Hermetis: Real-time Per-flow Distribution Measurement Using Sliding Window Mechanism

***Abstract*—Probabilistic data structures have a wide range of applications in data analysis, such as cardinality estimation, entropy calculation, and heavy hitter detection. However, these methods ignore the integration of a timely aging data cleaning mechanism, hindering their ability to analyze data based on the latest time window and thereby failing to capture the real-time per-key distribution effectively. To address this limitation, we propose Hermetis, a real-time per-key distribution measurement solution based on a sliding window mechanism. The core idea of Hermetis is to separate frequent and infrequent items by structuring them into two components based on their different arrival rates: a lean part for frequent items and a coarse part for infrequent items. We introduce a hybrid clearing strategy under a global clock, which clears the aged data efficiently through a controlled memory expansion. Additionally, a finite-state transition and inversion-based counter memory allocation and reclamation strategy is designed for the coarse part to ensure memory resilience and compactness under sliding window conditions. Hermetis also includes two key optimizations: a centralized refreshing strategy for improved throughput and an asynchronous timeline-based time-tag generation strategy to reduce query errors in the coarse part. Extensive experiments show that Hermetis significantly outperforms the state-of-the-art baseline methods, yielding an average throughput improvement of 2.86× and an average error reduction of over 10% in accuracy when the memory usage is 2MB. The theoretical analysis further supports its advantages with complexity bounds and error guarantees. All related codes of Hermetis are open-source and available at GitHub.**

***Index Terms*—Data stream processing, Approximate distribution estimation, Sketch**

I. Introduction

Recent years have seen probabilistic data structure-based stream processing techniques playing a significant role in network measurement. Using network measurements to map network traffic conditions, such as traffic behavior classification [1]–[5], performance diagnosis [6]–[11], and anomaly detection [12]–[17], *etc*. Leveraging these measurements for threat situation detection is a key development direction for detecting threats [18].

Among various types of traffic characterization, measuring ow key distribution can provide finer-grained information, such as packet size and delay. This detailed flow-level data holds great potential for improving anomaly detection, link load adjustment, and user quality of service. The core task of measuring the distribution of each flow in real-time in a sliding window is to continuously track the dynamic distribution characteristics (such as frequency, and quantile) of each flow key (such as IP address, and port) in the latest time window.

Case 1: Network latency monitoring. Website administrators must continuously monitor and assess the access latency experienced by different users, with particular emphasis on tail latency, as high latencies can significantly degrade the user experience. By measuring the request latency of each user (identified by their stream key) across various time windows, the system can facilitate the identification of users and request patterns associated with higher latencies.

Case 2: Internet anomaly detection. To ensure data timeliness, the system first filters out historical noise from infrequent anomalous flows. Real-time monitoring of the data distribution characteristics of each flow—such as the 90th percentile and entropy—enables the detection of network attacks or unauthorized access points, thereby enhancing network security.

Although there has been considerable works devoted to achieving accurate traffic distribution queries, however, none of the existing flow processing methods [19]–[23] consider the aging mechanism of data distribution. When the flow characteristics change rapidly, the flow key distribution may deviate significantly [24], [25]. Therefore, to address the need for real-time measurement of flow distributions, improving the data aging process for outdated data to more accurately reflect immediate distribution changes is crucial to maintain measurement accuracy in dynamic data environments.

In this paper, we propose a new scheme named Hermetis, which can accurately estimate per-key distributions in real time with high throughput. The core idea of Hermetis is to separate frequent and infrequent flow keys, enabling optimized data structure designs based on their different arrival rates. Specifically, Hermetis uses two distinct parts: the lean part for frequent flow keys and the coarse part for infrequent ones.

In order to realize a timely and low memory-consumption expired data clearing mechanism, Hermetis reconsiders the balance between measurement resources and measurement accuracy, revisits the accuracy and storage efficiency of the memory expansion mechanism [26], [27] and the window slicing mechanism [28], [29], then proposes a hybrid clearing strategy for aged data under global clock. That is, combining two methods: ”aging data check during project insertion” and ”time thread scanning check.” Furthermore, it integrates the data structures of the lean and rough components to design the algorithm effectively.

Aiming at reducing memory overhead, the design of the coarse part combines the counter size self-adjustment design with the counter resource recovery mechanism. When inserting data, the appropriate data insertion slot is allocated to the data item by combining slot adjustment and cuckoo slot eviction strategy. When the aging data of the coarse part is cleared, we propose a finite state transition mechanism to recycle the

corresponding storage slot of the data item in time.

We use a centralized refresh strategy to periodically traverse Hermetis globally, reducing the computing overhead caused by frequent sliding window scans. In addition, asynchronous timeline tags reduce data aging errors and enhance query accuracy by setting different timeline offsets for different arrays in the coarse part.

Our main contributions can be summarized as follows:

- We propose a new data structure, called Hermetis, which can automatically separate frequent and infrequent items, and design different data insertion and clearance algorithms for the lean and coarse parts.
- We introduce a hybrid clearing strategy to manage aged data using a global clock. This approach effectively addresses the differences in arrival frequency between frequent and infrequent items, supporting data aging threads while ensuring limited memory expansion.
- An adaptive counter update strategy is proposed for the coarse part of Hermetis, which can continuously provide sufficient counter memory size for newly inserted data streams as well as timely reclaim the occupied memory for aging data even under fluctuating network traffic distributions.
- We rigorously analyze the time and space complexity and accuracy of Hermetis and validate our analysis through extensive experiments on real network traffic traces. We demonstrate that Hermetis achieves higher throughput, accuracy, robustness, and memory utilization than state-of-the-art solutions. The source code is available at Github [30].

II. MOTIVATION

A. Problem Formulation

Data Stream. A data stream S is an infinitely ordered sequence of elements arriving over time, defined as:

$$S = \langle e_1, e_2, e_3, \ldots \rangle ,$$

where $e_i = (k_i, v_i, t_i)$ represents the i-th data item. Here, k_i is the stream key (a critical identifier such as an IP address or a 5-tuple), v_i is the value (e.g., packet size, latency), and t_i is the timestamp denoting the arrival time of the data item.

Time Window. A time window W is a fixed-length temporal interval defined as:

$$W = [t_{\text{current}} - N, t_{\text{current}}] ,$$

where N is the window size (e.g., the most recent N seconds or N items), and t_{current} is the current time. The sliding window mechanism retains only data within W; data outside this range are expired.

Aging Mechanism. The aging mechanism A removes stale data from the window W. Let $W_{\text{old}} = \{e_i \in W \mid t_i < t_{\text{new}} - N\}$, then:

$$A(W) = W \setminus W_{\text{old}}.$$

Per-Flow Histogram. For each data item $e_i = (k_i, v_i, t_i)$ in the stream, where v_i represents a measurable attribute value, the range $\mathcal{V}$ of v_i is partitioned into M mutually disjoint intervals $I_1, I_2, \ldots, I_M$. For any stream key k, its per-flow distribution within the time window W is represented by a histogram $H(k)$, where the j-th counter is defined as:

$$H(k)[j] = \sum_{\substack{e_i \in S_W, \\ k_i = k,\, v_i \in I_j}} 1, \quad j = 1, 2, \ldots, M.$$

Thus, the distribution $H(k)$ for stream key k is expressed as:

$$H(k) = \{H(k)[1], H(k)[2], \ldots, H(k)[M]\} .$$

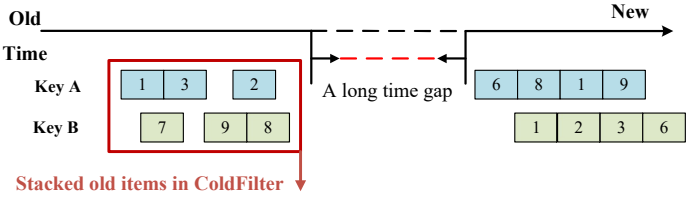


Fig. 1: Example of errors in data distribution measurements within a time window.

B. Background

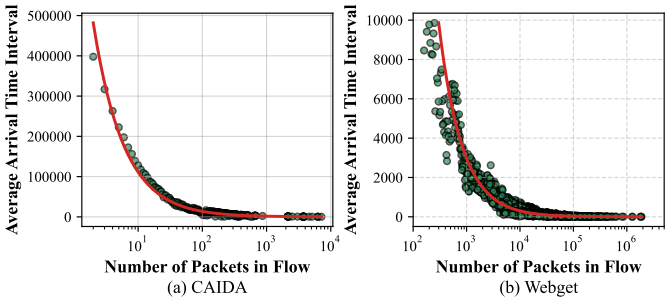
Considering that infrequent flows constitute the majority of data flows in the network, we cannot accurately recognize the distribution of flow key values within a limited time window due to their low frequency of occurrence. For instance, if a flow key appears only a few times over a long period and its tail quantile exceeds a threshold T, we might erroneously conclude that the flow key's distribution value consistently exceeds T, which is statistically implausible. In addition, existing work [22], [23] generally uses ColdFilter [31] and Bloom Filter [32] as a cold element filter, but these methods cannot adapt to the data timeliness problem of keys. This is because the data aging mechanism fails to reflect the latest data distribution in a timely manner, especially in an environment where the data distribution changes rapidly. For example, sketchPolymer uses coldFilter as a non-frequent item filter. We assume the calculation of the 50^{th} percentile and the setting of the threshold $T = 5$. As ColdFilter lacks a data aging mechanism for each stream (as shown in Fig. 1), it retains the early record of key **A**. Consequently, the recorded value sequence of key **A** is $(1, 3, 2, 6, 8, 1, 9)$. SketchPolymer will report that the 50^{th} percentile of key **A** is 3, which is less than the threshold T. However, the latest distribution of key **A** is $(6, 8, 1, 9)$, and the latest 50^{th} percentile is 6, which is greater than the threshold T. In contrast, for key **B**, ColdFilter tends to report that the 50^{th} percentile of key **B** is 6, which is greater than the threshold T, while the latest 50^{th} percentile of key **B** is 2, which is less than the threshold T. This demonstrates that ColdFilter struggles to process key value distribution changes in real time when using larger time window intervals. Conversely, smaller time window intervals fail to fully capture the key value distribution, reducing the accuracy of traffic distribution monitoring.

Existing schemes based on window scanning mechanism [26], [27], [33]–[35] can widely extend various types of sketches to sliding sketch scenarios [36]–[38]. However, these solutions have the following major problems.

1) The sketch scheme based on the scanning mechanism stores frequent items and infrequent items in each time window and shares the same counter. We conducted a statistical analysis on the number of flow packets (Flow Size) and the average arrival time interval (Average Arrival Time Interval) on the CAIDA and Webget datasets. As shown in Fig. 2, the number of flow packets and the average arrival time interval show a significant power function relationship, indicating that the arrival time of frequent items is highly concentrated, while the arrival interval of infrequent items is long and scattered. Existing work cannot reasonably utilize space, resulting in poor query performance.
2) The scanning-based sketch is based on the idea of maintaining a sequence by stream key [39], [40], which requires a large number of counter extensions. Although some schemes [34], [35] try to adaptively adjust the counter size, the storage overhead is still huge; The method based on time block partitioning [7], [41] divides the time window into several sub-windows. By using partition blocks to record information, the data structure can perform aging operations in the oldest block and update operations in the latest block. However, the time block-based partitioning method also has expensive memory overhead due to the maintenance of duplicate data structures at multiple time nodes. At the same time, this method of maintaining time windows in a coarse-grained manner through block partitioning can cause large errors for real-time distributed queries.
3) Since each stream key requires multiple counters to represent its distribution, the sketch scheme based on scanning mechanism involves frequent hash mapping operations and data aging operations, which seriously reduces the throughput performance. Memento [42] performs Window Update (deletes obsolete data) or performs Full update operation (deletes obsolete data and adds new data) with a certain probability. This probability sampling-based strategy can optimize throughput, but it may cause significant distribution estimation errors for traffic distribution estimation (especially infrequent flows).

The above analysis provides us with the following intuition when designing a memory-efficient per-flow real-time distribution measurement scheme:

1) By utilizing the arrival rules of frequent and infrequent items, designing data structures and timestamp tags with low storage overhead, we have an opportunity to implement an efficient and accurate aging data removal mechanism;
2) By coupling data structure design and timestamp tags of items, we can maintain the compactness of the data structure;
3) Under the premise of maintaining compact memory usage, it is necessary to improve the throughput of the solution by optimizing the time cost of aging data removal.



(a) CAIDA (b) Webget

Fig. 2: Relationship between flow size and average arrival time interval in CAIDA and Webget datasets.

III. Design of Hermetis

In this section, we design the data structure and algorithm of Hermetis and propose two optimization techniques to further improve the recognition accuracy and memory efficiency of Hermetis.

A. Overview of Hermetis

We propose Hermetis, a compact data structure for real-time measurement of per-key distribution. Our key idea is to adopt different strategies for frequent and infrequent traffic keys. Given the diversity of application requirements for real-time per-key distribution, we propose a fine-grained monitoring method that accurately measures and adapts to each stream's distribution in the current time window.

Data structure: As shown in Fig. 3, the data structure of Hermetis consists of two parts: a "lean part" that records frequent traffic and a "coarse part" that records infrequent traffic. The lean part is modeled as a hash table with l buckets. The flow key K in each bucket $\mathcal{B}$ records the flow key maintained in the bucket, and the negative counter NC records the flow count mapped to the bucket but the corresponding flow key is different from the flow key K in the bucket. The histogram array (H) contains m histogram intervals. For each histogram interval I, we split it into the histogram interval ID part bid, the start arrival time T_{first}, the last arrival time T_{last} and the counter part C. The hash function $h_k^{(lean)} : \{0,1\}^* \rightarrow [l], k \in [k_1]$ is associated with the lean part. The coarse part contains r arrays $A[0], A[1], \ldots, A[r-1]$, each array consisting of c slots. Each slot can hold several entries, where each entry contains a fingerprint fp, which is an 8-bit hash value of $\langle e, bid \rangle$ generated by the hash function h_{fp}, a time tag $time\_tag$, and a count value v that records the number of times the data item is inserted. The hash function $h_k^{(coarse)}$ is associated with the array $A[k], k = 0, 1, ..., r-1$. The detailed data structure of the lean part and the coarse part can be found in III.C and III.D.

Hermetis further proposes two optimization mechanisms to improve the throughput and query accuracy of Hermetis.

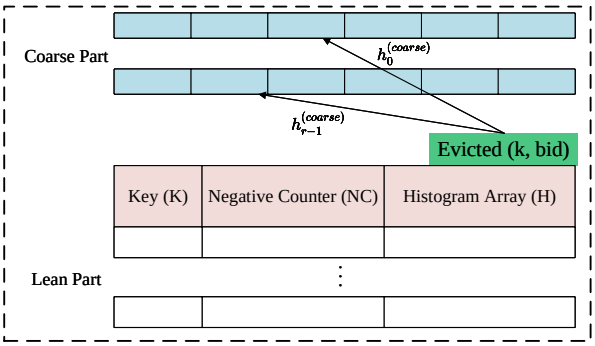


Fig. 3: Data structure of Hermetis.

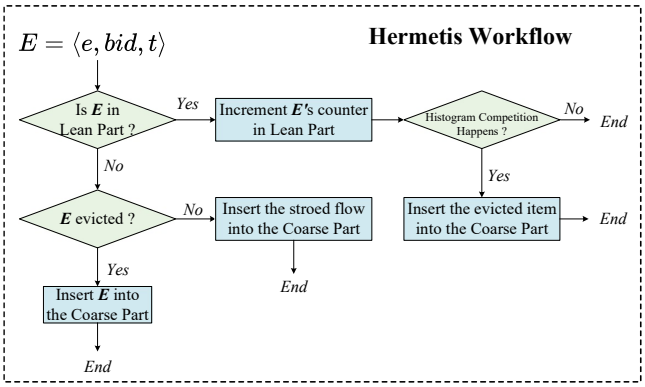


Fig. 4: Item insertion workflow in Hermetis.

- Centralized refresh strategy. We observed that most of the frequent items have a continuous arrival time. The mechanism of clearing expired data "by the way" when inserting data can already clean up the aging data well; while the infrequent stream arrival intervals are distributed in a few sub-time windows. Hermetis proposed to use a centralized refresh strategy instead of a sliding window scanning strategy, which can reduce the overhead of frequent scanning and updating, thereby improving processing efficiency.
- Asynchronous timeline-based time tag generation strategy. In order to reduce the data aging operation error caused by using time tags in the coarse part, our core idea is to use different timeline offsets for multiple arrays, and each array uses a completely independent hash map. When querying, we still use a query strategy similar to CM sketch.

B. Opeartions of Hermetis

Insertion Operation (Fig. 4): The workflow of Hermetis consists of two parts: the lean part insertion and the coarse part insertion. Given an item $E = \langle e, bid, t \rangle$, Hermetis first computes the insertion index $h_t^{(lean)}(e), t \in [k_1]$ of the item E in the lean part, and if there is a corresponding bucket $\mathcal{B}$ at the candidate locations maintains stream e or bucket $\mathcal{B}$ is empty, we insert the item $E = \langle e, bid, t \rangle$ into this bucket. If histogram competition occurs during insertion, we insert the evicted item $E' = \langle e, bid', v', t' \rangle$ into the coarse part. If the key e is not maintained in these buckets, the lean part will decide whether the item maintained in bucket $\mathcal{B}$ is evicted according to the stream key eviction strategy. If yes, item E will be inserted into the lean part and the stream e' maintained in a certain bucket $\mathcal{B}'$ will be evicted to the coarse part; otherwise item E will be evicted to the coarse part.

Querying Operation (Algorithm 1): For the item $E = \langle k, bid, t \rangle$ query, if k is in the lean part then we return the results of the lean part of the query (lines 2-7, *i.e.*, locate the buckets by hashing k, and take the sum of the d nearest counters to moment t). If k is not in the lean part then we query the results of the coarse part of the query (lines 9-23, *i.e.*, locate the r counters by hashing (k, bid) and take the smallest counter).

Algorithm 1 Querying operation in Hermetis.

Input: item e, interval ID bid, arriving time t
Output: $result$
1: $result = 0$
2: **if** e is in the lean part **then**
3: query the histogram interval I according to bid
4: $stp = \frac{t}{N/d} \bmod (d+1)$
5: **for** idx in $[0, d-1]$ **do**
6: $result+ = I.Counter\left[(stp - idx) \bmod (d+1)\right]$
7: **end for**
8: **else**
9: Calculate $fp = h_{fp}(e, bid)$, $time\_tag = \frac{t}{N/d} \bmod 4$
10: $coarse\_res = \infty$
11: **for** k in $[0, r-1]$ **do**
12: $idx = h_k^{(coarse)}(e, bid)$
13: candidate slots $Slot_1 = Sketch[k][idx]$, $Slot_2 = Sketch[k][idx \oplus h_k^{(coarse)}(fp)]$
14: **for** i in $[1, 2]$ **do**
15: **if** there is an j in $Slot_i$ *s.t.* $Slot_i[j].fp == fp$ **then**
16: $coarse\_res = min(coarse\_res, Slot_i[j].v)$
17: **end if**
18: **end for**
19: **end for**
20: **if** $coarse\_res \neq \infty$ **then**
21: $result+ = coarse\_res$
22: **end if**
23: **end if**
24: **return** $result$

Hybrid clearing strategy for aged data under global clock: Hermetis maintains a data aging thread under the global clock for the lean part and the coarse part respectively, and both adopt two data update modes: data aging thread scanning and clearing when inserting data items. The difference is that in order to balance accuracy and memory overhead, the lean part and the coarse part adopt different data structure designs for the maintenance of aging data. Since the lean part aims to maintain frequent items, in order to adapt to frequent data insertion and aging data clearing operations, the lean part uses more counter extensions and maintains two timestamps

to adapt to the frequent insertion and data aging operations of traffic. In the coarse part, considering the low arrival rate of infrequent items in the time window, the data aging mechanism of each item is maintained by a 2-bits time tag. When a data item e arrives at the coarse part at time t, the time tag of E is assigned $\left\lfloor \frac{dt}{N} \right\rfloor \bmod 4$ accordingly.

In addition to clearing expired data when inserting data, we also use the data aging thread to delete expired information. When the data aging thread of the lean part scans the bucket where the stream key is located, the lean part resets the "oldest" counter in each interval I; similarly, when the data aging thread of the coarse part scans a slot, the coarse part compares the time tag maintained in each entry in the slot with the time tag at the current moment, and resets the entry if the time tag maintained in the entry is expired.

Centralized refresh strategy: Further, we observed that due to the continuity of the time arrival of most frequent stream keys, the mechanism of clearing expired data "by the way" when inserting data can already clean up aged data well; while the arrival intervals of infrequent streams are distributed in a few sub-time windows. Therefore, we use a centralized refresh strategy instead of a sliding window scanning strategy [33] to filter aged data. The centralized refresh strategy refers to the data aging pointer of the lean part (coarse part, $resp.$), which traverses the buckets (slots, $resp.$) of the lean part (coarse part, $resp.$) once with a period of $\frac{N}{d}$. We compare the performance of these two strategies (corresponding to the HER-refreshing and HER-sliding models, respectively) in section V.F and V.G.

C. Lean Part

Data structure: As shown in Fig. 5, we split the lean part into a key part, a negative value counter part, and a histogram array part. The histogram array part consists of m histogram intervals. The ID part in each histogram interval I records the ID of the histogram interval to which the currently inserted data item belongs. The counter part C consists of $d+1$ sub-counters, denoted as $Counter[0], Counter[1], \ldots, Counter[d]$. The timestamps $I.T_{first}$ and $I.T_{last}$ record the earliest and latest arrival times of the items recorded in the histogram interval in the current time window. Since the hash table always maintains frequent items in the time window, based on this point, in addition to using an additional thread traversal to check data expiration like Clock-Sketch [43], the lean part also updates the histogram interval when inserting data items with the help of the global clock and the timestamps maintained in the histogram interval. In addition, the lean part maintains a data aging thread traversal pointer, which indicates the bucket index currently accessed by the data aging thread and clears the counter that is "farthest" from the current moment.

Initialization: We denote the hash table of the lean part as T. Before data insertion, we initialize each counter and all timestamps in the l buckets of the lean part to 0, and set the data aging thread to 0. We set the key value part in each interval to null. Define a mapping function $f_{ID}: \mathbb{N}^+ \to \mathbb{N}^+$, f_{ID} maps the stream value v to a discrete ID bid.

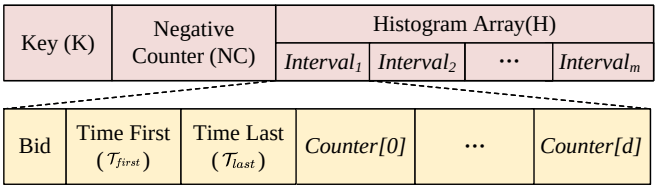


Fig. 5: Data structure of the lean part.

Algorithm 2 Inserting operation in lean part.

Input: interval I,item e, interval ID bid, arriving time t

$pos = \frac{t}{N/d} \bmod (d+1)$
if $(I.T_{first} == 0 \wedge I.T_{last} == 0) \vee t - I.T_{last} > N$ **then**
  for idx in $[0, d]$ **do**
    $I.Counter[idx] = 0$
  end for
  $I.Counter[pos] = v$
  $I.T_{first} = I.T_{last} = t$
else if $t - I.T_{first} > N + \frac{N}{d}$ **then**
  $clean\_step = \left\lfloor \frac{t - I.T_{first} - N}{N/d} \right\rfloor$
  for idx in $[1, clean\_step]$ **do**
    $I.Counter[(pos + idx) \bmod (d+1)] = 0$
  end for
  $I.Counter[pos] += v$
  $I.T_{first} = t - N$
  $I.T_{last} = t$
else
  $I.Counter[pos] += v$
  $I.T_{last} = t$
end if

Insertion: For each arriving item $E = \langle e, bid, t \rangle$, we first check whether e is contained in k_1 alternative candidate buckets $T\left[h_1^{(lean)}(e)\right], T\left[h_2^{(lean)}(e)\right], \ldots, T\left[h_{k_1}^{(lean)}(e)\right]$. Consider the following three cases.

Case 1: $e \in T$. At this time, there is a bucket $\mathcal{B}$ that stores the key e. We first traverse the histogram intervals in the histogram array part and check whether there is an interval that stores the stream value ID bid of e. If there is an interval I_0 in bucket B that maintains the stream value ID bid, we perform the interval update operation on I_0 as shown in Algorithm 2. Otherwise, if there is an empty interval I_1, set the stream value ID maintained by I_0 to bid, and set T_{first} and T_{last} to t. If there is no empty interval, randomly select an interval I' with the smallest counter value. $E' = \langle e, v' = \sum_{j=0}^{d} I'.C[j], t' = I'.Tlast \rangle$ is evicted to the coarse part and the interval I' is cleared. Then the item E is inserted into the interval I' .

Case 2: $e \notin T$, and one of the k_1 replacement candidate buckets is an empty bucket $T[h_i(e)]$. At this time, we save the key e to bucket $T[h_i(e)]$ and perform the interval update operation on interval $Interval_1$ as shown in Algorithm 2.

Case 3: $e \notin T$, and there is no empty bucket $T[h_i(e)]$ among the k_1 candidate replacement buckets. At this time, we

try to select the bucket $T[h_j(e)]$ with the smallest stream key cardinality among the k_1 mapping buckets for replacement. Let the stream count value in $T[h_j(e)]$ be V, and the negative counter count value be NC. We replace the stream key maintained in bucket $T[h_j(e)]$ with e with probability $b_0^{-(V-NC-v)}$ and evict the original stream key in $T[h_j(e)]$ to the coarse part. Otherwise, insert item E into the coarse part and update the negative counter count value to NC $+ v$.

Algorithm 3 Item insertion algorithm for coarse part.

Input: item e, interval ID bid, value v,arriving time t
Output: 0 or 1

Calculate $fp = h_{fp}(e, bid)$
for k in $[0, r-1]$ **do**
  $idx = h_k^{(coarse)}(e, bid)$
  $Slot_1 = A[k][idx]$, $Slot_2 = A[k][idx \oplus fp]$
  $time\_tag = (t + \frac{kN}{rd})/(N/d) \bmod 4$
  if there is an $i \in [1,2]$ and j such that $Slot_i[j].fp = fp$ **then**
    if $(time\_tag - Slot_i[j].time\_tag) \bmod (d+1) = d$ **then**
      $Slot_i[j].time\_tag = time\_tag$
    end if
    $Slot_i[j].v = v$
    if overflow happens **then**
      $flag = Entry\_adjust(Slot_i, j)$
      if $flag = 0$ **then**
        $State\_transition(Slot_i, j)$
      end if
    end if
  else if there is an empty and capable entry $Slot_i[j']$ **then**
    insert $(fp, v, time\_tag)$ into $Slot_i[j']$
  else
    randomly select $i' \in [1,2]$
    $Kick\_out(MAXLOOP, Slot_{i'}, 0)$
    insert $(fp, v, time\_tag)$ into $Slot_{i'}[0]$
  end if
end for
FUNCTION $Entry\_adjust(Slot, j)$:
  if there is an capable entry $Slot[j']$ for $Slot[j]$
    $Swap(Slot[j'], Slot[j])$; **return** 1
  end if
  return 0

D. Coarse Part

In the coarse part, we design a sketch to store infrequent items. The SOTA adaptive counter size solutions [44]–[46] include DHS and BitMatcher, *etc*. To solve the hash collision problem, BitMatcher employs a Cuckoo filter-like structure to balance bucket loads and ensure measurement accuracy in non-aging scenarios. However, when removing expired data items under the data window mechanism, due to the irreversible state transition of BitMatcher, the new data fills the old counter, which causes the counter size to not match the actual frequency, and frequently triggers Cuckoo hash to clean up the overflow data. The key reason why BitMatcher is inconsistent with the updated data stream distribution is that it does not consider the counter space recovery of expired data under the window mechanism. For the removal of expired data, the sliding window-based adaptive counter data structure should have corresponding state reversal rules to adapt to the new data distribution.

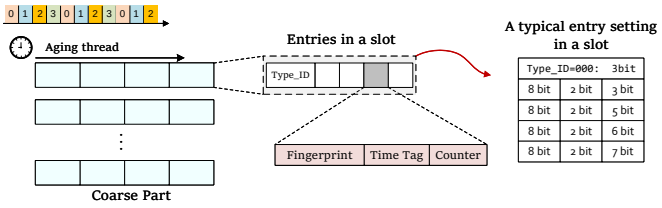


Fig. 6: Data structure of the coarse part.

We propose coarse part to automatically adjust the counter granularity according to the distribution of the inserted data stream. The main idea of the coarse part is to use finite state transitions to accurately adjust the counter length when new data items are inserted and expired data items are removed. Specifically, for each new entry, the coarse part first tries to record the data item in a candidate bucket, and the state of the bucket is dynamically adjusted according to the frequency of the data item. If an eviction operation occurs, we try to move the eviction item to another candidate bucket. At the same time, the coarse part executes a centralized refresh data aging strategy at the end of each sub-window. When the aged data in the entry is cleared, the coarse part executes the state inverse transition rule to recycle the counter resources in the bucket to adapt to the distribution of the new data item.

Data Structure: As shown in Fig. 6, the coarse part consists of a sketch of r rows and c columns and a data aging thread. Each slot in the sketch can accommodate several entries. Each entry consists of a triple $(fp, time\_tag, v)$, namely an 8-bit fingerprint fp about $\langle e, bid\rangle$ generated by the hash function h_{fp}, a time tag $time\_tag$, and an integer counter v that records the inserted value of the data item. The hash function $h_k^{(coarse)} : \{0,1\}^* \to [c], k \in [r]$ maps the item $\langle e, bid\rangle$ to the corresponding slot in the coarse part.

Different from the lean part, we use fingerprint instead of directly recording $\langle e, bid\rangle$ in the entry. Since at most 4 entries are recorded in each bucket in our design, the 8-bit fingerprint tag is sufficient to efficiently avoid hash collisions. In addition, the data aging thread clears the aged data according to the global time tag. As shown in Fig. 7.a, $\langle x_1, x_2, \ldots, x_B\rangle$ represents the state of the bucket, and the black box 3-bits binary values adjacent to them are the corresponding $TDPE_ID$s. When accessing a bucket, we first use the $TDPE\_ID$ to decode the number of entries and the counter size of each entry. For example, the initial state $\langle 3\ 5\ 6\ 7\rangle$ on the left means that there are 4 entries in the bucket at this time, and the flag is $\boxed{000}$, and the counter field occupies 3/5/6/7 bits respectively. It is verified that the fingerprint occupies 8 bits × 4 = 32 bits, the time tags occupies 2 bits × 4 = 8 bits, the counters occupies

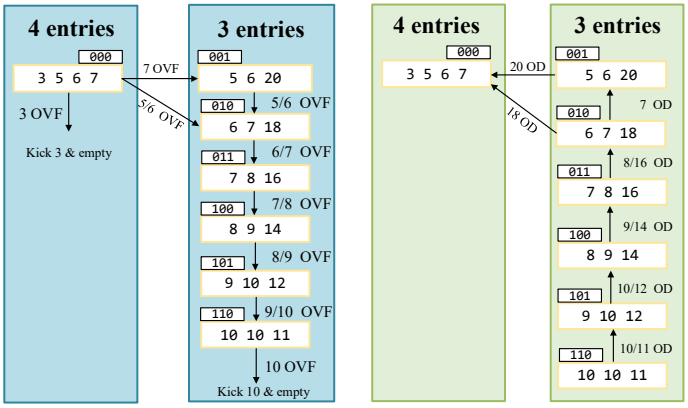
3+5+6+7 = 21 bits, and the $TDPE\_ID$ occupies 3 bits, with a total size of 64 bits. Similarly, if we find that the flag of a bucket is 001, we can decode the state in the bucket as $\langle 5\ 6\ 20 \rangle$ according to Fig. 7.a.

Item Insertion: The pseudo-code for the coarse part of the data item insertion algorithm is shown in Algorithm 3. For each data item $E = \langle e, bid, t, v \rangle$ inserted into the coarse part, for $0 \leq k \leq r-1$, Hermetis first computes the fingerprint fp of E and the two candidate slot indices $Slot_1, Slot_2$ in $A[k]$ according to the following formula.

$$fp = h_{fp}(e, bid), \tag{1}$$

$$Slot_1 = A[k][h_k^{(coarse)}(e, bid)], \tag{2}$$

$$Slot_2 = A[k][h_k^{(coarse)}(e, bid) \oplus fp]. \tag{3}$$



(a) State Transition table and rules. (b) State inversion table and rules.

Fig. 7: State transition and inversion in the coarse part.

Algorithm 4 State transition in the coarse part.

Input: $Slot$, entry number j
Output: 0 or 1
1: $ID = TDPE\_ID\ of\ Slot$
2: **if** $ID = 0$ **then**
3: $Kick\_out(MAXLOOP, Slot, 0$
4: **if** $j == 3$ **then**
5: transite $Slot$'s state to $TDPE\_ID\ =\ 1$
6: **else if** $j \neq 0$ **then**
7: transite $Slot$'s state to $TDPE\_ID\ =\ 2$
8: **end if**
9: **else if** $j \neq 2$ **then**
10: $Kick\_out(MAXLOOP, Slot, 0$
11: **if** $ID! = 6\ \wedge\ Slot\,[2]\ can\ be\ compressed$ **then**
12: transite $Slot$'s state to $TDPE\_ID = ID + 1$
13: **end if**
14: **end if**
15: FUNCTION $Kick\_out(maxloop, Slot, j)$:
16: **if** $maxloop < 0$ **then return** 0
17: Compute $Slot'$ be the alternate slot of $Slot[j]$
18: **if** $Slot'$ has a capable entry for $Slot[j]$ **then**
19: put $Slot[j]$ into this entry; **return** 1
20: **else** choose a smallest and capable j' from $Slot'$ **then**
21: $Kick\_out(--maxloop, Slot', j')$
22: put $Slot[j]$ into $Slot'[j']$; **return** 1

Then we retrieve all entries in the candidate slot. If e is in $Slot_1$ or $Slot_2$, we check if the $time\_tag$ in the corresponding entry is expired, if it is expired we set the counter value to 1 and the $time\_tag$ to the current $time\_tag$; otherwise, we add 1 to the counter of the corresponding entry. if item e is new, we insert it into the empty entry and set the counter value to 1 and the $time\_tag$ to the current $time\_tag$. If both slots are full, we randomly select $Slot_1[1]$ or $Slot_2[1]$, and subtract 1. After insertion, if an overflow occurs, we first try to put or swap it into a larger entry, as in line 12 of Algorithm 3. If this fails, the following in-slot state transitions are required. We do not discuss all the strategies about state transition in detail here due to space limitation. The pseudocode for state transition can be found in Algorithm 4, and the running examples of the state transition rules in the coarse part can be found in [45].

Algorithm 5 State inversion in the coarse part.

Input: $Slot$, entry number j
1: $ID = TDPE\_ID\ of\ Slot$
2: **if** $ID == 0\ \vee j == 0\ \vee (ID == 1\ \wedge\ j == 1\ )$ **then**
3: **return**
4: **else if** $ID == 2\ \wedge\ j == 2$ **then**
5: transite $Slot$'s state to $TDPE\_ID\ =\ 0$
6: **else**
7: transite $Slot$'s state to $TDPE\_ID\ =\ ID - 1$
8: **end if**
9: **return**

Asynchronous timeline-based time tag generation strategy: further, in order to reduce the query error caused by using time labels in the coarse part, the asynchronous timeline data labeling strategy uses different timeline offsets for r arrays. Specifically, we introduce a time offset $\Delta_i = \frac{iN}{rd}$ when calculating the time label for the i-th array sketch[i], $i = 0, 1, 2, ..., r-1$. When calculating the time label, we take $time\_tag_i = (t + \Delta_i)/(N/d) \bmod 4$ to replace the generation method of the synchronous timeline data label in the second line of Algorithm 3 to improve the accuracy of data cleaning. We compare the performance of these two strategies in section V.E.

State inversion: Similar to the state transition rule, when the aged data in the entry is cleared, we hope to execute as few decisions as possible to achieve bit-level counter resource recycling. The state inversion rule is shown in Fig. 7.b. For $4 \leq TDPE\_ID \leq 7$, if $entry_2$ and $entry_3$ in the slots are aged, the flag bit in the current bucket is reduced by 1; for $TDPE\_ID = 3$, if the largest in the slot $entry_3$ is aged, the flag bit in the current slot is set to 0; if in the slot $entry_2$ is aged, the flag bit in the current slot is rolled back to 1. For $TDPE\_ID = 1$, we only set the flag bit in the current slot

to 0 when $entry_3$ is aged. The detailed pseudo code of the coarse part state inversion is shown in Algorithm 5.

IV. MATHEMATICAL ANALYSIS

In this section, we build upon the theoretical frameworks presented in [20], [28], [33], [45], [47] to examine Hermetis. We begin by analyzing the lean and coarse part individually, and then combine them to derive the error bounds for Hermetis.

A. Analysis of Lean Part

Theorem 1. *The time complexity of the lean part is $O(k_1) + O(\frac{dlm}{N})$. The space complexity of the lean part is $O(lm(d+1))$, where l is the number of slots in the lean part, m is the number of intervals in a slot, N is the size of the sliding window and d is the number of sub-windows.*

Proof. Each item requires at most k_1 hash operations to locate its slot in the lean part, and whenever an item arrives or the clock increases, $\frac{(d+1)lm}{N}$ counters need to be scanned, the time cost of scanning buckets is $O(\frac{dlm}{N})$. Thus, the time complexity is $O(k_1) + O(\frac{dlm}{N})$. The lean part contains l slots. For each slot, it contains K, NC, and m intervals equipped with $d+1$ counters. Since the size of each counter is fixed, the space complexity is $O(lm(d+1))$. □

Theorem 2. *For each queried interval in a certain histogram, the average relative error of lean part ARE_L can exceed εN_L with a probability at most $\sum_{x=1}^{\Phi_L} \frac{1}{\varepsilon l f_x^2 (b-1)}$, where $N_L(\Phi_L)$ is the number(cardinality) of items of the sliding window mapped into the lean part, respectively. f_x is the frequency of x-th item, b_0 is the constant for probabilistic decay and ε is any small positive number.*

Proof. Suppose that lean part records a data stream S_L with N_L items, and E_L is the item set ($|E_L| = \Phi_L$). Let e_i be the i-th item in E_L, whose actual frequency is f_i. Then

$$\begin{aligned} P(ARE_L > \varepsilon N_L) &= P(\frac{1}{\Phi_L}\sum_{x=1}^{\Phi_L}\frac{||\widehat{f_x} - f_x||}{f_x} > \varepsilon N_L) \\ &= P(\sum_{x=1}^{\Phi_L}\frac{||\widehat{f_x} - f_x||}{f_x} > \varepsilon N_L \Phi_L) \\ &\le \sum_{x=1}^{\Phi_L} P(||\widehat{f_x} - f_x|| > \varepsilon N_L \Phi_L f_x). \end{aligned} \tag{4}$$

As is shown in [33], [47], we have

$$P(||\widehat{f_x} - f_x|| > \varepsilon f_x \Phi_L N_L) \le (\frac{k_1}{\varepsilon l \Phi_L f_x^2 (b_0 - 1)})^{k_1}, \tag{5}$$

then combine (4) and (5), we get

$$P(ARE_L > \varepsilon N_L) \le \sum_{x=1}^{\Phi_L} (\frac{k_1}{\varepsilon l \Phi_L f_x^2 (b_0 - 1)})^{k_1}. \tag{6}$$

□

B. Analysis of Coarse Part

Theorem 3. *The time complexity and space complexity of the coarse part are $O(r)$ and $O(rc)$, respectively. r and c are the numbers of rows and columns in the coarse part, respectively.*

Proof. Each item requires r hash operations in the coarse part, respectively. Thus, the time complexity of the coarse part is $O(r)$. The coarse part contains c counters. Thus, the space complexity of the coarse part is $O(rc)$. □

Theorem 4. *For each queried interval in a certain histogram, the average relative error of coarse part ARE_C can exceed εN_C with a probability at most $\frac{1}{\varepsilon} O(max\left\{\frac{r}{c}, \frac{\alpha}{rN_C}\right\})$, where $N_C(\Phi_C)$ is the number of (different) items of the sliding window mapped into the coarse part, α is the mathematical expected frequency of the batches(as defined in [28]) in the coarse part and ε is any small positive number.*

Proof. Suppose that coarse part records a data stream S_C with N_C items, and E_C is the item set ($|E_C| = \Phi_C$). Let e_i be the i-th item in E_C, whose actual frequency is f_i. Assume that there are B entries in each bucket of coarse part on average. The final frequency estimation of item e_i is

$$\widehat{f_i} = f_i - X_i - S_i + Y_i + B_i, \tag{7}$$

where X_i is the decrement brought by the replacement strategy (when an item cannot find a vacancy in the bucket). S_i is the decrement brought by the sliding scan strategy. Y_i is the increment due to fingerprint collisions (which only leads to overestimation). And B_i is the increment due to the error of coarse part does not clean a batch. As is shown in [45], $E(X_i) \le min\left\{f_i, \frac{N_C}{rc} \times \frac{1}{B+1}\right\}$,and $E(Y_i) \approx \frac{rN_C}{2^{\mathcal{F}+1}c}$, where $\mathcal{F}$ is the length of the fingerprint. Then we will calculate S_i,B_i separately.

For $E(S_i)$, we note that $E(S_i) = E_{S_i > \frac{re}{c}N_C}(S_i) + E_{S_i \le \frac{re}{c}N_C}(S_i) \le P(S_i > \frac{re}{c}N_C)f_i + P(S_i \le \frac{re}{c}N_C) \cdot \frac{re}{c}N_C$. In the coarse part, items are removed after d rounds of the aging process. To ensure that no item is prematurely deleted before its designated time window ends in a sliding window of size N, the aging frequency is set to $\frac{N}{d}$. As a result, the time interval between the insertion of an item and its removal extends to $\frac{dN}{(d-1)}$. This means that the coarse part effectively retains items for the entire duration of $\frac{dN}{(d-1)}$ from their insertion. As is shown in [33], we have $P(S_i > \frac{re}{c}N_C) < e^{-(r-1)}$. So $E(S_i) \le e^{-(r-1)}f_i + \frac{re}{c}N_C \sim O(\frac{r}{c}N_C)$.

For $E(B_i)$, since in the coarse part, B_i is caused by the batch miss matching. Consider the case that when an item of e_i arrives at time t_0, and the last coming item of e_i arrived at time t_1,Assume $t_1 - t_0 > 2N$, meaning that the second occurrence of e_i is the start of a batch and the aged items can be cleaned. As is shown in [28], an old batch under asynchronous timeline technique maybe miss-aging with probability at most $\lambda \le \frac{(1-P_0)^r}{r}$, where $P_0 \sim e^{-O(\frac{N}{d})}$. Consider the recorded old batches in e_i's entry, the number of batches the coarse part not cleaned satisfies $BN_i + 1 \sim Geo(1-\lambda)$. Then the expected

BN_i is $E(BN_i) = \frac{\lambda}{1-\lambda}$. Then $E(B_i) = E(BN_i) \cdot \alpha \approx \frac{\alpha(1-P_0)^r}{r} < \frac{\alpha}{r}$.

For $\widehat{f}_i = f_i - X_i - S_i + Y_i + B_i$,we conclude that $\widehat{f}_i - f_i \approx \frac{rN_C}{2^{\mathcal{F}+1}c} + \frac{\alpha(1-P_0)^r}{r} - min\left\{f_i, \frac{N_C}{rc} \times \frac{1}{B+1}\right\} - O(\frac{r}{c}N_C) \sim O\left(max\left\{\frac{rN_C}{c}, \frac{\alpha}{r}\right\}\right)$. For $P(\frac{||\widehat{f}_i - f_i||}{f_i} > \varepsilon N_C) = P(||\widehat{f}_i - f_i|| > \varepsilon f_i N_C) \leq P(||\widehat{f}_i - f_i|| > \varepsilon N_C)$. By Markov's inequality, $P(\frac{||\widehat{f}_i - f_i||}{f_i} > \varepsilon N_C) \leq \frac{E(||\widehat{f}_i - f_i||)}{\varepsilon N_C} \leq \frac{1}{\varepsilon} O\left(max\left\{\frac{r}{c}, \frac{\alpha}{rN_C}\right\}\right)$. □

Based on our Theorem 2 and Theorem 4, Theorem 5 summarize the error bound of the average relative error of Hermetis.

Theorem 5. *For each queried interval in a certain histogram, the average relative error of Hermetis ARE can exceed εN with a probability at most $\sum_{x=1}^{\Phi_L} \left(\frac{\theta k_1}{\varepsilon(\theta+1) l \Phi_L f_x^2 (b_0-1)}\right)^{k_1} + \frac{1}{\varepsilon(\theta+1)} O\left(max\left\{\frac{r}{c}, \frac{\alpha}{rN_C}\right\}\right)$, where $\theta = \frac{N_L}{N_C}$.*

Proof. From our definition,

$$\begin{aligned} P\left(ARE > \varepsilon N\right) &= P\left(\frac{1}{\Phi}\sum_{x=1}^{\Phi}\frac{\left|\left|\widehat{f}_x - f_x\right|\right|}{f_x} > \varepsilon N\right) \\ &= P\left(\sum_{x=1}^{\Phi}\frac{\left|\left|\widehat{f}_x - f_x\right|\right|}{f_x} > \varepsilon N\Phi\right). \end{aligned} \tag{8}$$

Note that we are trying to have the lean part record frequent flow as well as have the coarse part record infrequent flow, then

$$\begin{aligned} &P\left(ARE > \varepsilon N\right) \\ &= P\left(\sum_{x=1}^{\Phi}\frac{\left|\left|\widehat{f}_x - f_x\right|\right|}{f_x} > \varepsilon N\Phi\right) \\ &\approx P\left(\sum_{x=1}^{\Phi}\frac{\left|\left|\widehat{f}_x - f_x\right|\right|}{f_x} > \varepsilon N(\Phi_L + \Phi_C)\right) \\ &\leq P(\sum_{x\in E_L}\frac{||\widehat{f}_x - f_x||}{f_x} > \varepsilon N\Phi_L) \\ &+ P(\sum_{x\in E_C}\frac{||\widehat{f}_x - f_x||}{f_x} > \varepsilon N\Phi_C) \\ &= P(ARE_L > \frac{(\theta+1)}{\theta}\varepsilon N_L) + P(ARE_C > (\theta+1)\varepsilon N_C). \end{aligned} \tag{9}$$

Combine the conclusion of our Theorem 2 and Theorem 4, we get the result mentioned above. □

V. PERFORMANCE EVALUATION

In this section, we first describe the experimental setup and metrics, and then we conduct experiments to evaluate Hermetis. We summarize our findings as follows:

- Hermetis achieves high accuracy for various types of query tasks **(Exps** $\#1 - \#2$**)**.
- Hermetis is robust to different parameter configurations **(Exp** $\#3$**)**.
- Effectiveness of Hermetis optimization strategies **(Exps** $\#4 - \#5$**)**.
- Hermetis achieves high insertion throughput. **(Exp** $\#6$**)**.

TABLE I: Abbreviations of algorithms in experiments.

Abbreviation	Full name
WCSS	Window Compact Space-Saving [26]
ECM	Exponential Count-Min Sketch [39]
SWAMP	Sliding Window Approximate Measurement Protocol [50]
SWCM	Splitter Windowed Count-Min Sketch [51]
SL-CM(CO, HK)	Sliding Sketchs combined with CM(CO,HK) [33]
WCSS	Window Compact Space-Saving [26]
λ-Algorithm	λ-sampling Algorithm [52]
HER-sliding	Hermetis with the sliding strategy
HER-refreshing	Hermetis with the refreshing strategy

A. Experiment Setup and Metrics

Dataset: We use two real-world traces in our experiments. (1) CAIDA: The traces collected in the Equinix-Chicago monitor from CAIDA in 2018 [48]. We use the trace with a monitoring interval of 60s, which contains around 27M packets. (2) Web Latency Dataset: The Web Latency Dataset is collected by Webget [49] through 182 probes deployed worldwide. We take the latency of each request as a value and index it with the source IP address of the request as a keyword.

The experiments are conducted in count-based sliding windows, where each packet arrival represents a unit of time.

Implementation Platform: We run experiments on a machine with a 4-core CPU (13th Gen Intel(R) Core(TM) i5 13600KF). All algorithms are implemented in C++. The source code can be found at [30].

Metrics: We evaluate the performance metrics whenever the window slides $\frac{N}{5d}$. The average value is used to represent the experimental result for a given parameter setting.

- **Point Query ARE:** $\frac{1}{n_1}\sum_{i=1}^{n_1}\frac{||\widehat{f}_i - f_i||}{f_i}$, where n_1 is the number of relevant intervals in the current time window, f_i represents the real frequency, and $\widehat{f}_i$ represents the estimated frequency.
- **Point Query Hit Rate:** the ratio of intervals whose IREs are less than a threshold 10%.
- **Hist Query ARE:** $\frac{1}{n_2}\sum_{i=1}^{n_2} HRE_i$, where n_2 is the number of relevant keys in the current time window.
- **Hist Query Hit Rate:** the ratio of keys whose HREs are less than a threshold 1%.
- **Throughput:** We test the million items per second (Mips) of different algorithms to measure the throughput.

Baseline: We compare our algorithm with eight state-of-the-art sketch-based algorithms under the same memory usage: WCSS, ECM, SWAMP, SL-HK, SWCM, SL-CM, SL-CO, and λ-Algorithm. On this basis, we analyze the effects of the parameters in Hermetis and our optimization strategy on the system performance. The abbreviations of the algorithms in the experiment are shown in Table I.

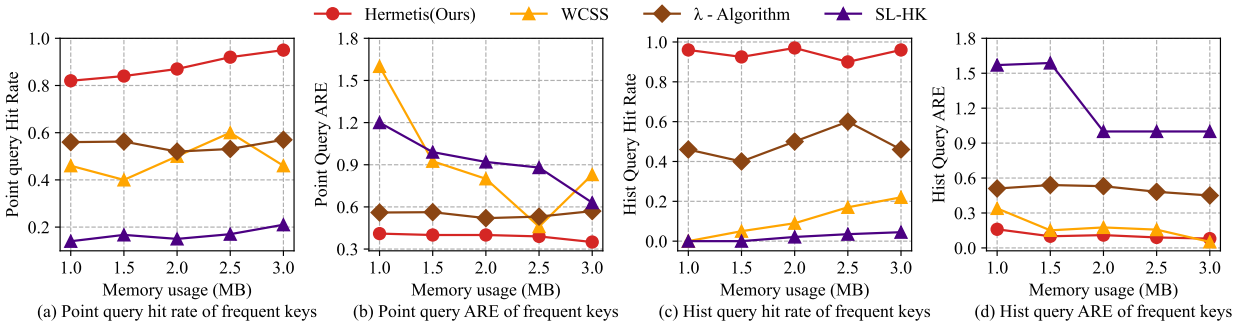


Fig. 8: Performance evaluation of frequent keys query accuracy in CAIDA dataset. The window size is 30000.

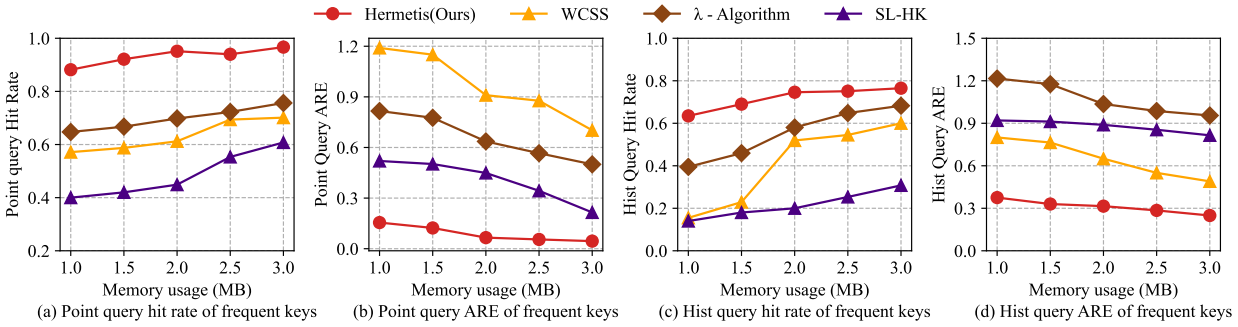


Fig. 9: Performance evaluation of frequent keys query accuracy in Webget dataset. The window size is 30000.

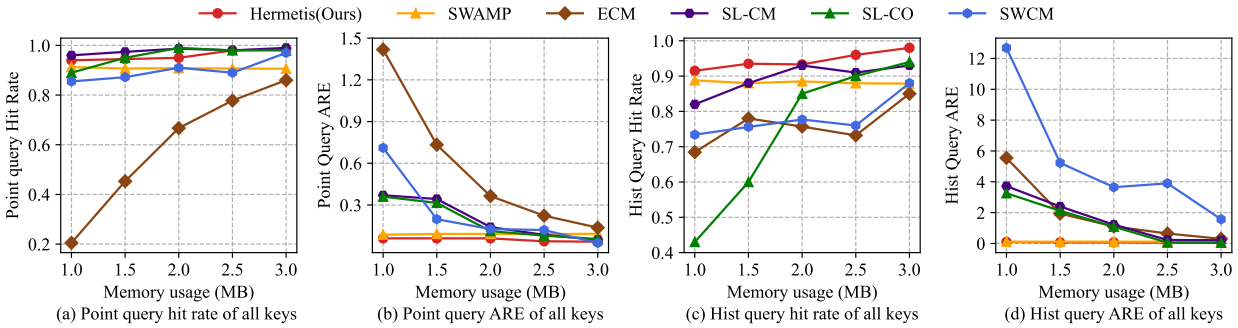


Fig. 10: Performance evaluation of all keys query accuracy in CAIDA dataset. The window size is 30000.

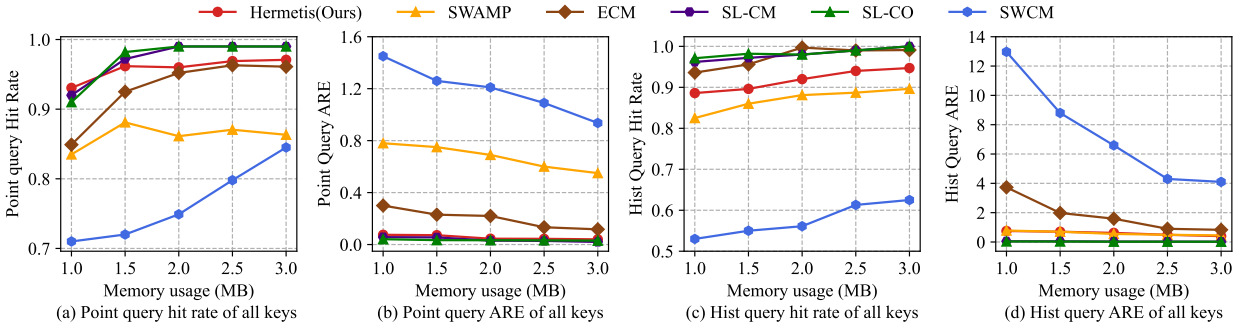


Fig. 11: Performance evaluation of all keys query accuracy in Webget dataset. The window size is 30000.

B. Evaluation of frequent keys query accuracy

(Exp#1) Performance evaluation of frequent keys query accuracy. We compared 3 methods: WCSS, λ-Algorithm and SI-HK. Our results show that Hermetis exhibits good accuracy and memory for frequent key queries robustness under configuration. As shown in Fig. 8a, 8c, the hit rate of point query and histogram query for frequent flow queries is significantly

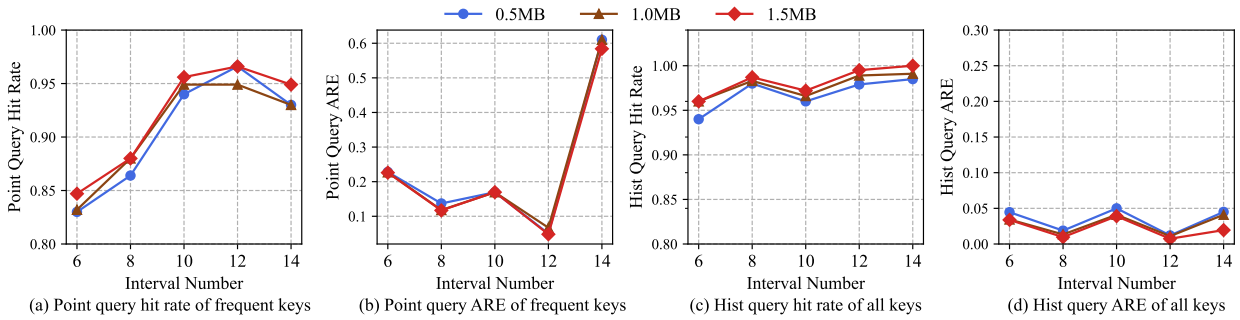


Fig. 12: Experiments with the frequent keys on parameter m and memory usage.

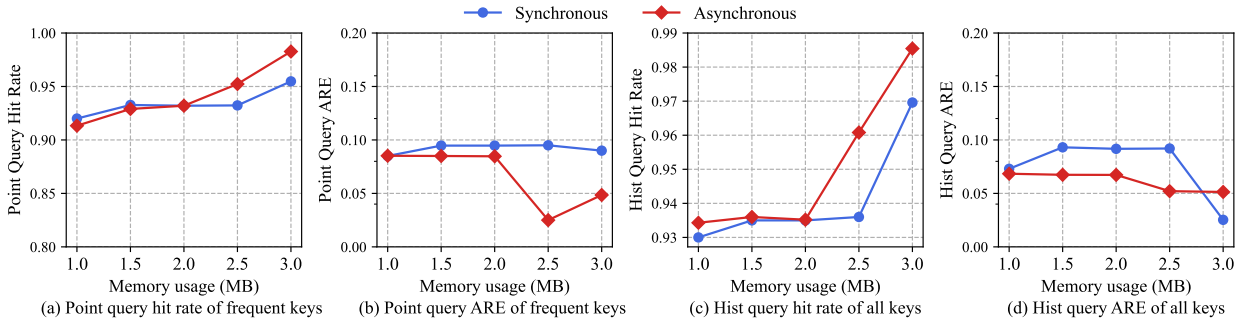


Fig. 13: Experiments with the all keys on Synchronous/Asynchronous strategies.

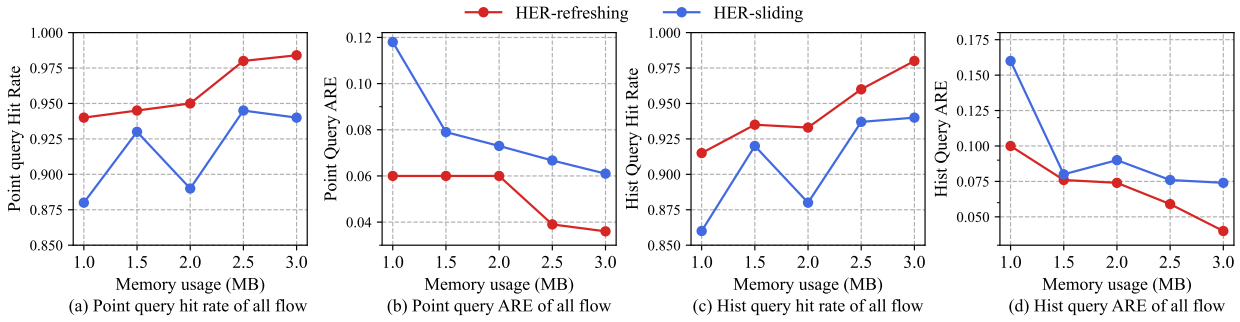


Fig. 14: Performance evaluation of the query accuracy on refreshing/sliding strategies.

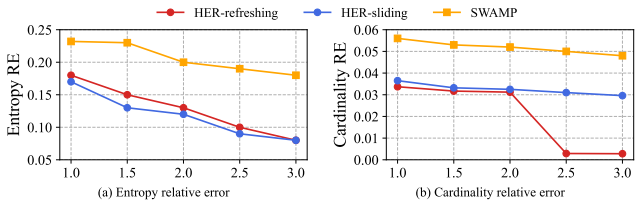


Fig. 15: Entropy and cardinality estimation on refreshing/sliding strategies.

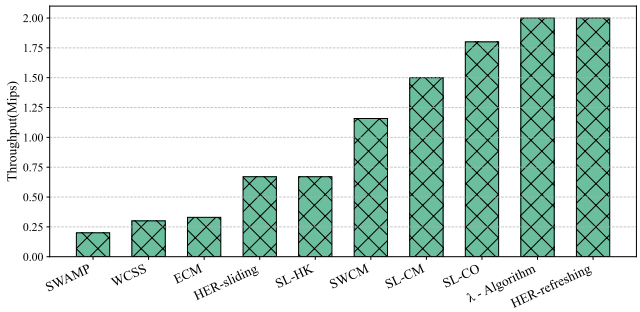


Fig. 16: Throughput of different algorithms.

higher than the three baseline schemes. With 2 MB of memory, Hermetis achieves a point query hit rate that is 1.74, 1.673, and 5.8 times higher than WCSS, λ-Algorithm and SI-HK, respectively. The AREs of WCSS and SL-HK point/histogram queries are greatly affected by memory usage. As shown in Fig. 8b, in point queries, the ARE of Hermetis is about 1, 0.075 and 1.3 lower than those of WCSS, λ-Algorithm and SI-HK respectively. At the same time, as shown in Fig. 8d, the ARE of Hermetis in histogram query is about 0.6, 3.818 and 13.44 times lower than WCSS, λ-Algorithm and SI-HK

respectively.

For the Webget dataset, as illustrated in Fig. 9a and 9c, the hit rate of point query and histogram query for frequent flow queries is notably higher than the three baseline schemes. Specifically, with 2 MB of memory, the hit rate of Hermetis point query is 1.5, 1.6, and 4.7 times higher than that of WCSS, λ-Algorithmm, and SI-HK, respectively. The hit rates of Hermetis histogram query are 9.5, 1.8, and 40.3 times higher than those of WCSS, λ-Algorithm, and SI-HK, respectively. The AREs of WCSS and SI-HK point/histogram queries are significantly influenced by memory usage. As shown in Fig. 9b, in point queries, the ARE of Hermetis is approximately 0.9, 0.07, and 1.1 lower than those of WCSS, λ-Algorithm, and SI-HK, respectively. Additionally, as depicted in Fig. 9d, the ARE of Hermetis in histogram query is about 0.5, 3.5, and 12.8 times lower than those of WCSS, λ-Algorithm, and SI-HK, respectively. These results highlight the robustness and efficiency of Hermetis in handling frequent key queries on the Webget dataset. , and the details are shown in our technical report

C. Evaluation of all keys query accuracy

(Exp#2) Performance evaluation of all keys query accuracy. We compared five baseline methods—SWAMP, ECM, SL-CM, SL-CO, and SWCM. Results indicate that Hermetis demonstrates high accuracy and robustness in all-keys queries across various memory configurations, significantly outperforming ECM and SWCM. Fig. 10a and 10c show that Hermetis consistently achieves a higher hit rate for all flow queries than the baselines. With 2 MB of memory, Hermetis's point query hit rate is 1.054, 1.232, 0.96, 0.959, and 1.04 times greater than that of SWAMP, ECM, SL-CM, SL-CO, and SWCM, respectively, while its histogram query hit rate is higher by 1.05, 1.424, 1.097, and 1.2 times. The relative error (ARE) of Hermetis's point and histogram queries remains below 0.1, substantially lower than other baselines except SWAMP, which, however, has significantly lower hit rates compared to Hermetis. SL-CM and SL-CO exhibit similar histogram query hit rates but show greater sensitivity to memory allocation in terms of ARE. As shown in Fig. 10b, Hermetis's point query ARE is approximately 0.53, 5.07, 1.35, 0.86, and 1.117 times lower than that of SWAMP, ECM, SL-CM, SL-CO, and SWCM, respectively, and in Fig. 10d, its histogram query ARE is 0.61 to 48.32 times lower across these baselines. For the Webget dataset, as is shown Fig. 11, Hermetis still performs significantly better than SWCM and SWAMP. ECM is only slightly better than Hermetis in terms of histogram query hit rate. Compared with SL-CM and SL-CO, Hermetis achieves the robustness and accuracy on different datasets at the cost of tolerable performance degradation on the Webget dataset.

D. Evaluation on Lean Part parameter m

(Exp#3) Experiments with the frequent keys on parameter m and memory usage. Fig. 12 shows the performance of the lean part for different memory usage (0.5MB, 1.0MB and 1.5MB) and m (number of intervals) settings. Fig. 12a shows that the point query hit rate tends to increase and then decrease with m, reaching a peak of about 0.97 at $m = 12$. Higher memory configurations help improve the hit rate. For the average relative error (Fig. 12b), as m increases, the error first decreases but fluctuates significantly at $m = 14$. Overall, Hermetis maintains a low error level in most cases and reaches its minimum at $m = 12$, indicating that Hermetis can significantly reduce point query error with appropriate parameters. Fig. 12c shows that Hermetis maintains a high hit rate close to 1.0 in histogram queries regardless of memory configuration or parameter m setting. Fig. 12d shows that its error range is consistently below 0.05, and memory increase slightly reduces the error, while parameter m has a small effect on the error, further confirming the robustness of Hermetis in histogram queries.

Combining our validation results, we recommend setting m between 10 and 12 to ensure a high point query hit rate, control the average relative error and avoid error fluctuations at $m = 14$.

E. Evaluation on Coarse Part Optimization

(Exp#4) Experiments with the all keys on synchronous and asynchronous strategies. Fig. 11 shows the performance of the coarse part with synchronous and asynchronous time tag generation strategies under different memory configurations.

For the point query accuracy, as shown in Fig. 13a, the asynchronous strategy outperforms the synchronous strategy in point query hit rate, especially in large memory configurations, the asynchronous strategy improves the hit rate significantly, depicting that the asynchronous timestamp generation strategy is more effective in improving the query accuracy in larger memory environments. For the average relative error (Fig. 13b), the asynchronous strategy has lower error in 1.5MB and above memory configurations, and especially in 2.0MB memory, its error is significantly better than that of the Synchronous strategy, which shows the advantage of the asynchronous strategy in optimizing the error in specific memory conditions. For the histogram query accuracy, as shown in Fig. 13c, the asynchronous strategy has a hit rate close to 1.0 under large memory configurations (especially 3.0MB), which is significantly better than the synchronous strategy, indicating that the asynchronous strategy significantly improves the query accuracy under resource-rich conditions. Fig. 13d shows that the ARE of histogram queries under different memory configurations for the asynchronous strategy is about half of that of the synchronous strategy, which exhibits stronger error control.The comprehensive evaluation results show that the asynchronous strategy outperforms the synchronous strategy in most of the metrics, especially in larger memory environments, where it exhibits significant advantages.

F. Evaluation on Sliding v.s. Refreshing strategies

(Exp#5) Experiments on refreshing/sliding strategies. Figures 14 and 15 illustrate the impact of different aging data clean strategies in Hermetis: the sliding window (HER-sliding) and centralized refreshing approach (HER-refreshing). Both

strategies outperform SWAMP in entropy and cardinality prediction accuracy. As shown in Fig. 14, HER-refreshing outperforms HER-sliding in hit rate and error control across point and histogram queries. For point queries, HER-refreshing's hit rate nears 1.0 under higher memory (e.g., 2.5MB) with low relative error (Fig. 14a, 14b). Similarly, for histogram queries, HER-refreshing maintains a hit rate close to 1.0 under high memory with consistently low error (Fig. 14c, 14d), further decreasing as memory grows. These results affirm that the refreshing strategy enhances query accuracy and error control across query types. As seen in Fig. 15a, HER-refreshing significantly reduces entropy prediction error with increased memory and approaches HER-sliding's performance, with both falling below 0.10 error when memory exceeds 2.5MB. In cardinality prediction (Fig. 15b), HER-refreshing achieves near-zero error at 2.0MB and above, outperforming both HER-sliding and SWAMP, demonstrating improved handling of aged data.

G. Evaluation on Throughput

(Exp#6) Experiments on throughput of different algorithms. This study evaluated the throughput of Hermetis and several baseline models. We measured the number of items processed per second (in millions of items per second, Mips) using 2 MB of memory and the CAIDA dataset. Fig. 16 shows the comparison of the throughput of different models under the same experimental conditions. HER-refreshing and λ-algorithm perform the best with a throughput close to 2.0 Mips, which is significantly better than the other models, this is mainly attributed to the fact that HER-refreshing employs an aging data removal strategy based on centralized refreshing, which greatly reduces the overhead of frequent scanning and updating and significantly improves the processing efficiency. In contrast, HER-sliding is limited by the resource consumption of sliding windows and the overhead of frequent data aging processing resulting in lower throughput. SL-CO and SL-CM also perform better in throughput, weaker than HER-refreshing and λ-algorithm, with about 1.5 Mips, but the window scanning required for their high-frequency data updating increases the computational overhead. In comparison, SWAMP, WCSS and ECM have poor throughputs, none of them exceeding 1.0 Mips, especially SWAMP, which has the lowest throughput due to the frequent replacement of the cyclic fingerprint cache (CFB). In summary, HER-refreshing outperforms the other models in terms of throughput, verifying the superiority of its memory management and processing efficiency, while the performance of the other models is closely related to the efficiency of their memory management and data aging strategies.

VI. CONCLUSION

This paper introduces the Hermetis solution designed to enable per-flow real-time distribution estimation under sliding window mechanism. The key techniques of Hermetis are a hybrid clearing strategy for aged data under global clock and a memory management strategy based on finite state tuning for aging data removal of frequent and infrequent items, respectively, as well as memory elasticity management in time-varying traffic environments. In addition, we further propose an asynchronous timeline-based time tag generation strategy and a centralized refreshing strategy to improve query accuracy as well as throughput. We build a prototype of Hermetis and evaluate its feasibility and effectiveness through comprehensive experiments using real traffic traces from the CAIDA dataset. The experimental results show that the Hermetis algorithm is fair and accurate for frequent and infrequent terms. It is also robust to different parameter configurations and exhibits high throughput efficiency.

In our future work, we aim to introduce advanced compression techniques to better handle high-traffic scenarios. Additionally, expanding the generality of Hermetis to support diverse traffic distributions and dynamically evolving network environments will be a key focus, ensuring its applicability across a broader range of use cases.

REFERENCES

[1] G. Zhou, Z. Liu, C. Fu, Q. Li and K. Xu, "An efficient design of intelligent network data plane", in *2023 USENIX Annual Technical Conference, USENIX Security 23*, 2023, pp. 6203-6220.

[2] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak and N. Zilberman, "IIsy: Hybrid in-network classification using programmable switches", *IEEE/ACM Transactions on Networking*, 2024.

[3] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos and A. Madeira, "FlowLens: Enabling efficient flow classification for ML-based network security applications", in *Network and Distributed Systems Security Symposium*, 2021.

[4] B. M. Xavier, R. S. Guimarães, G. Comarela and M. Martinello, "Programmable switches for in-networking classification", in *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021, pp. 1–10.

[5] Z. Guan, C. Liu, G. Xiong, Z. Li and G. Gou, "FlowTracker: Improved flow correlation attacks with denoising and contrastive learning", *Computers Security*, vol. 125, pp. 103018, Feb. 2023.

[6] Y. Lei, L. Yu, V. Liu and M. Xu, "Printqueue: performance diagnosis via queue measurement in the data plane", in *Proceedings of the ACM SIGCOMM 2022 Conference*. 2022, pp. 516-529.

[7] Y. Zhou, J. Bi, T. Yang, K. Gao, J. Cao, D. Zhang, et al., "Hypersight: Towards scalable high-coverage and dynamic network monitoring queries", *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 6, pp. 1147-1160, 2020.

[8] W. -X. Liu, J. Cai, S. Ling, J. -Y. Zhang and Q. Chen, "QALL: Distributed Queue-Behavior-Aware Load Balancing Using Programmable Data Planes," *IEEE Transactions on Network and Service Management*, vol. 21, no. 2, pp. 2303-2322, 2024.

[9] T. Zhang, R. Huang, J. Wei, S. Zhou, C. Ruan, K. Chen, J. Wang and G. Min, "Taming the Aggressiveness of Heterogeneous TCP Traffic in Data Center Networks," *IEEE/ACM Transactions on Networking*, vol. 32, no. 3, pp. 2253-2268, 2024.

[10] L. Li, K. Xu, D. Wang, C. Peng, K. Zheng, R. Mijumbi and Q. Xiao, "A Longitudinal Measurement Study of TCP Performance and Behavior in 3G/4G Networks Over High Speed Rails," *IEEE/ACM Transactions on Networking*, vol. 25, no. 4, pp. 2195-2208, 2017.

[11] Y. Zhao, G. Cheng and Y. Tang, "SINT: Toward a Blockchain-Based Secure In-Band Network Telemetry Architecture," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 2667-2682, 2023.

[12] C. Misa, R. Durairajan, A. Gupta, R. Rejaie and W. Willinger, "Leveraging prefix structure to detect volumetric ddos attack signatures with programmable switches", in *2024 IEEE Symposium on Security and Privacy*, 2024, pp. 4535-4553.

[13] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang et al., "LightGuardian: A full-visibility lightweight in-band telemetry system using sketchlets", in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 2021, pp. 991-1010.

[14] L. Rudman and B. Irwin, “Characterization and analysis of ntp amplification based ddos attacks,” in *2015 Information Security for South Africa (ISSA)*. IEEE, 2015, pp. 1–5.

[15] C.Rossow, “Amplification hell: Revisiting network protocols for ddos abuse.” in *Network and Distributed Systems Security Symposium ser. NDSS*, 2014, pp. 1–15.

[16] M. Kührer, T. Hupperich, C. Rossow and T. Holz, “Hell of a handshake: Abusing TCP for reflective amplification DDoS attacks,” in *8th USENIX Workshop on Offensive Technologies*, 2014.

[17] M. Kührer, T. Hupperich, C. Rossow and T. Holz, ”Exit from hell? Reducing the impact of amplification DDoS attacks”, in *23th USENIX Annual Technical Conference, USENIX Security 14*, 2014, pp. 111-125.

[18] Y. Zhou, G. Zhao, R. Alroobaea, A. M. Baqasah and R. Miglani, ”Research on data mining method of network security situation awareness based on cloud computing”, *Journal of Intelligent Systems*, vol. 31, no. 1, pp. 520-531, 2022.

[19] R. Shahout, R. Friedman and R. Ben-Basat, ”Together is better: Heavy hitters quantile estimation”, in *Proceedings of the ACM on Management of Data*, 2023, pp. 1-25.

[20] J. He, J. Zhu and Q. Huang, ”Histsketch: A compact data structure for accurate per-key distribution monitoring”, in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, IEEE, 2023.

[21] Y. Wu, A. Yuan, Z. Shi, Y. Li, Y. Zhao, P. Chen, T. Yang and B. Cui, ”Online Detection of Outstanding Quantiles with QuantileFilter,” in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, IEEE, 2024.

[22] S. Dong, Z. Fan, T. Yang, H. Xue, P. Chen and Y. Wu, ”M4: A Framework for Per-Flow Quantile Estimation,” in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, IEEE, 2024.

[23] J. Guo, Y. Hong, Y. Wu, Y. Liu, T. Yang and B. Cui, ”Sketchpolymer: Estimate per-item tail quantile using one sketch”, in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023.

[24] C. Masson, J. E. Rim and H. K. Lee, ”DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees”, in *Proceedings of the VLDB Endowment*, vol. 12, no. 12, 2019, pp. 2195-2205.

[25] G. Cormode, Z. Karnin, E. Liberty, J. Thaler and P. Veselý, ”Relative error streaming quantiles,” in *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2021, pp. 96-108.

[26] R. Ben-Basat, G. Einziger, R. Friedman and Y. Kassner, ”Heavy hitters in streams and sliding windows,” in *35th Annual IEEE International Conference on Computer Communications*, 2016, pp. 1-9.

[27] Y. Zhou, Y. Zhou, S. Chen, and Y. Zhang, “Per-flow counting for big network data stream over sliding windows,” in *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*, 2017, pp. 1–10.

[28] Z. Liu, C. Kong, K. Yang, T. Yang, R. Miao, Q. Chen, Y. Zhao, Y. Tu and B. Cui, ”HyperCalm Sketch: One-Pass Mining Periodic Batches in Data Streams,” in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 14-26.

[29] R. Gu, S. Li, H. Dai, H. Wang, and Y. Luo, ”Adaptive online cache capacity optimization via lightweight working set size estimation at scale”, in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 467-484.

[30] The source codes and the techniqual report of Hermetis. https://github.com/HermetisSketch/Hermetis.

[31] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig. “Cold filter: A meta-framework for faster and more accurate stream processing,” in *SIGMOD Conference*, 2018, pp. 741-756.

[32] Y. Zhang, J. Li, Y. Lei, T. Yang, Z. Li, G. Zhang, and B. Cui, ”On-off sketch: A fast and accurate sketch on persistence”, in *Proceedings of the VLDB Endowment*, vol. 14, no. 2, pp. 128-140, 2020.

[33] 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 *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1015-1025, 2020.

[34] Y. Wu, S. Jiang, S. Dong, Z. Zhong, J. Chen, Y. Hu, T. Yang, S. Uhlig, and B. Cui, ”Microscopesketch: Accurate sliding estimation using adaptive zooming,” in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 2660-2671, 2023.

[35] A. Srivastava, A. C. König and M. Bilenko, ”Time Adaptive Sketches (Ada-Sketches) for Summarizing Data Streams”, in *Proceedings of the 2016 International Conference on Management of Data*, pp. 1417-1432, 2016.

[36] Y. Chabchoub and G. Heébrail, ”Sliding HyperLogLog: Estimating Cardinality in a Data Stream over a Sliding Window,” in *2010 IEEE International Conference on Data Mining Workshops*, 2010, pp. 1297-1303.

[37] H. Tang, Y. Wu, T. Li, C. Han, J. Ge and X. Zhao, ” Efficient identification of TOP- k heavy hitters over sliding windows,” *Mobile Networks and Applications*, vol. 24, no. 5, pp. 1732-1741, 2019.

[38] J. Shan, Y. Fu, G. Ni, J. Luo, Z. Wu, ”Fast counting the cardinality of flows for big traffic over sliding windows, ” *Frontiers of Computer Science*, pp. 119–129, 2017.

[39] O. Papapetrou, M. Garofalakis and A. Deligiannakis, ”Sketch-based querying of distributed sliding-window data streams”, in *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 992-1003, 2012.

[40] Y. Wu, Z. Fan, Q. Shi, Y. Zhang, T. Yang, C. Chen, Z. Zhong, J. Li, A. Shutul and T. Tu, ”She: A generic framework for data stream mining over sliding windows”, in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1-12.

[41] B. Turkovic, J. Oostenbrink, F. Kuipers, I. Keslassy and A. Orda, ”Sequential zeroing: Online heavy-hitter detection on programmable hardware”, in *IFIP Networking Conference (Networking)*, IEEE, 2020, pp. 422–430.

[42] R. Ben-Basat, G. Einziger, I. Keslassy, A. Orda, S. Vargaftik, and E. Waisbard, “Memento: making sliding windows efficient for heavy hitters,” in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, 2018, pp. 254–266.

[43] P. Chen, D. Chen, L. Zheng, J. Li and T. Yang, ”Out of many we are one: Measuring item batch with clock-sketch”, in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 261-273.

[44] 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 *27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2021, pp. 2285-2293.

[45] 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.

[46] Y. Du, H. Huang, Y. -E. Sun, S. Chen and G. Gao, ”Self-Adaptive Sampling for Network Traffic Measurement,” in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021.

[47] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen and L. Xi, ”HeavyKeeper: An Accurate Algorithm for Finding Top- k Elephant Flows,” in *IEEE/ACM Transactions on Networking*, 2019, pp. 1845-1858.

[48] Caida Anonymized Internet Traces 2018 Dataset, http://www.caida.org/data/passive/passive dataset.xml.

[49] A. S. Asrese, S. J. Eravuchira, V. Bajpai, P. S. lahti, and J. Ott. Measuring web latency and rendering performance: Method, tools and longitudinal dataset. https://doi.org/10.5281/zenodo.2547512, January 2019.

[50] E. Assaf, R. Ben-Basat, G. Einziger and R. Friedman, ”Pay for a Sliding Bloom Filter and Get Counting, Distinct Elements, and Entropy for Free,” in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 2204-2212.

[51] N. Rivetti, Y. Busnel and A. Mostéfaoui, ”Efficiently Summarizing Data Streams over Sliding Windows,” in *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015, pp. 151-158.

[52] R. Y. Hung, L.-K. Lee, and H.-F. Ting, “Finding frequent items over sliding windows with constant update time,” *Information Processing Letters*, 2010.