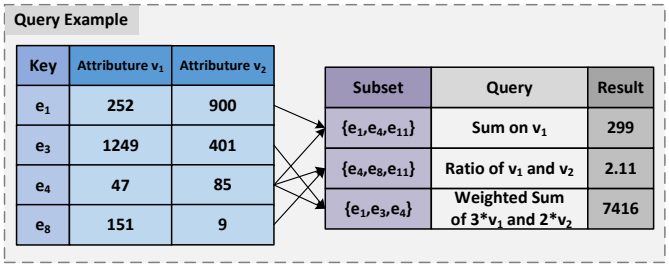


Fig. 6: A query example of Hyper-USS ($n = 2$).

the winning probability of the incoming item is set according to the following adjusted formula,

$$\mathcal{P} = \frac{\sqrt{\sum_t (v_t/A_t)^2}}{\sqrt{\sum_t (v_t/A_t)^2} + \sqrt{\sum_t (\mathcal{B}_k[h_k(e)].V[t]/A_t)^2}} \quad (2)$$

When the attributes are imbalanced, the L2 norm in the algorithm can be replaced with a normalized one to improve accuracy, while the rest of the algorithm remains unchanged.
(3) Query. We extract all non-empty buckets in the resulting sketch to query *the subset sum* and build a table with $n+1$ columns (one key and n attributes). Then we output the query result of the subset sum over the table.
Support for Subset Ratio: The ratio of subsets can be expressed as the division of the sum of the two subsets. Among them, if the subset sum serving as the denominator is zero, the query result will return '`ERROR`'.
Support for Subset Weighted Sum: Subset weighting can be expressed as the sum of weighted subsets, with fixed weights assigned to each query. The weights of each group of queries can be preconfigured according to specific needs.
①**Query Example (Fig. 6):** The table shows an example of a table built from the result sketch. To query the sum over the subset $\{e_1, e_4, e_{11}\}$ on the attribute v_1, we apply the operations on the table. For e_1 and e_4, we aggregate the first and the third rows in the v_1 column of the table. For e_{11}, as it does not appear in the table, its attributes are all estimated as 0. The query result is $252 + 47 + 0 = 299$. To query the ratio of attributes v_1 to v_2 for the subset $\{e_4, e_8, e_{11}\}$. We sum the last two rows of the table separately in the v_1 and v_2 columns, and then compute the ratio of the two sums. The query result is $(47+151)/(85+9) = 2.11$. When querying the weighted sum of attributes v_1 and v_2 for the subset $\{e_1, e_3, e_4\}$, weights are assigned as $w_1 = 3$ for v_1 and $w_2 = 2$ for v_2. We accumulate the attribute values of the first three rows, then multiply by their respective weights and sum them up. The query result is $3 * (252 + 1249 + 47) + 2 * (900 + 401 + 85) = 7416$.
②**Analysis on Computational Cost for Query:** As all attributes corresponding to the item are updated simultaneously during insertion, querying for e only requires hashing to locate the buckets, matching the ID, and acquiring all the attribute values. Since the number of locations to be checked, d, is typically small, the time and space complexity of the query operation can be considered $O(d) \approx O(1)$ (shown in Table II). For subset queries, we only need to scan the table (see Fig. 6).
③**Support for Arbitrary Predicate Combinations:** Since the ID of each item is constructed by concatenating the attributes that are allowed in filtering conditions, our algorithm

supports subset queries with arbitrary predicate combinations. For example (see Fig. 1), the ID is composed of the five-tuple of a packet: {ipSrc, ipDest, portSrc, portDest, protocol}. Users can filter keys by specifying any combination of the five attributes (*e.g.,* {ipSrc, ipDest, protocol} or {ipSrc, portDest}) and setting exact values or ranges for each selected attribute.

C. The Rationale of Hyper-USS

As the subset ratio and subset weighted sum queries can be reduced to the subset sum query, the high accuracy on subset sum estimation is the key problem. We provide the rationale behind the design of Hyper-USS.

Firstly, as pointed out by prior work [4], accurate subset sum estimation requires unbiased estimation. Applying a biased sketch, *e.g.*, CM sketch [17], to subset sum estimation, will result in unacceptably accumulated errors. The design of Hyper-USS ensures that, when two items compete for one bucket (`Case 3` in the Insertion), it still provides unbiased estimation on any attribute for both items. To this end, Hyper-USS assigns a winning probability to each item and adjusts the attributes of the selected winner based on that probability. The detailed proof is shown in § IV-A.

TABLE III: Optimization goals of different sketches. $S_i(\cdot), \widehat{S}_i(\cdot)$ denote the real and estimated sum on the i^{th} statistical attribute respectively.

Sketch	Optimization Goal
USS/CocoSketch	minimize $\sum_e \left(S_i(e) - \widehat{S}_i(e)\right)^2$
Hyper-USS	minimize $\sum_i \sum_e \left(S_i(e) - \widehat{S}_i(e)\right)^2$

Given the unbiasedness, we achieve accurate subset sum estimation by optimizing the variance of the estimation. As shown in Table III, the SOTA sketches on subset query over single statistical attributes, USS and CocoSketch, minimize the sum of estimation variance on all single keys. Hyper-USS aims to provide accurate estimation of all statistical attributes and therefore minimizes the sum of variances for all keys and attributes. It is noticeable that, for the insertion of Hyper-USS, the specific choice of the winning probability $\mathcal{P}$ does not affect the unbiasedness property. By setting the winning probability proportional to the L2 norm of all attributes, Hyper-USS accomplishes the optimization goal. Besides, selecting the bucket with the smallest L2 norm for updating also targets the optimization goal. The detailed proof is shown in § IV-B.

IV. Mathematical Analysis

In this section, we provide mathematical analysis for Hyper-USS. We first prove the unbiasedness of the subset sum estimation on any statistical attribute in § IV-A. Then we prove how the Hyper-USS achieves their optimization goals of variance optimization in § IV-B. We further provide the analysis of the error bound in § IV-C.

A. Unbiasedness of Hyper-USS

Theorem 1. *For any statistical attribute v_i, Hyper-USS provides unbiased sum estimation for any subset $\mathcal{S}$,*

$$\mathbb{E}\left[\widehat{S}_i(\mathcal{S})\right] = S_i(\mathcal{S})$$

Here $\widehat{S}_i(\cdot)$ denotes the estimated subset sum on i^{th} attribute, and $S_i(\cdot)$ denotes the real subset sum on i^{th} attribute.

Proof. We first prove that Hyper-USS gives unbiased sum estimation for any single key e on any attribute. Consider inserting the item $(e, v_1, v_2, \cdots, v_n)$. If one bucket matches e, the i^{th} attribute in the bucket will increase by v_i, and the increment of estimation for e is unbiased. If no matched bucket is found, the bucket with the smallest L2 norm is updated, and we suppose the updated bucket is $\mathcal{B}_t[h_t(e)]$. After the insertion, the expected increment of estimation for the key e on attribute v_i is,

$$\frac{v_i}{\mathcal{P}} \cdot \mathcal{P} + 0 \cdot (1 - \mathcal{P}) = v_i$$

The expected increment for the key $\mathcal{B}_t[h_t(e)].ID$ is,

$$(\frac{\mathcal{B}_t[h_t(e)].V[i]}{1-\mathcal{P}} - \mathcal{B}_t[h_t(e)].V[i]) \cdot (1-\mathcal{P}) - \mathcal{B}_t[h_t(e)].V[i] \cdot \mathcal{P} = 0$$

As a result, during the insertion, the estimated sum of any key is unbiased. For any subset S, we have

$$\mathbb{E}\left[\widehat{S}_i(\mathcal{S})\right] = \sum_{e \in \mathcal{S}} \mathbb{E}\left[\widehat{S}_i(e)\right] = \sum_{e \in \mathcal{S}} S_i(e) = S_i(\mathcal{S})$$

□

B. Variance Optimization in Hyper-USS

Analysis for the choice of $\mathcal{P}$. We first consider the basic version of Hyper-USS with only one array and one associated hash function, and discuss why the choice of winning probability $\mathcal{P}$ is theoretically optimal for the optimization goal.

Theorem 2. *In the basic version with $d = 1$, Hyper-USS minimizes the sum of variances of all keys on all attributes, shown as follows.*

$$\textit{minimize} \quad \sum_{i=1}^{n} \sum_{e} \left(S_i(e) - \widehat{S}_i(e)\right)^2 \tag{3}$$

Proof. We consider the increment of Eq. (3) for the insertion of each item $(e, v_1, \cdots, v_n)$. If one bucket matches e or there is at least one empty bucket, i^{th} attribute of item e has an increment of v_i, and the increment of Eq. (3) is 0. Otherwise, suppose the updated bucket is $\mathcal{B}[h(e)]$, and the record attributes are $u_i, 1 \leqslant i \leqslant n$. The increment variance only involves the keys of e and $\mathcal{B}[h(e)].ID$, and we have,

$$\begin{aligned}\sum_{i=1}^{n} \sum_{e} \Delta\left(S_i(e) - \widehat{S}_i(e)\right)^2 &= \sum_{i=1}^{n} \mathcal{P} \cdot \left(\left(\frac{v_i}{\mathcal{P}} - v_i\right)^2 + u_i^2\right) \\ &\quad + (1-\mathcal{P}) \cdot \left(v_i^2 + \left(\frac{u_i}{1-\mathcal{P}} - u_i\right)^2\right) \\ &= \frac{\sum_{i=1}^{n} v_i^2}{\mathcal{P}} + \frac{\sum_{i=1}^{n} u_i^2}{1-\mathcal{P}} - \sum_{i=1}^{n} v_i^2 - \sum_{i=1}^{n} u_i^2\end{aligned}$$

When setting

$$\mathcal{P} = \frac{\sqrt{\sum_i v_i^2}}{\sqrt{\sum_i v_i^2} + \sqrt{\sum_i u_i^2}}$$

the variance increment is minimized. □

Analysis for selecting the bucket with minimal L2 norm. We then analyze the general case of Hyper-USS and discuss why we chose to update the bucket with the minimal L2 norm.

Theorem 3. *In the basic version with $d > 0$, Hyper-USS minimizes the sum of the variance of all keys on all attributes, shown as follows.*

$$\textit{minimize} \quad \sum_{i=1}^{n}\sum_{e}\left(S_i(e)-\widehat{S}_i(e)\right)^2 \tag{4}$$

Proof. Suppose the incoming item is $(e, v_1, \cdots, v_n)$. According to the proof in Theorem 2, when there is no matched bucket and no empty bucket, suppose the updated bucket is $\mathcal{B}_t[h_t(e)]$, and its attributes are $u_i, 1 \leqslant i \leqslant n$. The minimal increment of Eq. (4) will be,

$$\begin{aligned}\sum_{i=1}^{n}\sum_{e}\Delta\left(S_i(e)-\widehat{S}_i(e)\right)^2 &= \frac{\sum_{i=1}^{n} v_i^2}{\mathcal{P}} + \frac{\sum_{i=1}^{n} u_i^2}{1-\mathcal{P}} - \sum_{i=1}^{n} v_i^2 - \sum_{i=1}^{n} u_i^2 \\ &= 2\cdot\sqrt{\sum_{i=1}^{n} v_i^2}\cdot\sqrt{\sum_{i=1}^{n} u_i^2}\end{aligned}$$

For the insertion of Hyper-USS, we look for the bucket with the minimal L2 norm among d hashed buckets as the updated bucket. Therefore, when $d > 0$, Eq. (4) is also minimized. □

C. Error Bound Analysis

This section presents an analysis of the error bounds associated with Hyper-USS.

Lemma 4. *During the insertion of Hyper-USS, for the updated bucket, the increment of L2 norm is no larger than the L2 norm of the incoming item. Furthermore, for a bucket which has been updated for k times, the L2 norm of the values of the bucket has an upper bound kL.*

Proof. Suppose the incoming item $(e, v_1, \cdots, v_n)$ updates the bucket $\mathcal{B}_t[h_t(e)]$ and $u_i = \mathcal{B}_t[h_t(e)].V[i]$.
If the key in the bucket matches e, the increment of L2 norm is,

$$\sqrt{\sum_{i=1}^{n}(u_i+v_i)^2} - \sqrt{\sum_{i=1}^{n} u_i^2} \leq \sqrt{\sum_{i=1}^{n} v_i^2}$$

Otherwise, recall that we have the winning probability as follows.

$$\mathcal{P} = \frac{\sqrt{\sum_{i=1}^{n} v_i^2}}{\sqrt{\sum_{i=1}^{n} v_i^2} + \sqrt{\sum_{i=1}^{n} u_i^2}}$$

If e wins the competition, the increment of L2 norm is,

$$\sqrt{\frac{\sum_{i=1}^{n} v_i^2}{\mathcal{P}^2}} - \sqrt{\sum_{i=1}^{n} u_i^2} = \sqrt{\sum_{i=1}^{n} v_i^2}$$

If the key in the bucket wins, the result L2 norm is,

$$\sqrt{\frac{\sum_{i=1}^{n} u_i^2}{(1-\mathcal{P})^2}} - \sqrt{\sum_{i=1}^{n} u_i^2} = \sqrt{\sum_{i=1}^{n} v_i^2}$$

Therefore, the increment of L2 norm is no larger than the L2 norm of incoming item, and it is obvious that after k updates, the L2 norm of the bucket has an upper bound kL. □

Lemma 5. *During the insertion of Hyper-USS, for an update to a bucket which has been updated for k times, the increment of total variance of items in the bucket has an upper bound $2kL^2$.*

Proof. By Theorem 3, the increment of total variance is,

$$\sum_{i=1}^{n}\sum_{e}\Delta\left(S_i(e)-\widehat{S}_i(e)\right)^2 = 2\cdot\sqrt{\sum_{i=1}^{n} v_i^2}\cdot\sqrt{\sum_{i=1}^{n} u_i^2} \leq 2kL^2$$

□

Lemma 6. *For a bucket which has been updated for M times, the variance of the estimated value $\hat{S}$ for an arbitrary key mapped to the certain bucket and an arbitrary attribute has an upper bound $(M+1)ML^2$.*

Proof. For key e and an arbitrary attribute j, the estimated value of Hyper-USS is $\hat{S}_j(e)$. By Lemma 5, the variance can be bounded by summing up all increments of variance for each update,

$$\begin{aligned}\mathrm{Var}\left[\hat{S}_j(e) \mid M\right] &\leq \sum_{e':h(e')=h(e)}\sum_{i=1}^{n}\mathrm{Var}\left[\hat{S}_i(e') \mid M\right] \\ &\leq \sum_{k=1}^{M} 2kL^2 = (M+1)ML^2\end{aligned}$$

□

Theorem 5. *In the basic design, the error of Hyper-USS's estimation on an arbitrary attribute of an arbitrary key can be bounded as follows, where U is the total number of inserted items and L is the upper bound of L2 norm for each inserted item.*

$$\Pr\left[|\hat{S}_j(e) - S_j(e)| \geqslant \epsilon\right] \leqslant \frac{4U^2L^2}{w^2\epsilon^2}$$

Proof. Recall that the estimated value by Hyper-USS is unbiased by Theorem 1. For each update, the probability that the inserted item is hashed to the same bucket is $p = \frac{1}{w}$. By Lemma 6, the variance of estimated value $\hat{S}_j(e)$ can by bounded,

$$\begin{aligned}\mathrm{Var}\left[\hat{S}_j(e)\right] &= \mathrm{E}\left[\mathrm{Var}\left[\hat{S}_j(e) \mid M\right]\right] + \mathrm{Var}\left[\mathrm{E}\left[\hat{S}_j(e) \mid M\right]\right] \\ &= \mathrm{E}\left[\mathrm{Var}\left[\hat{S}_j(e) \mid M\right]\right] \\ &\leq \sum_{M=0}^{U}\binom{U}{M}p^M(1-p)^{U-M}(M+1)ML^2 \\ &\leq \frac{4U^2L^2}{w^2}\end{aligned}$$

According to Chebyshev's inequality, we have

$$\Pr\left[|\hat{S}_j(e) - S_j(e)| \geqslant \epsilon\right] \leq \frac{\mathrm{Var}\left[\hat{S}_j(e)\right]}{\epsilon^2} \leq \frac{4U^2L^2}{w^2\epsilon^2}$$

□

Corollary 1. *The error of Hyper-USS's estimation on an arbitrary attribute of a subset $\mathcal{T}$ can be bounded as follows.*

$$\Pr\left[|\hat{S}_j(\mathcal{T}) - S_j(\mathcal{T})| \geqslant \epsilon\right] \leqslant \frac{4\,|\mathcal{T}|\,U^2L^2}{w^2\epsilon^2}$$

V. Speed Optimization for Hyper-USS

We propose three speed-optimized versions, each addressing a key factor that affects insertion efficiency when the number of statistical attribute n continues to grow: (1) the high computational overhead of L2 norm calculation (§ V-A), (2) the inefficiency of item positioning due to multiple hash computations (§ V-B), and (3) the insufficient use of the inherent parallelism of the algorithm (§ V-C). In data stream processing, insertion performance is often more critical than query performance. This is because streaming data arrives continuously at high speed, making insertion the bottleneck of the main processing pipeline, while queries are triggered relatively infrequently. Therefore, we focus primarily on improving insertion speed in this section.

A. Precomputed L2-Norm Optimization

In this section, we introduce Precomputed L2-Norm Optimization (PLNO), which precomputes the L2 norm of high-dimensional attributes, trading a modest increase in space complexity for a substantial improvement in data insertion throughput. Network data streams typically exhibit a skewed distribution: frequent items change rapidly, while infrequent items evolve more slowly. Effectively distinguishing between these two types is critical to improving algorithmic efficiency.

If attribute frequencies are uniformly updated and the L2 norm is recalculated for all data streams upon arrival to determine item replacement probabilities within buckets, such indiscriminate processing may significantly increase computational load, particularly for frequent items with rapidly changing attributes. Based on the insertion design of the basic version (§ III-B), we know that during the bucket competition phase, which determines the replacement position, it is primarily necessary to identify items with smaller L2 norms, which typically correspond to infrequent items.

This insight lays the foundation for our speed optimization: instead of precomputing the L2 norm for every incoming data stream, we focus only on infrequent items. To this end, we introduce a differentiated processing strategy into Hyper-USS to accelerate the computation of replacement probabilities. Specifically, before selecting replacement positions, we update and record the L2 norms of attributes only for infrequent items.

(1) Data Structure. The data structure of PLNO remains consistent with the basic version design. It consists of d bucket arrays, each containing w hash buckets. Each bucket records a key, an n-dimensional attribute vector V, the precomputed L2 norm $|V|$ of the vector, and a 1-*bit* Flag. The additional fields introduced in this version are $|V|$, which represents the L2 norm of $V = \langle v_1, \ldots, v_n \rangle$, and the Flag, which is used to distinguish between frequent and infrequent items.

(2) Insertion. Our design is shown in Algorithm 1. To insert an item (e, V), we use d hash functions to map the key of e to d buckets ($\mathcal{B}_i[h_i(e)], 1 \leqslant i \leqslant d$). When matching the key of the incoming e with those already recorded in the buckets, there are three possible cases:

`Case 1:` e is already recorded in one of the d hash buckets, specifically $\mathcal{B}_k[h_k(e)]$. We increase and update the attributes $\langle v_1, \ldots, v_n \rangle$ in the bucket $\mathcal{B}_k[h_k(e)]$, without the need to update the L2 norm. The Flag is set to false, marking that the current L2 norm is not up to date. If selected for replacement, the L2 norm must be recalculated.

`Case 2:` e is not recorded in any of the d hash buckets, and at least one bucket is still empty. We select the first empty bucket to record the newly arriving e. Upon the first insertion of an item, we update the L2 norm and set the Flag to true.

Algorithm 1: Insertion of (e, V) into bucket arrays (Optimized version: PLNO)

Input: Incoming item e, its attribute vector $V =< v_1, v_2, ..., v_n >$
Output: Updated bucket arrays with the item e inserted
Function `ReplaceCompetingItem(`*i, j, e, V*`)`**:**
  if $\mathcal{B}_i[j].Flag$ *is false* **then**
    $\mathcal{B}_i[j].|V| \leftarrow \sqrt{\sum_{t=1}^{n}(\mathcal{B}_i[j].v_t)^2}$;
  else
    Use the stored L2 norm $\mathcal{B}_i[j].|V|$;
  Calculate L2 norm of V, $|V| \leftarrow \sqrt{\sum_{t=1}^{n}(v_t)^2}$;
  Calculate winning probability $\mathcal{P}$ for e via Eq. (1)
  if *random number* $a \in [0, 1]$, $a \leq \mathcal{P}$ **then**
    Replace item in $\mathcal{B}_i[j]$ with e;
    foreach $t \in \{1, 2, \ldots, n\}$ **do**
      $\mathcal{B}_i[j].v_t \leftarrow \mathcal{B}_i[j].v_t / \mathcal{P}$;
    $\mathcal{B}_i[j].|V| \leftarrow |V|$;
    return;
  else
    foreach $t \in \{1, 2, \ldots, n\}$ **do**
      $\mathcal{B}_i[j].v_t \leftarrow \mathcal{B}_i[j].v_t / (1 - \mathcal{P})$;
    return;
  return;
for $i \leftarrow 1$ **to** d **do**
  $j \leftarrow h_i(e)$**;**
  Record $\mathcal{B}_i[j]$ in $BucketRecords[i]$;
  if *key of the item* e *matches key in* $\mathcal{B}_i[j]$ **then**
    foreach $t \in \{1, 2, \ldots, n\}$ **do**
      $\mathcal{B}_i[j].v_t \leftarrow \mathcal{B}_i[j].v_t + v_t$;
    $\mathcal{B}_i[j]$.Flag $\leftarrow$ false;
    return;
  else if *there is an empty bucket is* $\mathcal{B}_i[j]$ **then**
    Store (e, V) in the empty bucket of $\mathcal{B}_i[j]$;
    return;
Find the bucket $BucketRecords[k]$ with the smallest L2 norm of attributes within $BucketRecords$;
`ReplaceCompetingItem(`*k, $h_k(e)$, e, V*`)`;
return;

`Case 3:` e is not recorded in any of the d hash buckets, and all buckets are occupied, we will find the bucket with the smallest L2 norm of its values. Then, e uses the previously mentioned Joint Variance Optimization technique to compete with the item in that bucket with the smallest L2 norm. Notably, here we adjust the strategy so that it is not always necessary to ensure the accuracy of the L2 norms of all items. Only after the replacement position has been selected do we need to compute and update the L2 norms of the two

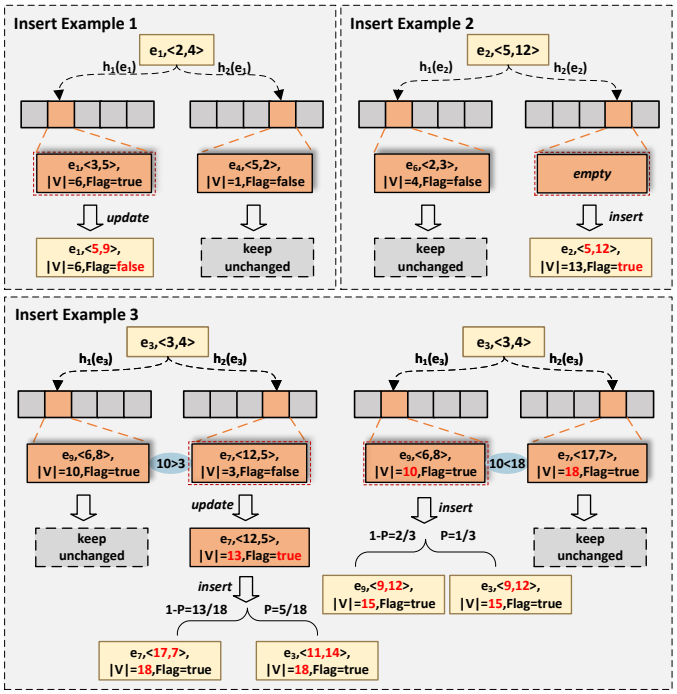


Fig. 7: Insertion examples for PLNO ($n = 2, d = 2$).

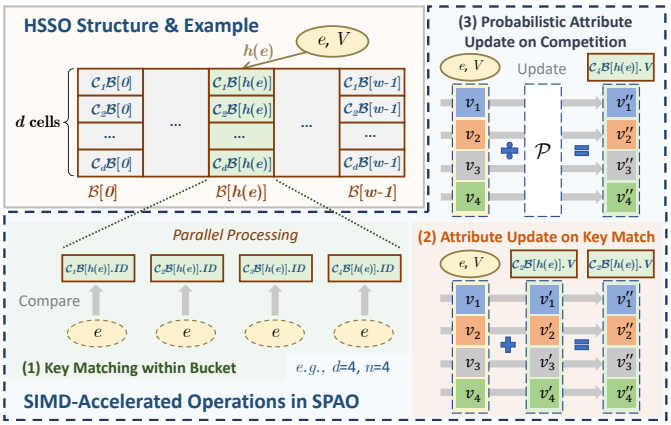


Fig. 8: Data structure & examples of HSSO and SPAO. $v_1, v_2, \ldots$ represent the attribute values carried by the incoming item e; v'_1, v'_2, $\ldots$ denote the attribute values stored in the cells matched by e; v''_1, v''_2, $\ldots$ indicate the updated attribute values in the corresponding cell during the competition.

competing items. This deviation in the L2 norm does not affect the unbiasedness of the technique when searching for a replacement in the d hash buckets.

Insertion Examples (Fig. 7): We present three examples of inserting items where the number of attributes is $n = 2$, and the number of hash functions is $d = 2$. In the first example, we insert the item $(e_1, 2, 4)$. The bucket in the first array corresponds to the key, so its attribute information V is updated. In this case, the insertion of e_1 does not update the $|V|$, and the Flag is set to false. In the second example, we insert the item $(e_2, 5, 12)$. As neither bucket matches the key and one is empty, e_2 is inserted into the empty bucket, where V and $|V|$ are updated, with the Flag set to true.

In the third example, we insert the item $(e_3, 3, 4)$. Neither of the two hash buckets matches the key of e_3 , and both are occupied. The algorithm then selects the bucket in the second array with the smallest recorded $|V|$. Two cases may occur: First, if the recorded $|V|$ accurately reflects the true value (*i.e.*, Flag is true), the item can be replaced as described earlier. Alternatively, if the recorded $|V|$ underestimates the actual value, the existing item may be incorrectly selected for replacement. In this case, the new item e_3 replaces it with a probability $\mathcal{P}$. However, since $|V|$ is updated during replacement, this mistake will be corrected in future updates (as illustrated in Insert Example 3 of Fig. 7). This example highlights the good fault tolerance of our algorithm.

B. Hash Simplification Structure Optimization

In this section, we propose the second speed-optimized version of Hyper-USS, called Hash Simplification Structure Optimization (HSSO). Our engineering analysis indicates that the main performance bottleneck of Hyper-USS lies in the computation of hash functions. The frequent use of multiple hash functions results in prolonged processing time, accounting for over 50% of the total data insertion time. Unlike existing sketch solutions that build one or more sketches for each attribute and update all of them upon the arrival of each data item, our design updates only one sketch. Under this structure, the cost of computing multiple hashes constitutes a significant portion of the total insertion time. To address this issue, HSSO reduces the number of hash functions used and relies on only a single hash function to satisfy the algorithm's process requirements.

(1) Data Structure (Fig. 8). In the HSSO, a single hash function is employed, which has been switched from the original BobHash to MurmurHash, known for its high performance and low collision rate. The data structure is comprised of w hash buckets, each containing d cells. The information recorded in each cell is consistent with that in each bucket of PLNO. We define $C_i(B[j])$ (where $1 \leqslant i \leqslant d, 0 \leqslant j \leqslant w-1$) as the i^{th} cell in the j^{th} bucket. All w hash buckets share the same hash function, denoted as $h(\cdot)$.

(2) Insertion. To insert an item (e, V), we now utilize a solitary hash function $h(\cdot)$ to associate the key of e with a specific bucket $B[h(e)]$ among w hash buckets. When comparing the key of the incoming e with the keys of items already recorded in the d cells of the bucket:

`Case 1:` If e is already recorded in one of the d cells, the algorithm updates the corresponding attribute V without recalculating the L2 norm at this stage. The Flag is set to false to indicate that the L2 norm is outdated. If this cell is later selected for replacement, the L2 norm will be recalculated.

`Case 2:` If e is not recorded in any of the d cells, and there is at least one empty bucket available, we opt to record e in an unoccupied cell. Notably, upon the initial insertion, the L2 norm is refreshed, and the Flag is set to true.

`Case 3:` If e is not recorded in any of the d cells and all cells within the bucket are filled, we identify the cell with the smallest L2 norm. e is put in contention with the minimum item in the cell, following a process similar to PLNO.

As outlined in Algorithm 2, the algorithm employs a single

Algorithm 2: Insertion of (e, V) into buckets (Optimized version: HSSO)

Input: Incoming item e, its attribute vector $V =< v_1, v_2, ..., v_n >$
Output: Updated buckets with the item e inserted
Function `ReplaceCompetingItem(i, j, e, V)`:
  if $\mathcal{C}_i(\mathcal{B}[j]).Flag$ *is false* **then**
    $\mathcal{C}_i(\mathcal{B}[j]).|V| \leftarrow \sqrt{\sum_{t=1}^{n}(\mathcal{C}_i(\mathcal{B}[j]).v_t)^2}$;
  else
    Use the stored L2 norm $\mathcal{C}_i(\mathcal{B}[j]).|V|$;
  Calculate L2 norm of V, $|V| \leftarrow \sqrt{\sum_{t=1}^{n}(v_t)^2}$;
  Calculate winning probability $\mathcal{P}$ for e via Eq. (1)
  if *random number* $a \in [0, 1]$, $a \leq \mathcal{P}$ **then**
    Replace item in $\mathcal{C}_i(\mathcal{B}[j])$ with e;
    foreach $t \in \{1, 2, \ldots, n\}$ **do**
      $\mathcal{C}_i(\mathcal{B}[j]).v_t \leftarrow v_t/\mathcal{P}$
    $\mathcal{C}_i(\mathcal{B}[j]).|V| \leftarrow |V|$ **return**;
  else
    foreach $t \in \{1, 2, \ldots, n\}$ **do**
      $\mathcal{C}_i(\mathcal{B}[j]).v_t \leftarrow \mathcal{C}_i(\mathcal{B}[j]).v_t/(1-\mathcal{P})$
    return;
  return;
$j \leftarrow h(e)$;
for $i \leftarrow 1$ **to** d **do**
  Record $\mathcal{C}_i(\mathcal{B}[j])$ in $CellRecords[i]$;
  if *key of* e *matches key in* $\mathcal{C}_i(\mathcal{B}[j])$ **then**
    foreach $t \in \{1, 2, \ldots, n\}$ **do**
      $\mathcal{C}_i(\mathcal{B}[j]).v_t \leftarrow \mathcal{C}_i(\mathcal{B}[j]).v_t + v_t$;
    $\mathcal{C}_i(\mathcal{B}[j])$.Flag $\leftarrow$ false;
    return;
  else if *there is an empty cell in* $\mathcal{C}_i(\mathcal{B}[j])$ **then**
    Store (e, V) in the empty cell of $\mathcal{C}_i(\mathcal{B}[j])$;
    Calculate L2 norm of V for e,
     $\mathcal{C}_i(\mathcal{B}[j]).|V| \leftarrow \sqrt{\sum_{t=1}^{n}(v_t)^2}$;
    $\mathcal{C}_i(\mathcal{B}[j])$.Flag $\leftarrow$ true;
    return;
Find the cell $CellRecords[k]$ with the smallest L2 norm of attributes within $CellRecords$;
`ReplaceCompetingItem(k, j, e, V)`;
return;

hash function in place of the previous d hash functions. Concerning data insertion, the operations conducted on hash buckets in the PLNO are translated to cell operations in the HSSO, with the rest of the process remaining consistent.

C. SIMD Parallel Acceleration Optimization

In this section, we introduce the SIMD Parallel Acceleration Optimization (SPAO) version, which fully exploits the inherent support of our algorithm for consecutive memory access in order to adapt to the Single Instruction, Multiple Data (SIMD) parallel strategy [27] for performance acceleration.

Parallel processing strategies typically rely on data being stored in contiguous memory. The single-hash optimization based on HSSO aggregates relevant data into the same bucket, allowing them to be stored in contiguous memory and thereby enhancing the inherent parallelism of the algorithm. The core advantage of SIMD lies in its ability to exploit vector instruction sets available in modern processors (*e.g.*, SIMD instruction sets such as the AVX instruction set from Intel and the NEON instruction set from ARM) for efficient parallel execution and data-intensive computation. Combined with the structural characteristics of our algorithm, this further improves insertion performance. Specifically, SPAO leverages the SIMD strategy to optimize computational efficiency by parallelizing three main parts of our algorithm (shown in Fig. 8): (1) matching the key of the new item with the d cells in the bucket, (2) updating the corresponding attributes if a match is found, and (3) probabilistically updating each attribute with probability $\mathcal{P}$ or $1-\mathcal{P}$ during a competition.

It is important to note that this optimization is only effective when the execution environment supports it. That is, the CPU could provide the necessary SIMD instruction sets. Compared to previous versions, this optimization achieves the best performance when a large number of attributes can be processed in parallel and memory alignment is satisfied, which is constrained by the width of SIMD registers. For example, suppose the processor supports AVX or AVX2 (with 256-bit registers) and each attribute is 32 $bits$ in size. In that case, the performance gain is most significant when the number of updated attributes is a multiple of 8, allowing fully aligned batch updates without incurring additional overhead. This is confirmed in our subsequent experimental evaluation (§ VI-B).

VI. Experimental Results

We conduct extensive experiments to compare Hyper-USS and its speed-optimized versions with the SOTA sketch solutions [3]–[6] on subset queries over multi-attribute data streams, aiming to answer the following questions:
Q1: How does the insertion throughput of our methods compare with USS [3] and CocoSketch [4] when multiple statistical attributes are involved (§ VI-B);
Q2: How does the accuracy of our methods compare with USS and CocoSketch when multiple statistical attributes are involved (§ VI-C);
Q3: How is the performance of our methods affected by different parameter settings (§ VI-D);
Q4: How does our algorithm perform when subset query support is extended to arbitrary predicate combinations, as filtering conditions compared with Omnisketch [5], [6] (§ VI-E).

In particular, OmniSketch supports subset queries with arbitrary predicate combinations (§ II-B) by maintaining a separate sketch for each filtering attribute. To straightforwardly support multiple statistical attributes, one needs to build a separate OmniSketch instance for each. Since each instance already contains multiple sketches, this greatly reduces the practicality of the algorithm. Therefore, for queries involving multiple statistical attributes, we adopt USS and CocoSketch as baselines, while OmniSketch remains the best candidate for supporting arbitrary predicates over one statistical attribute.

A. Experimental Setup

Implementation: We implement Hyper-USS and its competitors, and the code is open-sourced on GitHub [28]. As in

previous work [3], we implement a throughput-enhanced USS with a hash table and a double-linked list, because the naive USS is too slow. In the optimized USS, the hash table is used to accelerate the process of checking whether and where a key is stored in the data structure, and the double link list is used to accelerate finding the minimal bucket by sorting buckets. For CocoSketch, we use the recommended parameters in [4]. For OmniSketch, we follow the experimental settings and datasets used in [5], [6]. In the default configuration for Hyper-USS and its speed-optimized versions, the parameter d is set to 4. And we set the key length to 16 bytes and the length of each statistical attribute to 4 bytes. The number of attributes[3] carried by each data item is denoted as n.

Computation Platform: We conduct all the experiments on a single-socket CPU server (Intel(R) Core(TM) i9-10980XE CPU@3.00GHz) with a total of 36 cores (18 cores per socket with 2 threads per core), accompanied by 128GB of memory. The CPU supports a range of SIMD instruction sets, including SSE (128-bit registers) and AVX/AVX2 (256-bit registers).

Datasets: We use one synthetic and three real-world datasets.

(1) The Synthetic dataset. The keys in the synthetic dataset are randomly generated integers. To simulate the heavy-tailed distribution in real-world workload, the frequency of keys follows the Zipf distribution [46] with skewness 1.5 according to [23], [47]. The dataset contains 50M items, each with a configurable key size and a configurable number of statistical attributes. Attribute values are sampled from exponential distributions with means ranging from 1 to 16.

(2) The Criteo dataset. The Criteo dataset [48] contains feature values and click feedback for millions of display ads over 24 days. The Criteo dataset contains 26 categorical features and 13 integer features. USS [3] uses this dataset to evaluate subset queries. We select 4 categorical features as the key and 13 integer features as attributes and use the first 50M items.

(3) The CAIDA dataset. The CAIDA dataset [49] contains one hour of anonymous network traces collected from the Equinix-Chicago monitor in 2018. We use the source IP and destination IP as the ID of items and use two statistical attributes: the packet size and the packet interval. We use 1-minute intervals, which contain around 27M items and 85K distinct items.

(4) The SNMP dataset. The SNMP dataset [5] contains records collected from the wireless network of Dartmouth College during the fall of 2003, and contains 8.2 million records with 11 fields. All 11 fields are selected as the key. A statistical attribute is defined as the count of keys matching filters over selected subsets of these fields.

Metrics: We use the following metrics to evaluate insertion efficiency and estimation accuracy.

(1) Throughput (Mips). Millions of items per second. The throughput numbers are the median value among 5 independent trials. Insertion time and throughput are reciprocal, assuming consistent units.

(2) Average Relative Error (ARE). $\frac{1}{|\mathbf{\Psi}|}\sum_{\mathcal{S}\in\mathbf{\Psi}}\frac{|f(\mathcal{S})-\widehat{f}(\mathcal{S})|}{f(\mathcal{S})}$, where $f(\mathcal{S})$ is the ground truth, $\widehat{f}(\mathcal{S})$ is the output query result, and $\mathbf{\Psi}$ is the query set.

(3) Average Absolute Error (AAE). $\frac{\sum_{\mathcal{S}\in\mathbf{\Psi}}\left|f(\mathcal{S})-\widehat{f}(\mathcal{S})\right|}{|\mathbf{\Psi}|}$.

[3]Unless stated otherwise, "attributes" refer to statistical attributes.

B. Throughput Evaluation

We evaluate the insertion throughput of Hyper-USS and its three speed-optimized versions by comparing them with two SOTA sketches, CocoSketch and USS, under subset queries involving multiple statistical attributes on the Synthetic and Criteo datasets. We evaluate the insertion throughput of algorithms under varying memory sizes in both high-dimensional and low-dimensional settings, determined by the number of attributes. Settings with more than 8 attributes are considered high-dimensional, and those with fewer are low-dimensional. This threshold is based on the 256-bit SIMD registers used in our implementation, which can process 8 or any multiple of 8 attributes in parallel. We validate our tests under both common memory settings (around 300–700 KB) [3], [4], [20]–[22], [26] and small memory settings (around 20–60 KB). The evaluation under the small memory settings is further conducted to maximize the likelihood that the sketch runs entirely within the L1/L2 cache, thereby improving computational efficiency.

The tests show that Hyper-USS and its speed-optimized versions achieve higher insertion throughput. For high-dimensional attribute data insertion, SPAO performs best when the number of attributes is a multiple of 8, as this fully utilizes the 256-bit SIMD registers; in other cases, HSSO proves to be the most efficient, while PLNO offers a balanced trade-off between efficiency and accuracy (§ VI-C).

1) Experiments on the Synthetic Dataset:

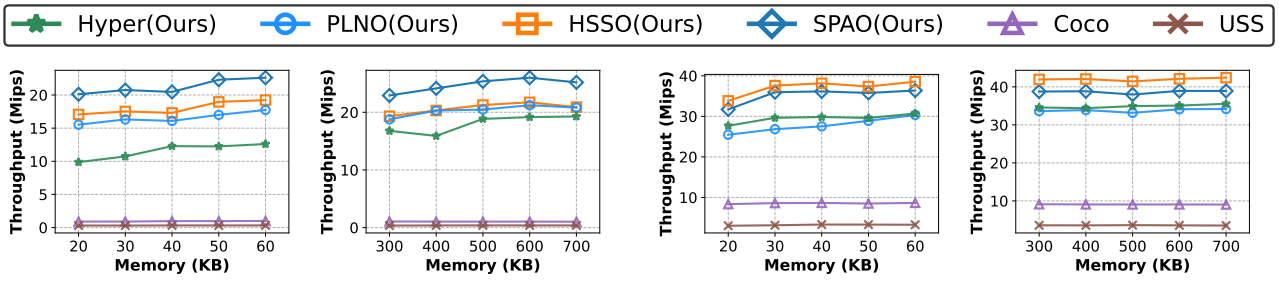
Since the number of attributes is configurable, we generate the dataset with up to 32 statistical attributes.

High-Dimensional Attribute Data Insertion(Fig. 9(a)): Experimental results show that the speed-optimized versions of Hyper-USS achieve significant acceleration under small memory. Specifically, when the number of statistical attributes n=32, the average throughput of the basic version and three speed-optimized versions reaches nearly 12.60 Mips, 17.74 Mips, 19.23 Mips, and 22.61 Mips, respectively. In comparison, CocoSketch and USS achieve only 1.04 Mips and 0.383 Mips. As shown in Table IV, the speedup of Hyper-USS over CocoSketch and USS ranges from 11.12× to 20.74× and from 31.90× to 58.03×, respectively. Additionally, under common memory ranging from 300 to 700 KB, our algorithms achieve speedups ranging from 17.18× to 23.52× over CocoSketch and from 49.29× to 66.82× over USS.

TABLE IV: Throughput Improvement over SOTA (*20-60 KB*).

Algorithm Comparison		**Throughput Improvement**	
		Synthetic Dataset	
Our	Baseline	*n=32*	*n=4*
Hyper-USS	CocoSketch	11.12×	2.58×
	USS	31.90×	8.34×
PLNO	CocoSketch	16.06×	2.51×
	USS	45.31×	8.15×
HSSO	CocoSketch	17.49×	3.53×
	USS	49.21×	10.81×
SPAO	CocoSketch	20.74×	3.22×
	USS	58.03×	10.01×

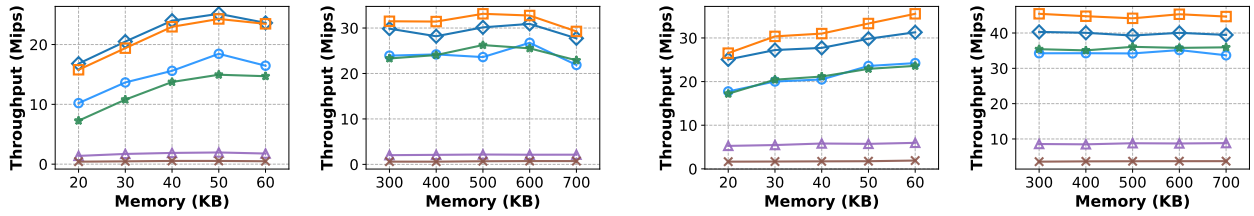
Low-Dimensional Attribute Data Insertion(Fig. 9(b)): Experimental results on low-dimensional attribute insertion under both small and common memory show that HSSO achieves the best optimization performance. This is because the accelera-



(a) High-dimensional attributes ($n = 32$)

(b) Low-dimensional attributes ($n = 4$)

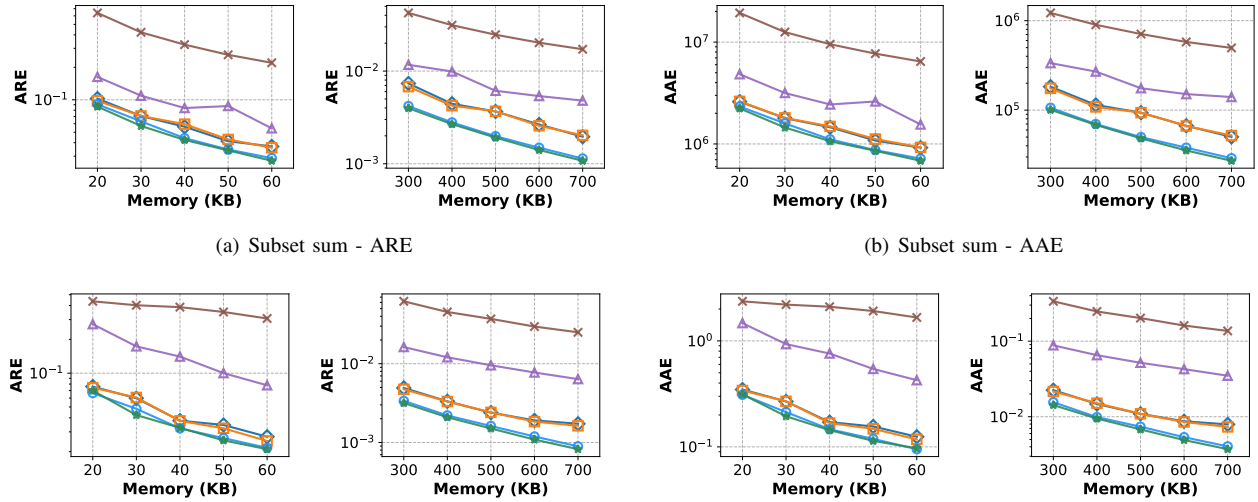
Fig. 9: Experimenting for insertion throughput on the Synthetic Dataset.



(a) High-dimensional attributes ($n = 13$)

(b) Low-dimensional attributes ($n = 4$)

Fig. 10: Experimenting for insertion throughput on the Criteo Dataset.



(a) Subset sum - ARE

(b) Subset sum - AAE

(c) Subset ratio - ARE

(d) Subset ratio - AAE

Fig. 11: Experiments for the subset query on the Synthetic dataset.

tion of SPAO depends on a large and well-aligned number of attributes. When the number of attributes is smaller than what the SIMD register can process at once, additional overhead is incurred to manage batch processing, making SPAO less efficient than HSSO in such cases. The insertion throughput of HSSO reaches 42.12 Mips, while CocoSketch and USS achieve 9.12 Mips and 3.62 Mips, respectively. This increases the throughput by 3.62× and 10.64×, respectively.

2) Experiments on the Real-World Dataset:

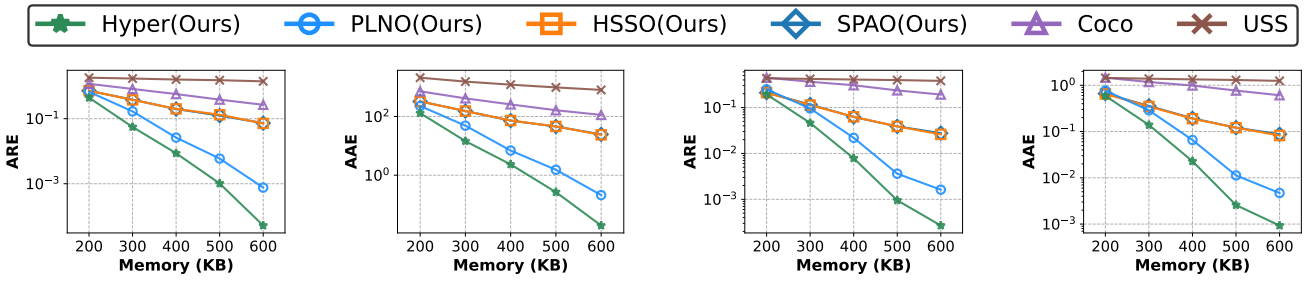
Following [3], we use the Criteo dataset as a representative real-world dataset for evaluation in this subsection.

High-Dimensional Attributes Data Insertion(Fig. 10(a)): For high-dimensional attribute data under small memory, SPAO and HSSO achieve nearly identical throughput, reaching approximately 25.04 Mips. In contrast, CocoSketch and USS only reach 1.95 Mips and 0.54 Mips, corresponding to improvements of 11.83× and 45.40×, respectively. It is worth noting that the dataset contains at most 13 attributes, which is not a multiple of 8. As a result, the advantage of SPAO is less pronounced, and it even performs slightly worse than HSSO under common memory.

Low-Dimensional Attributes Data Insertion(Fig. 10(b)): For low-dimensional attribute data processed under common memory conditions, the HSSO that we proposed achieves a throughput of 46.02 Mips, representing improvements of 4.13× and 11.26×, respectively.

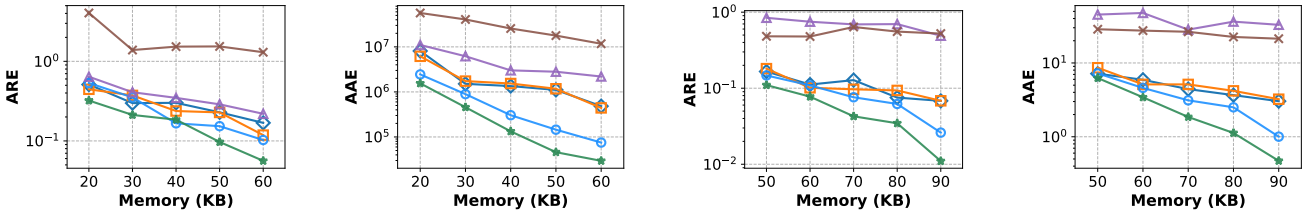
C. Accuracy Evaluation

We evaluate the accuracy of algorithms on both subset queries with multiple attributes and point queries. We compute the average metric across all trials and all statistical attributes on selected items. It is worth noting that large AAE values may result from the high magnitudes of statistical attributes in some datasets. Our evaluation focuses on the relative AAE differences across algorithms. ARE remains the primary accuracy metric, particularly when its value is below 1. In



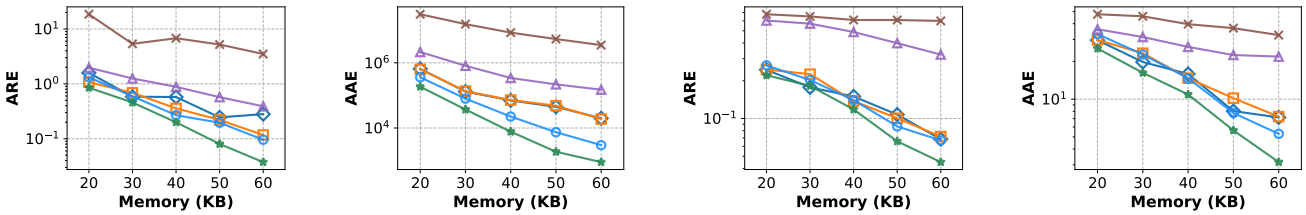
(a) Point sum - ARE (b) Point sum - AAE (c) Point ratio - ARE (d) Point ratio - AAE

Fig. 12: Experiments for the point query on the Synthetic dataset.



(a) Subset sum - ARE (b) Subset sum - AAE (c) Subset ratio - ARE (d) Subset ratio - AAE

Fig. 13: Experiments for the subset query on the Criteo dataset.



(a) Point sum - ARE (b) Point sum - AAE (c) Point ratio - ARE (d) Point ratio - AAE

Fig. 14: Experiments for the point query on the Criteo dataset.

subset queries, only part of the item ID is matched, with the rest treated as wildcards; in point queries, the entire ID is matched.

1) Experiments on the Synthetic Dataset:

We evaluate accuracy on the synthetic dataset, where each item carries 32 statistical attributes (*i.e.*, n=32).

Subset query (Fig. 11(a) - 11(d)): We use subset sum and subset ratio estimation as representatives of the subset query for evaluation. Experimental results show that Hyper-USS is more accurate for subset queries than the SOTA sketch solutions when handling multiple statistical attributes simultaneously. For subset sum estimation, Hyper-USS lowers the ARE and AAE by approximately 38% and 46%, respectively, compared to CocoSketch. It lowers both the ARE and AAE by one order of magnitude compared to USS. For subset ratio estimation, Hyper-USS lowers both the ARE and AAE by nearly one order of magnitude compared to CocoSketch, and by more than one order of magnitude compared to USS. It, along with its speed-optimized versions, outperforms CocoSketch and USS in ARE and AAE.

Point query (Fig. 12(a) - 12(d)): Similarly, we use point sum and point ratio estimation as examples of the point query with multiple statistical attributes for evaluation. The experimental results show that Hyper-USS achieves the highest accuracy for point queries. Meanwhile, its three speed-optimized versions also deliver higher accuracy than CocoSketch and USS. As the memory increases, the accuracy with the Hyper-USS improves more significantly. As the memory increases to 600 KB, Hyper-USS achieves ARE and AAE that are more than one order of magnitude lower than those of the two baselines.

2) Experiments on the Real-World Dataset:

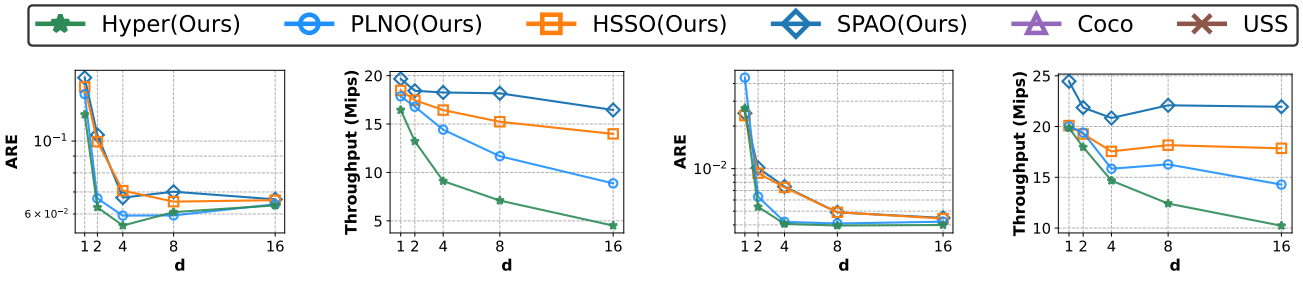
We evaluate the accuracy of subset queries and point queries using the Criteo dataset, where each item carries 4 statistical attributes (*i.e.*, n=4). Specifically, subset queries are assessed on the heavy hitter subsets, while point queries are evaluated on the individual heavy hitters [30], [31], [50].

Subset query (Fig. 13(a) - 13(d)): Experimental results indicate that Hyper-USS is the most accurate method among the compared algorithms. For subset sum and ratio estimation, Hyper-USS lowers the ARE by 57% and 92% compared to CocoSketch, and by more than one order of magnitude compared to USS, respectively. And the AAE shows a consistent trend with ARE. Meanwhile, the three speed-optimized versions also achieve higher accuracy than the compared algorithms.

Point query (Fig. 14(a) - 14(d)): For point sum estimation, Hyper-USS lowers the ARE by 71% and more than one order of magnitude compared to CocoSketch and USS, respectively. For point ratio estimation, it lowers the ARE by 47% and 51% compared to the two baselines, respectively.

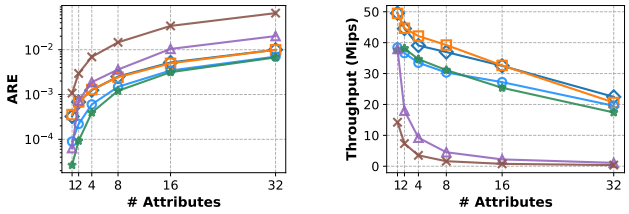
D. Parameter Settings

We evaluate the effect of different parameters on our algorithms using the synthetic dataset by default, unless otherwise specified, to explore a wider range of parameter settings.



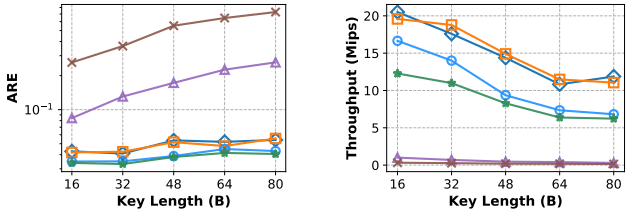
(a) small memory = 30 KB (b) common memory = 300 KB

Fig. 15: Experiments on the number of candidate positions d.



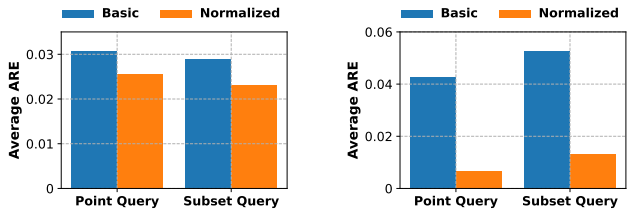
(a) ARE (b) Throughput

Fig. 16: Experiments on the number of statistical attributes.



(a) ARE (b) Throughput

Fig. 17: Experiments on the key length.



(a) The Criteo dataset (b) The CAIDA dataset

Fig. 18: Experiments on normalization of attribute values.

The effect of d (Fig. 15(a) - 15(b)): We vary the number of candidate positions d from 2^0 to 2^4 in the Hyper-USS, *i.e.*, 1 to 16. Set $n = 32$ and set the small memory size to 30 KB and the common memory size to 300 KB for the experiment. The experimental results show that the parameter d in Hyper-USS and its speed-optimized versions is a trade-off between accuracy and throughput. When d increases from 1, the accuracy first decreases and then slightly increases. When the memory size is small, the best results are achieved at $d = 4$ for Hyper-USS and its speed-optimized algorithm, with accuracies of 0.012, 0.023, 0.034, and 0.047, and corresponding throughputs of 7.23 Mips, 7.23 Mips, 7.23 Mips, and 7.23 Mips. For common memory sizes, the accuracies for SPAO and HSSO are 0.0049 at $d = 8$, corresponding throughputs of 22.09 Mips and 18.16 Mips. Both cases can achieve a balance between accuracy and insertion throughput.

The effect of the attribute number (Fig. 16(a) - 16(b)): We fix the key length to 16 bytes and vary the number of attributes from 2^0 to 2^5, *i.e.*, 1 to 32. When the number of attributes is 32, the ARE of Hyper-USS is 0.0067, while those of CocoSketch and USS are 0.0201, 0.0655, respectively. The throughput of Hyper-USS decreases as the number of attributes grows, but maintains higher than 17.41 Mips. We propose Hyper-USS and its speed-optimized versions, ordered by throughput in descending order: SPAO, HSSO, PLNO, and Hyper-USS. This is consistent with the results in § VI-B. When the number of attributes is 1, Hyper-USS degenerates to CocoSketch. However, even with only two attributes, Hyper-USS still outperforms CocoSketch, achieving nearly one order of magnitude lower ARE and 2.15× higher throughput.

The effect of the key length (Fig. 17(a) - 17(b)): We fix the number of attributes to 32 and vary the key length in the synthetic dataset from 16 bytes to 80 bytes. The experimental results indicate that as the key length increases, the advantage of Hyper-USS in accuracy becomes more significant. When the key length is 80 bytes, the ARE of Hyper-USS is 0.040, while those of CocoSketch and USS are 0.262 and 0.729, respectively. The throughput of Hyper-USS drops as key length grows, but is still higher than 6.23 Mips. The drop rate is higher than SOTA because they are too slow.

The effect of attribute value normalization (Fig. 18(a) - 18(b)): We compare the effect of using Eq. 1 and Eq. 2 to compute the replacement probability $\mathcal{P}$ in Hyper-USS on real-world datasets with imbalanced statistical attributes. For the Criteo, to simulate imbalanced statistical attributes, we scale one attribute of each item by 10^5. For the CAIDA, the value of packet interval is around 10^{-6}, and the value of packet size is usually 10^3; thus, two statistical attributes are already imbalanced. With imbalanced statistical attributes, ARE is a more suitable metric for accuracy. In each trial, we perform a subset query and a point query, respectively, and then record the average ARE. We conduct 1000 trials and calculate the average of ARE. The results show that the normalized setting performs better when attributes are imbalanced. In both point query and subset query, the mean ARE is reduced by up to 21% on the Criteo dataset and 75% on the CAIDA dataset, indicating the normalized setting improves accuracy.

E. Evaluation on Extended Tasks

To more comprehensively support subset queries over multi-attribute data streams, we allow arbitrary combinations of query predicates to serve as filtering conditions. Each condition can be specified using an exact value, a wild-

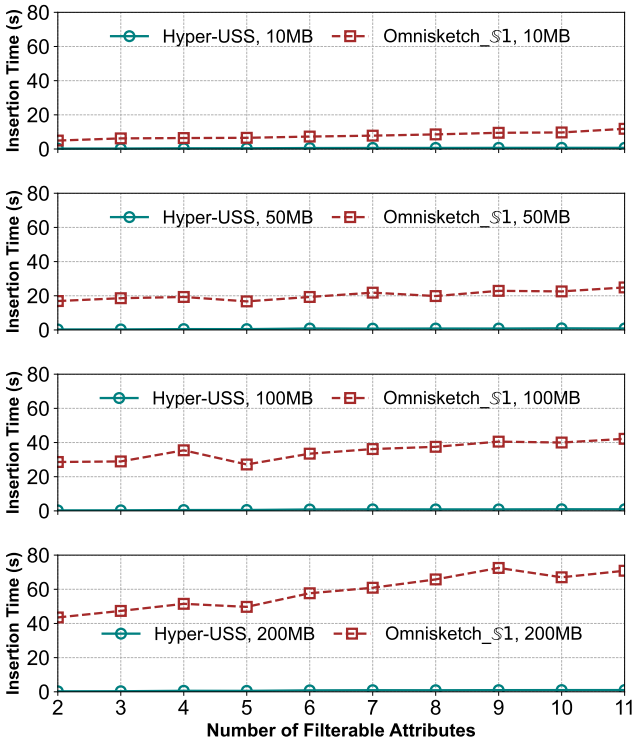


Fig. 19: Experiments on insertion time under varying memory.

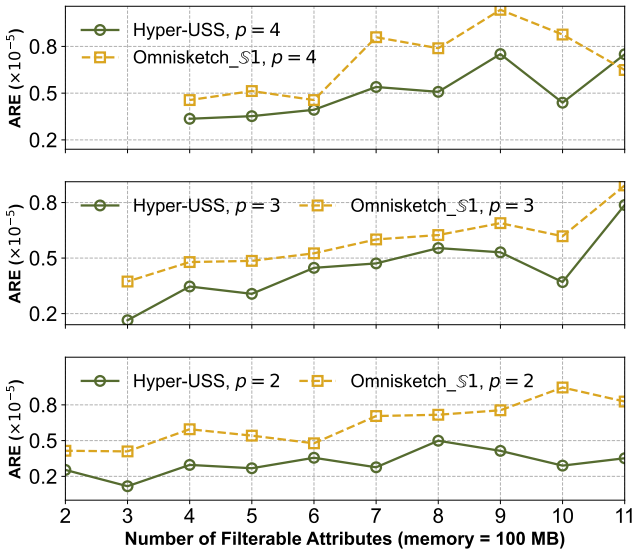


Fig. 20: Experiments on accuracy with different p.

card, or a range. OmniSketch represents the SOTA solution in this area (§ II-B). We compare our approach with OmniSketch(OmniSketch_𝕊1, its best-performing version) in terms of insertion time, estimation accuracy, and query execution time using the SNMP dataset. The query task is to retrieve the total number of items that satisfy the filtering conditions under different predicate combinations. Given that OmniSketch supports only a single statistical attribute as the aggregated result, we adopt the basic version of Hyper-USS as our representative algorithm. All experimental settings and the dataset follow those used in the OmniSketch work [5], [6].

The SNMP dataset contains up to 11 filterable attributes that can be used for predicate composition. We use p to denote the number of attributes in a predicate combination that are assigned either exact values or ranges. For each of these p attributes, whether it is assigned a value or a range depends on whether the attribute supports range specification. The remaining attributes are treated as wildcards (*). **For example**, when evaluating subset queries over a predicate combination consisting of 5 filterable attributes, $p = 2$ means that any 2 attributes are specified with exact values or ranges, while the remaining 3 are treated as wildcards (*). In our algorithm, the ID of each item is composed of all attributes that support querying in the filtering conditions, and only a single sketch is maintained. In contrast, Omnisketch needs to build a separate sketch for each filterable attribute.

Support for Arbitrary predicate combinations (Fig. 19 - 20: Fast insertion is the key requirement for stream processing and summarization. In terms of insertion time, compared to OmniSketch, our method improves the performance approximately 94.95% to 98.56% (*i.e.*, reduces the time from 4.95 to 70.84 seconds to just 0.25 to 1.02 seconds) (shown in Fig. 19), as it only needs to update a single sketch per item, whereas OmniSketch must update multiple sketches simultaneously. As for query accuracy, measured by the ARE, the difference between the two algorithms is small, as shown in Fig. 20. And our algorithm achieves slightly better accuracy than Omnisketch. We also compare query execution times under different memory sizes with different p (Table V). Although our algorithm takes slightly longer for queries compared to OmniSketch, query operations occur far less frequently than insertions in practice. Therefore, query operations usually have lower performance requirements than insertions.

TABLE V: Query Execution Time in **milliseconds**, for queries with different numbers of predicates p.

Algorithm (Memory)		**Predicate Count p**			
		2	**4**	**6**	**8**
Hyper-USS	10MB	20.02	17.37	16.68	15.21
	50MB	24.32	21.80	22.56	19.71
	100MB	30.62	26.92	24.58	24.04
	200MB	33.66	30.18	28.03	28.61
OmniSketch	10MB	0.18	0.13	0.12	0.09
	50MB	0.98	0.79	0.68	0.44
	100MB	2.02	1.61	1.25	1.13
	200MB	6.38	4.33	2.85	2.64

F. Case Study on Network Measurement

To further validate the practicality of our algorithm, we conduct a case study on network measurement using the CAIDA dataset [49], with query tasks including subset sum and subset ratio. Results are shown in Table VI. The tasks include queries such as "the total number of packets from a certain source IP prefix" and "the number of packets per second to a particular destination IP address". We use the five-tuple of the packet as the ID, and consider packet size and packet interval as the two statistical attributes.

Notably, we need to construct two separate CocoSketch and USS for the two statistical attributes, respectively. In contrast, we use Hyper-USS only a single sketch to record both attributes simultaneously. In subset ratio queries, CocoSketch and USS struggle to compute packet rates that need two attributes due to the difficulty of ensuring that the same item can

TABLE VI: Subset Query and Results for Network Measurement Case Study

Query Task	Ground Truth	Query Results			Query Type
		Ours	Coco	USS	
Total number of packets from source IP prefix 116.247.99.244/28	1,324,081	1,324,915	1,325,581	1,253,375	Subset Sum
Total number of packets sent to destination IP prefix 212.30.198.222/23	12,865,452	12,902,686	13,140,834	13,721,793	Subset Sum
The number of packets per second (PPS) from source IP address 116.247.99.244	1,103.42	1,196.46	–	–	Subset Ratio
The number of packets per second (PPS) sent to destination IP address 212.30.198.222	28,159.22	28,427.3	–	–	Subset Ratio

be simultaneously matched across separate sketches, as they replace items based on different attribute values, respectively. More specifically, when acquiring total packet size records in one sketch, the corresponding total packet interval in the other may fail to be retrieved. Our approach effectively addresses this limitation. Consequently, in practical query scenarios, our algorithm outperforms SOTA methods.

VII. Conclusion

A wide range of practical problems can be abstracted as subset queries over multiple attributes. Existing sketches are designed for single-attribute queries, making them inefficient for multi-attribute queries. We propose Hyper-USS, a novel sketching solution that provides the subset query over data streams involving multiple statistical attributes accurately and efficiently. With Joint Variance Optimization, Hyper-USS provides unbiased estimation and optimizes estimation variance jointly, addressing the challenge of accurately estimating multiple attributes in the sketch design. Further, our evaluation of Hyper-USS and its three speed-optimized versions shows that all algorithms surpass the SOTA in both query accuracy and insertion speed, highlighting their advanced capabilities and broad potential for future applications.

References

[1] 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.

[2] X. Gou, Y. Zhang, Z. Hu, L. He, K. Wang, X. Liu, T. Yang, Y. Wang, and B. Cui, "A sketch framework for approximate data stream processing in sliding windows," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 5, pp. 4411–4424, 2022.

[3] D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," in *SIGMOD 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018.

[4] 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.

[5] Punter, Wieger R and Papapetrou, Odysseas and Garofalakis, Minos, "Omnisketch: Efficient multi-dimensional high-velocity stream analytics with arbitrary predicates," *Proceedings of the VLDB Endowment*, vol. 17, no. 3, pp. 319–331, 2023.

[6] W. R. Punter, O. Papapetrou, and M. Garofalakis, "Omnisketch: Streaming data analytics with arbitrary predicates," *ACM SIGMOD Record*, vol. 54, no. 1, pp. 28–35, 2025.

[7] A. Manousis, "Enabling efficient and general subpopulation analytics in multidimensional data streams," *PVLDB*, 2022.

[8] H. Li, L. Wang, Q. Chen, J. Ji, Y. Wu, Y. Zhao, T. Yang, and A. Akella, "Chainedfilter: Combining membership filters by chain rule," *Proceedings of the ACM on Management of Data*, vol. 1, no. 4, pp. 1–27, 2023.

[9] Z. Wei, Y. Tian, W. Chen, L. Gu, and X. Zhang, "Dune: Improving accuracy for sketch-int network measurement systems," in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 2023, pp. 1–10.

[10] 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)*. IEEE, 2021, pp. 804–815.

[11] Q. Shi, C. Jia, W. Li, Z. Liu, T. Yang, J. Ji, G. Xie, W. Zhang, and M. Yu, "Bitmatcher: Bit-level counter adjustment for sketches," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, 2024, pp. 4815–4827.

[12] Q. Xiao, X. Cai, Y. Qin, Z. Tang, S. Chen, and Y. Liu, "Universal and accurate sketch for estimating heavy hitters and moments in data streams," *IEEE/ACM Transactions on Networking*, vol. 31, no. 5, pp. 1919–1934, 2023.

[13] J. Xu, Z. Bao, and H. Lu, "A framework to support continuous range queries over multi-attribute trajectories," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 2, pp. 1119–1133, 2023.

[14] P. O. Queiroz-Sousa and A. C. Salgado, "A review on olap technologies applied to information networks," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 14, no. 1, 2019.

[15] R. Ben-Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard, "Constant time updates in hierarchical heavy hitters," in *SIGCOMM 2017*. ACM, 2017.

[16] M. Cukier, R. Berthier, S. Panjwani, and S. Tan, "A statistical analysis of attack data to separate attacks," in *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE, 2006, pp. 383–392.

[17] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, 2005.

[18] M. Charikar, K. C. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theor. Comput. Sci.*, 2004.

[19] A. Metwally, D. Agrawal, and A. E. Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *ICDT 2005*, ser. Lecture Notes in Computer Science, T. Eiter and L. Libkin, Eds. Springer, 2005.

[20] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: adaptive and fast network-wide measurements," in *SIGCOMM 2018*. ACM, 2018.

[21] Y. Zhao, W. Zhou, W. Han, Z. Zhong, Y. Zhang, X. Zheng, T. Yang, and B. Cui, "Achieving top-k-fairness for finding global top-k frequent items," *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[22] 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.

[23] 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*, 2016, pp. 1449–1463.

[24] D. Ting, "Count-min: Optimal estimation and tight error bounds using empirical error distributions," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 2319–2328.

[25] 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)*. IEEE, 2021, pp. 864–875.

[26] C.-Y. Hsu, P. Indyk, D. Katabi, and A. Vakilian, "Learning-based frequency estimation algorithms." in *International Conference on Learning Representations*, 2019.

[27] J. Zhou and K. A. Ross, "Implementing database operations using simd instructions," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 2002, pp. 145–156.

[28] "Source code related to Hyper-USS." https://github.com/moxiaoshao/Fast_Hyper-USS.

[29] J. Li, Z. Li, Y. Xu, S. Jiang, T. Yang, B. Cui, Y. Dai, and G. Zhang, "Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams," in *KDD '20*. ACM, 2020, pp. 1574–1584.

[30] B. Zhao, X. Li, B. Tian, Z. Mei, and W. 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*, 2021, pp. 2285–2293.

[31] P. Pandey, S. Singh, M. A. Bender, J. W. Berry, M. Farach-Colton, R. Johnson, T. M. Kroeger, and C. A. Phillips, "Timely reporting of heavy hitters using external memory," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1431–1446.

[32] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong, "Finding persistent items in data streams," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 289–300, 2016.

[33] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: methods, evaluation, and applications," in *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, 2003.

[34] Z. Wei, G. Luo, K. Yi, X. Du, and J.-R. Wen, "Persistent data sketching," in *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, 2015, pp. 795–810.

[35] B. Shi, Z. Zhao, Y. Peng, F. Li, and J. M. Phillips, "At-the-time and back-in-time persistent sketches," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1623–1636.

[36] N. G. Duffield, C. Lund, and M. Thorup, "Priority sampling for estimation of arbitrary subset sums," *J. ACM*, 2007.

[37] M. Sharifzadeh and C. Shahabi, "The spatial skyline queries," in *Proceedings of the 32nd international conference on Very large data bases*, 2006, pp. 751–762.

[38] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An optimal and progressive algorithm for skyline queries," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 467–478.

[39] M. Tang, Y. Yu, W. G. Aref, Q. M. Malluhi, and M. Ouzzani, "Efficient parallel skyline query processing for high-dimensional data," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 10, pp. 1838–1851, 2018.

[40] G. Atluri, A. Karpatne, and V. Kumar, "Spatio-temporal data mining: A survey of problems and methods," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–41, 2018.

[41] M. Gyssens and L. V. Lakshmanan, "A foundation for multi-dimensional databases," in *VLDB*, vol. 97. Citeseer, 1997, pp. 106–115.

[42] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 985–1000.

[43] J. Lukić, M. Radenković, M. Despotović-Zrakić, A. Labus, and Z. Bogdanović, "A hybrid approach to building a multi-dimensional business intelligence system for electricity grid operators," *Utilities Policy*, vol. 41, pp. 95–106, 2016.

[44] J. Wang, S. Wu, H. Gao, J. Li, and B. C. Ooi, "Indexing multi-dimensional data in a cloud system," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 591–602.

[45] X. Song, M. Wu, C. Jermaine, and S. Ranka, "Statistical change detection for multi-dimensional data," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2007, pp. 667–676.

[46] D. M. Powers, "Applications and explanations of zipf's law," in *New methods in language processing and computational natural language learning*, 1998.

[47] X. Gou, L. He, Y. Zhang, K. Wang, X. Liu, T. Yang, Y. Wang, and B. Cui, "Sliding sketches: A framework using time zones for data stream processing in sliding windows," in *KDD '20*. ACM, 2020.

[48] "The Criteo 1TB Click Logs dataset." https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/.

[49] "Anonymized internet traces 2018," https://catalog.caida.org/details/dataset/passive_2018_pcap, accessed: 2022-6-29.

[50] K. Mirylenka, G. Cormode, T. Palpanas, and D. Srivastava, "Conditional heavy hitters: detecting interesting correlations in data streams," *The VLDB Journal*, vol. 24, pp. 395–414, 2015.

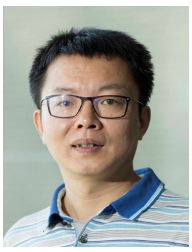
Ce Zheng is currently pursuing a Ph.D. at Beihang University. She graduated from the University of Chinese Academy of Sciences with a master's degree. Her research interests include sketching algorithms, network measurements, software-defined networking, and large language models (LLMs).

Zhouran Shi is an undergraduate student of Peking University majoring in Mathmatics. His research interests include large language models, sketches, and machine learning.

Ruijie Miao is currently pursuing the Ph.D. degree with the School of Computer Science, Peking University, under the supervision of Tong Yang. His research interests include network measurement and streaming algorithms.

Wenpu Liu is an undergraduate student in the School of Cyberspace Security at Southeast University. His current major research includes sketches and network measurement.

Tong Yang (Member, IEEE) is an associate professor in the School of Computer Science, Peking University. His research interests focus on data structures in networking, LLM, and data bases. He published 24 papers in SIGCOMM, SIGKDD, SIGMOD, and published 22 Trans on papers. He also served as the chair of international conferences and the editor of journals for several times. Many of his researches have been deployed in industry field, including Redis databases and ByteDance, *etc.*



Bin Cui (Fellow, IEEE) is a professor and Vice Dean in School of Computer Science at Peking University. He obtained his Ph.D. from National University of Singapore in 2004. His research interests include database system architectures, query and index techniques, big data management and mining. He is serving as Vice Chair of Technical Committee on Database China Computer Federation (CCF) and Trustee Board Member of VLDB Endowment. He was awarded Microsoft Young Professorship award (MSRA 2008), CCF Young Scientist award (2009), and Second Prize of Natural Science Award of MOE China (2014), *etc.*

Steve Uhlig obtained a Ph.D. degree in Applied Sciences from the University of Louvain, Belgium, in 2004. From 2004 to 2006, he was a Post-Doctoral Fellow of the Belgian National Fund for Scientific Research (F.N.R.S.). From 2004 to 2006, he was a visiting scientist at Intel Research Cambridge, UK, and at the Applied Mathematics Department of University of Adelaide, Australia. From 2006 to 2008, he was with Delft University of Technology, the Netherlands. Prior to joining Queen Mary, he was a Senior Research Scientist with Technische Universität Berlin/Deutsche Telekom Laboratories, Berlin, Germany. Since January 2012, he was the Professor of Networks and Head of the Networks Research group at Queen Mary University of London. From 2012 to 2016, he was a guest professor at the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. He's currently the Head of School of Electronic Engineering and Computer Science, QMUL.