P-Sketch: A Fast and Accurate Sketch for Persistent Item Lookup

Weihe Li and Paul Patras, *Senior Member, IEEE.*

***Abstract*—In large data streams consisting of sequences of data items, those appearing over a long period of time are regarded as persistent. Compared with frequent items, persistent items do not necessarily hold large amounts of data and thus may hamper the effectiveness of vanilla volume-based detectors. Identifying persistent items plays a crucial role in a range of areas such as fraud detection and network management. Fast detection of persistent items in massive streams is however challenging due to the inherently high data rates, while state-of-the-art persistent item lookup solutions routinely require large enough memory to attain high accuracy, which questions the feasibility of deploying them in practice. In this paper, we introduce P-Sketch, a novel approach to persistent item lookup that achieves high accuracy even with small memory (L1 Cache) budgets and maintains high update speed across different settings. Specifically, we introduce the concept of arrival continuity *(hotness)* that counts the number of consecutive windows in which an item appears, to effectively protect persistent items from being wrongly replaced by non-persistent ones. Through meticulous data analysis, we also reveal that items with higher persistence tend to possess a stronger hotness than non-persistent ones. Thus, we harness the information of persistence and hotness, and employ a probability-based replacement strategy to achieve a good balance between memory efficiency, lookup accuracy, and update speed. We also present a theoretical analysis of the performance of the proposed P-Sketch. Through trace-driven emulations, we demonstrate that our P-Sketch yields average F1 score and update throughput gains of up to 10.32× and respectively 2.9×, over existing schemes. Lastly, we show how to further boost the P-Sketch's update speed with Single Instruction Multiple Data (SIMD) instructions.**

***Index Terms*—data stream mining, persistent items, sketch**

I. Introduction

PERSISTENT items convey a lot of valuable information and can often be encountered in different data streaming processes, including malicious network behavior detection [1], click fraud detection [2], and vehicle traffic mining [3]. For instance, certain network threats (e.g., command and control in botnets) involve sending malicious items at a restricted speed but over a long timespan (e.g., only once an hour for 200 days), to avoid detection by intrusion detection systems (IDS) [1], [4]. Thus, as also indicated by [1], [5], finding persistent items can help in identifying stealthy DDoS and botnet attacks. In a similar fashion, automated scripted logic is used to perform click fraud by persistently clicking on advertisements over a long time period, to circumvent discovery by vanilla volume-based detectors and increase revenue in pay-per-click-based online advertising systems [2]. Persistent item detection also helps thwarting such illicit activity. Identifying persistent items in real-time is therefore critical in practice [6].

Persistent items often exhibit repetitive arrival patterns. Consider a stream (i.e., a sequence of items where each may appear multiple times) with N non-overlapping and contiguous time windows. The persistence of an item e is characterized by the number of different windows in which e appears. Its persistence is thus an integer between 0 and N. An item is regarded as α-persistent if its persistence is at least αN ($0 < \alpha \leq 1$), where α is a user-configured threshold that we term as the persistence threshold. Real-time persistent item lookup is however non-trivial, as keeping pace with the ever-increasing speeds of data streams while maintaining high accuracy is hard. For example, in a fully utilized 10 Gb/s link, the detection scheme requires a processing speed of at least 14.88 million packets per second [9]. Further, to achieve high speed operation, it is desirable to access only the CPU cache when processing items, which requires data structures that are compact enough to be accommodated in the L1 and L2 caches, which are much faster than the L3 cache [10].

Existing mechanisms for finding persistent items can be categorized into three types: sample-based [7], coding-based [13] and sketch-based [6], [11]. The core idea of sample-based approaches is to select items with a certain sampling rate and then bookmark the sampled items in a hash-based filter [12]. Even though the time and space overhead can be mitigated effectively via sampling, this kind of approaches may still record many non-persistent items, thus degrading memory efficiency. Moreover, the sampling rate is configured according to the size of the memory, which drops as the memory decreases, leading to increased error rates. To improve the space utilization, coding-based schemes store an item's code [14] rather than the item ID. Nevertheless, they need to encode every item that appears in each window, wasting much space to save non-persistent items. Besides, the computational overhead for encoding and decoding is high, which dramatically decreases the update speed, making it hard to match the speeds of data streams [6], [15]. In particular, even efficient linear time Raptor decoding approaches [14], [17] cannot meet the high update speed requirements of fast item processing tasks [18].

Compared with schemes that record each item's ID or code, sketch-based methods hash items into memory entries (*buckets*) and store the accumulated information of all data streams, achieving fast update speed and small memory usage at the cost of bounded errors [9], [18], [22]–[29], [31]. To find persistent items, such schemes mainly resort to a Bloom filter [12] or employ a state field to only increment persistence counters by one in a given time window, to avoid duplicates [6], [11]. However, Bloom filters incur false positives

Weihe Li and Paul Patras are with the School of Informatics, The University of Edinburgh, United Kingdom. E-mail: {weihe.li, paul.patras}@ed.ac.uk.

This work was partially supported by Cisco through the Cisco University Research Program Fund (Grant no. 2019-197006).

(items that are not in the set are incorrectly reported as being in the set), increasing the detection error. Moreover, each bucket only records the accumulated information, leading to many non-persistent items being mistakenly recognized as persistent, especially under small memory budgets where hash collisions are more likely.

Contributions: In this paper, we introduce a new sketch-based approach for persistent item lookup, named P-Sketch, which simultaneously overcomes the accuracy and throughput limitations of existing solutions, and achieves a good trade-off between accuracy, speed and memory efficiency. Specifically, through data analysis we first unveil a key insight about items encountered in streams: ***items with higher persistence usually have stronger hotness***[1]***, meaning that they will not be absent for a long period.*** Based on this observation, we replace items stored in buckets in a probabilistic manner. Consequently, when a persistent item is held in a bucket, the larger persistence and hotness it has, the more difficult it will be to substitute it with other items. Thus, persistent and non-persistent items become easier to be stored and respectively evicted.

To the best of our knowledge, our work is the first to harness the *hotness* attribute for persistent item lookup. Compared with existing methods, (1) P-Sketch does not utilize pointers and reserves a larger portion of the memory to record persistent items, which yields ***excellent memory efficiency*** and good detection performance, even when strictly relying on the CPU's L1 cache; (2) P-sketch stops hashing once a new item finds a bucket, effectively mitigating redundant hash operations, leading to ***faster update speeds***; we further accelerate the update process of P-Sketch through data parallelism enabled by SIMD instructions [36]; and (3) P-Sketch possesses ***high accuracy***, achieving an average F1 score of 0.93 with 64KB memory (175.63% higher than the currently best-performing persistent item lookup approach [11]). We present a formal analysis of P-Sketch's error bound on persistent item lookup, to theoretically validate its accuracy, which we further demonstrate via a spectrum of trace-driven evaluations. The source code of P-Sketch is publicly available at [20].

The rest of this paper is organized as follows. In Section II, we shed light on the limitations of existing schemes and motivate our design. We present our P-Sketch solution in Section III, then provide a formal theoretical analysis of its performance in Section IV. In Section V we summarize the results of extensive experiments, which confirm P-Sketch's effectiveness. We conclude the paper in Section VI.

II. Background and Motivation

We start by demonstrating that existing schemes for persistent item lookup can not guarantee both high accuracy and efficient memory usage, which in turn motivates our design.

A series of approximate stream processing schemes have been proposed recently to find persistent items in high-speed data streams. Frequency-based sketches [46], such as Count-Min Sketch [47], estimate persistence and adopt some supplementary data structures, e.g. a Bloom filter [12], to evict duplicates. However, Bloom filters incur false positive errors owing to hash collisions, especially under tight memory settings, which degrades the detection accuracy. In addition, Count-Min Sketch does not record the flow key, which results in significant overestimations of items' persistence. For instance, when j different items are hashed into the same bucket in a time window, the mapped counter will increase by j instead of by 1. Moreover, this solution is non-invertible and thus leads to excessive memory accesses, resulting in low update speeds [9].

To improve the lookup accuracy and reduce the detection overhead, Small-Space (SS) [7] tracks the persistence of items in a hash table via sampling. Even though sampling mitigates the space overhead, it still records many non-persistent items, leading to low memory efficiency. Also, the low sampling rate under limited memory increases the lookup error. Other sampling schemes are also suitable for persistent item detection, e.g., reservoir sampling [8], which is a simple probabilistic method for identifying a set of items in an unknown population. The sampling is however random, with no consideration for the frequency of occurrence of any of the items of interest, resulting in modest detection performance (the P-Sketch we introduce in this work attains up to 77% higher accuracy). To tackle the deficiencies of sampling approaches, PIE [13] encodes each item via a Raptor code [14]. By storing the code of each item rather than the ID, PIE effectively alleviates memory usage. However, PIE needs to store the code of each item in a time window even though most items are non-persistent. Besides, encoding and decoding require complicated computation, rendering PIE unable to match high-speed streams. Compared to PIE, which needs to record all items, recent work proposes to use multiple hash tables to track only potential persistent items according to the currently available information about the stream [4]. However, non-invertibility persists, which incurs substantial memory access, leading to low update throughput.

The On-Off sketch [6] aims to further increase detection accuracy and improve memory usage. There are two parts to the On-Off Sketch: the first is a set of persistence counters, and the second consists of g key-value pairs ($g = 2$ in the example shown in Fig. 1) that are associated with each of the counters. Both counters and key-value pairs also have an (On/Off) flag that is used to increase the persistence of the tracked items periodically. Specifically, when a new item arrives (e_3 in the example in Fig. 1), the On-Off Sketch first maps it to a counter via a hash function. If the hashed item was not previously recorded in any key-value pair associated with the counter to which the item mapped, and the flag of that counter is On (indicating this counter has not been accessed in this

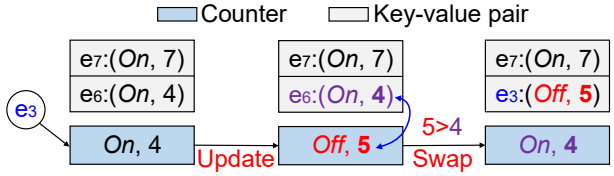


Fig. 1: An insertion example of the On-Off Sketch.

[1]refers to the number of consecutive windows in which an item appears.

time window), the On-Off Sketch first updates the counter's flag to Off and increases its value by one (from 4 to 5 in this example). Then it compares the updated value with the recorded value in each stored key-value pair and, if it finds an entry with a value smaller than the counter's value (e_6 in our example), then the flow key of that entry will be replaced with the key of the newly arrived item, while the value in that key-value pair will inherit the counter's value (i.e., 5). Lastly, the counter's flag and value are swapped with those of the key-value pair that was replaced (On, 4). Unfortunately, since multiple items may be hashed into the same counter, even if e_3 arrives only once, On-Off Sketch's simple replacement strategy may incorrectly identify it as persistent, resulting in low detection accuracy under small memory budgets.

WavingSketch [11] is a generic sketch that can be used in various applications and provides unbiased estimation. It consists of two parts: waving counters and associated multiple counters for the heavy part. Specifically, WavingSketch leverages the waving counter to estimate the item size. If the value of the waving counter exceeds the minimum item size in the heavy part, the content stored in the two counters will exchange. For persistent item lookup, WavingSketch applies a Bloom filter to remove duplicates in a time window. However, extra memory accesses will increase the update time and thus degrade the update throughput. Moreover, when the memory is tight, the severe false positive errors caused by the Bloom filter will dramatically degrade the lookup accuracy.

Our summary of existing persistent item lookup schemes leads us to conclude that (i) it is challenging for existing methods to achieve high detection accuracy under limited memory, especially with the L1 cache; (ii) owing to multiple memory accesses and complex update procedures, their throughput is unable to keep up with high-speed streams, thereby challenging their practical deployment. This motivates us to design a new mechanism that simultaneously achieves high memory efficiency, high update throughput and low detection error, thereby overcoming the inefficiencies observed, as we discuss in detail next.

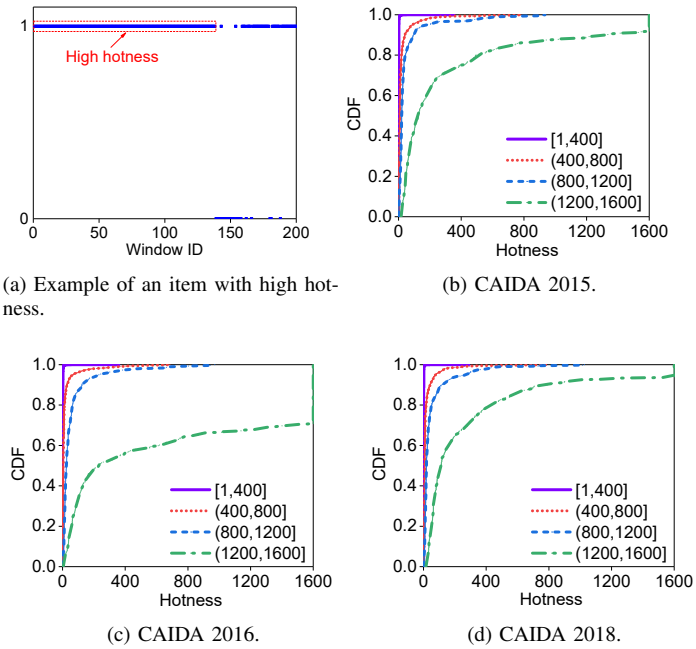
III. P-Sketch Design

Next we introduce our P-Sketch design, starting with its core principles, followed by its detailed functionality, including the corresponding data structure and basic operations.

A. Core Principles

Since sketches can achieve small memory footprints, high accuracy, and fast insertion and query speeds [45], we leverage this data structure in our design. Specifically, we instantiate our P-Sketch as two-dimensional array of buckets, containing r rows, each with w memory entries (*buckets*), where a bucket tracks the values of items hashed to that bucket [47]. Unlike most existing sketch-based approaches [9], [43], [47] that hash each item key into a bucket in each row and adds the associated counter to the item value, P-Sketch stops the hash operation once an incoming item finds an available bucket in a row. This enables storing more items in the sketch and thus significantly improves the memory efficiency. When an item cannot successfully find an available bucket, P-Sketch adopts a probability-based replacement to decide whether a new item can replace one already stored in a bucket. The probability is calculated based on the following key observation:

1) Key Observation: ***Items with a higher persistence typically have a stronger hotness than items with lower persistence.*** To verify this property, we analyze the relationship between the persistence and the hotness of different items with three one-hour real-world IP traces, namely CAIDA 2015, CAIDA 2016, and CAIDA 2018 [43]. Each trace consists of around 0.52M, 0.73M, and 0.77M items, respectively. We configure the total number of windows as 1600 [6] and classify items into four categories according their persistence values (i.e., $[1, 400]$, $(400, 800]$, $(800, 1200]$, $(1200, 1600]$). Note that the above classification method is based on quartiles. Other methods can also be leveraged, like using five classes instead of four, since our design principles are independent of the classification method.



(a) Example of an item with high hotness.

(b) CAIDA 2015.

(c) CAIDA 2016.

(d) CAIDA 2018.

Fig. 2: Relationship between the persistence and hotness under different datasets.

Firstly, we randomly select a persistent item from the CAIDA 2016 trace. As shown in Fig. 2(a), 1 indicates this item appears in a time window; conversely, 0 marks its absence. We observe that the item's arrival demonstrates high hotness, meaning it will not be absent for a long period of time. Then, we analyze the maximum hotness of each item under three different traces and report the results in the form of Cumulative Distribution Function (CDF). Figs 2(b), (c), and (d) reveal that items with higher persistence own higher hotness. Specifically, from Table I, we observe that the average hotness of items with persistence between $(800, 1200]$, $(1200, 1600]$ is much larger than that of items with persistence between $[1, 400]$ and $(400, 800]$.

TABLE I: Relationship between the persistence and average hotness in 3 datasets.

Persistence \ Dataset	CAIDA15	CAIDA16	CAIDA18
[1,400]	1.333639	1.450933	1.190465
(400,800]	23.19014	18.62784	20.46747
(800,1200]	52.01818	62.21523	54.44717
(1200,1600]	345.6216	653.1479	295.4784

2) Special Case: In practical scenarios, certain items may exhibit periodic occurrence patterns, appearing at regular intervals, e.g. once every two time windows. Despite their high persistence, these items may have a low hotness value. Upon conducting a detailed data analysis, we observed that in practice such items constitute only a tiny fraction of the overall persistent items and for persistent items with low hotness values, their arrivals likely follow a periodic pattern. In our analysis, we randomly selected three traces from the CAIDA datasets of 2015, 2016, and 2018. A detailed description of these traces is provided in Section V.A. Each trace is divided into 1600 windows, and we identify persistent items as those that appear in over 800 windows. Specifically, in the selected traces we found 682, 587, and 747 persistent items, respectively, which indicates that the proportion of persistent flows is extremely small. We then classify the hotness of these persistent items into four ranges: [0, 100), [100, 500), [500, 1000), and [1000, 1600]. Table II illustrates the distribution of hotness values among the persistent items.

From the table, we observe that the majority of persistent items exhibit high hotness values. For instance, in the CAIDA 2016 dataset, only 4.6% of persistent items have a hotness value smaller than 100. Additionally, we find that less than 2.5% of persistent items in the CAIDA 2015 dataset, 1% in the CAIDA 2016 dataset, and 2.68% in the CAIDA 2018 dataset have a hotness value smaller than 50 (not displayed in the table). This analysis leads us to conclude that most persistent items possess high hotness values. Therefore, hotness can serve as a valuable metric for protecting persistent items.

TABLE II: Proportion of persistent items with different hotness in 3 datasets.

Hotness \ Dataset	CAIDA15	CAIDA16	CAIDA18
[0,100)	6.45%	4.60%	4.82%
[100,500)	33.43%	30.49%	31.86%
[500,1000)	27.42%	26.06%	27.18%
[1000, 1600]	32.70%	38.85%	36.14%

Based on these observations, we conclude that items with higher persistence tend to have higher hotness than non-persistent ones. Thus, our P-Sketch estimates the replacement probability based on the value of persistence and hotness counters, to decide whether to record newly arriving items. Specifically, the successful replacement probability decreases as persistence and hotness counters increase, indicating that when the bucket stores a persistent item, it can be hardly substituted by non-persistent ones. Even for items with low hotness values, our probability-based eviction method can effectively prevent items with higher persistence from being easily evicted from the bucket, ensuring a high detection accuracy.

3) Differences from Existing Work: Note that while both P-Sketch and On-Off Sketch utilize an *On/Off* flag to track item persistence, they differ significantly in their update approaches. Unlike the naive eviction strategy employed by On-Off Sketch, our approach incorporates a probabilistic eviction mechanism based on multi-dimensional features. This strategy provides enhanced protection for potentially persistent items, safeguarding them from being easily evicted by a large number of non-persistent ones.

Furthermore, P-Sketch also differs from the Unbiased Space-Saving (USS) [30] method in two significant aspects. Firstly, our method employs a multi-feature-based eviction approach that considers factors such as item persistence and hotness when evicting incumbent items. This enables more accurate eviction decision-making. Secondly, P-Sketch eliminates the need to scan all buckets to identify the bucket with the minimum value, as required by USS. This optimization offers P-Sketch significantly higher update speeds.

B. P-Sketch Data Structure

Fig. 3 illustrates the data structure of the proposed P-Sketch, which consists of a two-dimensional array with r rows and w columns. Each row is assigned a different pairwise-independent hash function, denoted by $h_1, ..., h_r$. Let $B_{i,j}$ represent the bucket at the i-th row and j-th column, where $1 \leq i \leq r$, $1 \leq j \leq w$. Each bucket comprises four fields: (i) P denotes the accumulated persistence of the current candidate persistent item; (ii) H indicates the hotness of the current candidate persistent item; (iii) Key stands for the key of the current candidate persistent item; and (iv) $Flag$ is a status field (*On/Off*) of bucket $B_{i,j}$ to indicate whether the current candidate persistent item has been accessed or not in the recent time window, with status *On* meaning this item has not arrived in the current window yet. Otherwise, the persistence counter will increase by one and the status is set to *Off*. With the aid of the flag bit, P-Sketch effectively removes duplicates in a time window. At the beginning of each new window, all the flags will be reset to *On*. Note that the data structure keeps a fixed memory size, and thus we can pre-allocate static memory space before the measurement task begins [9], [43]. Although including a hotness counter reduces slightly the number of buckets that can be accommodated within a given memory budget, the benefits gained from protecting persistent items far outweigh the storage overhead incurred, as we reveal in Section V.

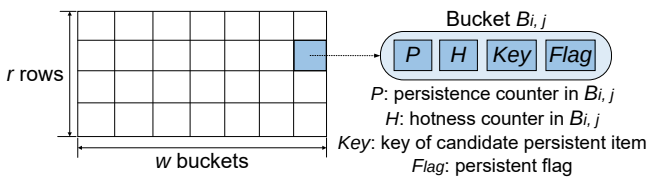


Fig. 3: Data structure of P-Sketch.

C. Basic Operations

The main operations of P-Sketch include *Update* and *Query*. *Update* is used to insert arriving items into the sketch, and *Query* is utilized to report the persistent items whose estimated persistence is over a pre-defined threshold.

1) ***Update****:* Algorithm 1 illustrates the update procedure of P-Sketch, which is triggered when a new item arrives, and it can be divided into two stages: finding an available bucket (*Stage I*) and probability-based replacement (*Stage II*). Initially, all the counters and the flags are set to 0 and *On*, respectively (Line 1).

Stage I: Once a new item e_m arrives, P-Sketch first checks whether this item belongs to a new time window (Lines 3–10). If so, P-Sketch first scans all the buckets, and for those whose flag bit is On (indicating that the candidate persistent item did not arrive in the last window) the corresponding hotness counter is decremented by 1. Then, the flags of all buckets are reset to On.

To insert the new item e_m, P-Sketch initially maps it into the bucket in the first row with the hash function h_1. (i) If the hashed bucket is empty, P-Sketch initializes the bucket by recording the item's key into that, setting both the persistence and hotness counters as 1, and turning the status flag to *Off* (Lines 11–17). (ii) If the hashed bucket has been occupied by e_m and the flag of the bucket is *On*, we increase the persistence and hotness counters by 1, and set the flag to *Off* (Lines 18–22). In either case, an available bucket for the current item is found. (iii) If the hashed bucket is already occupied by e_m and the flag of the bucket is set to *Off*, we disregard the incoming item (Lines 23–24). (iv) Otherwise, P-Sketch scans the buckets in the next row one by one with hash functions $h_2, h_3, ..., h_r$, respectively, to find an available bucket for the current item. Once an appropriate bucket is found, the process terminates. Unfortunately, suppose the new arrival cannot successfully find a bucket, indicating a hash collision occurs in each row. In that case, P-Sketch will compare it to the stored item with the smallest importance (the sum of persistence and hotness counters) among r rows, to determine whether to accept or evict the current item (Lines 25–26). Note that P-Sketch randomly selects one to update if multiple buckets have the same minimum value.

Stage II: If the status flag of the corresponding bucket is Off, indicating that the candidate persistent item stored in this bucket has arrived in current time window, the newly arrived item will give up the replacement (Lines 28–30). This approach guarantees the retention of the incumbent item in the sketch, preventing its eviction during this stage. To confirm the effectiveness of this strategy, we conduct experiments comparing P-Sketch with and without this particular abandonment (PA) in Section V-F. Otherwise, P-Sketch will estimate the replacement probability according to the smallest sum of persistence and hotness counters, which is calculated as $\frac{1}{\eta(B_{i,j}.P+B_{i,j}.H)+1}$. Here, η is a predefined constant (e.g., $\eta = 18$). If a candidate persistent item is stored in the bucket, the larger the value of persistence and hotness it has, the harder it is to be successfully substituted by others. On average, a new item has to arrive $\eta(B_{i,j}.P + B_{i,j}.H) + 1$ times to

Algorithm 1: P-Sketch's Update operation

Input: an item e_m, hash functions $h_1, h_2, ..., h_r$, $min \leftarrow +\infty$, $W \leftarrow$ window size, $c \leftarrow 0$

1 **Initialization:** The persistence and hotness counters, item key and flag of each bucket are initialized to 0, $NULL$ and On, respectively.

`// Stage I: finding an available bucket`

2 **for** $i = 1$ *to* r **do**

3 $c \leftarrow c + 1$;

4 **if** $c \mod W == 0$ **then** `// a new window`

5 **for** $j = 1$ *to* r **do**

`// traverse all buckets`

6 **for** $q = 1$ *to* w **do**

7 **if** $B_{j,q}.flag == On$ **then** `// this item has not arrived in the last window`

8 $B_{j,q}.h \leftarrow \max(B_{j,q}.h - 1, 0)$;

9 **else**

10 $B_{j,q}.flag == On$; `// reset the flag in all counters`

11 $index = h_i(e_m.key)$

12 **if** $B_{i,index} == NULL$ **then**

13 $B_{i,index}.key \leftarrow e_m.key$;

14 $B_{i,index}.p \leftarrow 1$; `// persistence counter`

15 $B_{i,index}.h \leftarrow 1$; `// hotness counter`

16 $B_{i,index}.flag \leftarrow Off$;

17 **return**;

18 **else if** $B_{i,index}.key == e_m.key \wedge B_{i,index}.flag == On$ **then**

19 $B_{i,index}.p \leftarrow B_{i,index}.p + 1$;

20 $B_{i,index}.h \leftarrow B_{i,index}.h + 1$;

21 $B_{i,index}.flag \leftarrow Off$;

22 **return**;

23 **else if** $B_{i,index}.key == e_m.key \wedge B_{i,index}.flag == Off$ **then**

24 **return**;

25 **else if** $B_{i,index}.p + B_{i,index}.h < min$ **then**

26 $min \leftarrow B_{i,index}.p + B_{i,index}.h$;

27 $R \leftarrow i$; $M \leftarrow h_R(e_m.key)$;

`// Stage II: Probability-based replacement`

28 **if** $B_{R,M}.flag == Off$ **then**

29 Discard the newly arrived item;

30 **return**;

31 **if** $random(0,1) < \frac{1}{\eta(B_{R,M}.p+B_{R,M}.h)+1}$ **then**

32 $B_{R,M}.key \leftarrow e_m.key$;

33 $B_{R,M}.p \leftarrow B_{R,M}.p + 1$;

34 $B_{R,M}.h \leftarrow B_{R,M}.h + 1$;

35 $B_{R,M}.flag \leftarrow Off$;

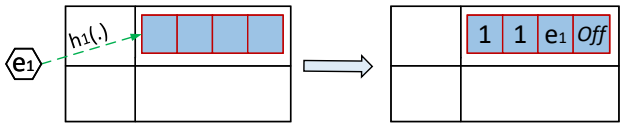
36 **return**;

37 **else**

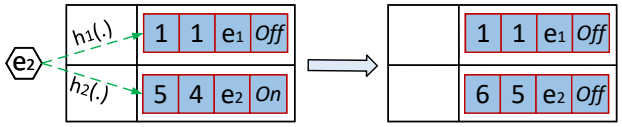
38 Discard the newly arrived item;

39 **return**;

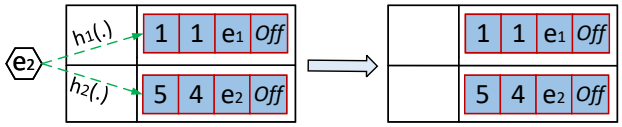
replace the item stored in the bucket [32]. In this fashion, the persistent and non-persistent items become easy to be saved and replaced, respectively. The replacement process involves two cases: (i) if the replacement is successful, P-Sketch will update the item key stored in the bucket with the new item's key, turn the flag to Off, and finally increment the counters by 1 [23], [32], [48] (Lines 31–36); (ii) if the replacement is unsuccessful, P-Sketch will evict the new arrival (Line 38–39).



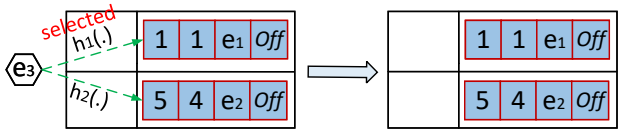
(a) e_1 finds an empty bucket.



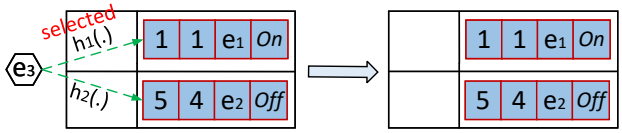
(b) e_2 updates its counters.



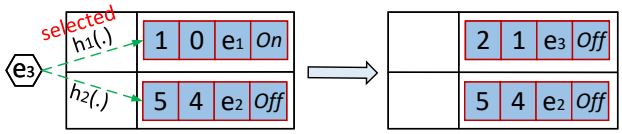
(c) e_2 gives up updating its counters.



(d) e_3 fails to replace e_1 (e_1 has arrived in current window).



(e) e_3 fails to replace e_1 (without triggering the replacement successfully).



(f) e_3 successfully replaces e_1.

Fig. 4: An example of P-Sketch's update process.

A Running Example: We apply several examples to illustrate the algorithm's update operation, as depicted in Fig. 4. For ease of understanding, we set $r = 2$ and $\eta = 1$ in these examples. When an item e_1 arrives, P-Sketch first maps it to a bucket with the hash function $h_1(e_1.key)$. Since the bucket $B_{1,h_1(e_1.key)}$ is empty (Fig. 4(a)), P-Sketch inserts e_1 into the bucket and stops hash actions. After that, to insert e_2, P-Sketch first maps it to a bucket via $h_1(e_2.key)$, where it collides with the item e_1. Then P-Sketch finds e_2 has been stored in the bucket with $h_2(e_2.key)$. As Fig. 4(b) shows, if the flag of e_2 is On, the corresponding persistence and hotness counters increase by 1 and the flag is updated to Off. Otherwise, the status of the bucket storing e_2 does not change in this time window (Fig. 4(c)). When e_3 arrives (Fig. 4(d,e,f)), since no usable bucket could be found, P-Sketch selects the bucket with a smaller sum of both counters and thus $B_{1,h_1(e_3.key)}$ is selected. As shown in Fig. 4(d), the flag of the selected bucket is Off, indicating item e_1 has arrived in current epoch and thus e_3 will be discarded. As displayed in Fig. 4(e), if the replacement of e_3 is unsuccessful, no change occurs in that bucket. Fortunately, as depicted in Fig. 4(f), if the flag of e_1 is On and e_3 successfully triggers the replacement mechanism with a replacement probability of 0.5, the item key in the bucket $B_{1,h_1(e_3.key)}$ sets to e_3, both counters increase by 1, and the flag turns to Off.

2) ***Query****:* To obtain persistent items, P-Sketch traverses all buckets, and if the estimated persistence $\hat{p}_i$ of an item e_i is greater than the pre-defined threshold αN, then e_i is reported as persistent.

D. A New Variant: Optimization with Fingerprints

To further optimize the memory usage of P-Sketch, we propose a new variant P-Sketch*. Compared with the vanilla P-Sketch, we leverage the *fingerprint* to replace the item key to optimize the memory usage‘. The fingerprint of an item is obtained through a specific hash function, producing a unique hash value [18]. Although hash collisions can occur among different items, the probability of such events is relatively low and can be considered negligible. For instance, in a scenario where the fingerprint size is set to 32 bits, and each row consists of 340 buckets, the probability of fingerprint collisions in a dataset containing 1,000,000 items is estimated to be 6.85×10^{-7}, indicating a very low likelihood of a collision. Despite the memory efficiency improvement achieved by harnessing fingerprints, there is a trade-off in terms of processing speed. We thoroughly analyze this trade-off between detection accuracy and processing speed in Section V-H.

In contrast to vanilla P-Sketch, which utilizes separate fields to track persistence, hotness, and flag, we can optimize the approach by utilizing a single 4-byte counter to track this information. The value of persistence and hotness counters is inherently bounded by the number of time windows N. Before initiating the detection task, we evaluate the feasibility of counter compression. With a 4-byte counter, allocating the lowest 16 bits to record item persistence allows for a maximum persistence value of 65,535. Similarly, assigning the highest 15 bits to record the hotness accommodates a maximum value of 32,767. If the number of time windows is less than 32,767, the compression of the flag and two counters into a 4-byte counter is feasible. For time windows ranging from 32,767 to 65,535, the two counters can be compressed into a 4-byte counter and a separate field can be used to record the flag. However, suppose the number of time windows exceeds 65,535. In that case, this approach to compression is no longer feasible, and we revert to the default method of tracking the persistence and hotness values in separate counters. In our specific case, with 1,600 time windows [6], successful compression into a 4-byte counter can be achieved. It is worth noting that conducting the compression operation with finer granularity

has the potential to enhance detection accuracy. We leave for future work the optimization of counter utilization to further improve the lookup process.

IV. Mathematical Analysis

We present a theoretical analysis of P-Sketch with a view to detection accuracy. We first derive the relationship between hotness and persistence. Then we prove that the expected persistence estimation given by P-Sketch is no more than the real persistence, followed by deriving the error bounds of P-Sketch.

A. Relationship Between Hotness and Persistence

Suppose we have a sequence of time windows consisting of a total of N windows, each of duration T. We assume that the arrival of items is independent and random, and the probability of arrival in each time window is θ.

First, let's calculate the expected value of hotness H, denoted as $E(H)$. Assuming an item appears in consecutive k time windows, the hotness H is equal to k. We can use the probability mass function of the binomial distribution to calculate the probability of having hotness k:

$$P(H=k)=\binom{N}{k}\cdot\theta^k\cdot(1-\theta)^{N-k}, \tag{1}$$

where $\binom{N}{k}$ represents the binomial coefficient, which selects k successive windows out of N windows.

To compute the expected value of hotness $E(H)$, we sum up the products of k and the corresponding probabilities:

$$E(H)=\sum_{k=0}^{N}k\cdot\binom{N}{k}\cdot\theta^k\cdot(1-\theta)^{N-k} \tag{2}$$

$$=\sum_{k=0}^{N}k\cdot\frac{N!}{k!(N-k)!}\cdot\theta^k\cdot(1-\theta)^{N-k} \tag{3}$$

We can simplify the expression further by observing that

$$k\cdot\frac{N!}{k!(N-k)!}=\frac{N!}{(k-1)!(N-k)!} \tag{4}$$

Therefore,

$$E(H)=\sum_{k=0}^{N}\frac{N!}{(k-1)!(N-k)!}\cdot\theta^k\cdot(1-\theta)^{N-k} \tag{5}$$

We can observe that when the element appears consecutively in more time windows, i.e., when k is larger, the corresponding multiplication term $\frac{N!}{(k-1)!(N-k)!}\cdot\theta^k\cdot(1-\theta)^{N-k}$ is also larger. As $E(H)$ is the sum of these multiplication terms, larger multiplication terms contribute more to the value of $E(H)$. From this, we can conclude that higher hotness leads to larger expected value $E(H)$.

Now, let's calculate the expected value of persistence P, denoted as $E(P)$. Persistence P represents the number of distinct time windows in which an item appears. To compute $E(P)$, we need to consider all possible values of P ranging from 0 to N.

Suppose an item appears in v distinct time windows. The probability of having persistence $P=v$ is given by:

$$P(P=v)=\binom{N}{v}\cdot\theta^v\cdot(1-\theta)^{N-v} \tag{6}$$

where P is no smaller than H, and v can be expressed as $k+\psi$, where ψ represents the number of non-consecutive occurrences.

To compute the expected value of persistence $E(P)$, we sum up the products of v and the corresponding probabilities:

$$E(P)=\sum_{v=0}^{N}v\cdot\binom{N}{v}\cdot\theta^v\cdot(1-\theta)^{N-v} \tag{7}$$

Then, we calculate the expected value of the persistence P, denoted as $E(P)$:

$$\begin{aligned}E(P)&=\sum_{v=0}^{N}v\cdot\binom{N}{v}\cdot\theta^v\cdot(1-\theta)^{N-v}\\&=\sum_{v=0}^{N}v\cdot\frac{N!}{v!(N-v)!}\cdot\theta^v\cdot(1-\theta)^{N-v}\\&=\sum_{v=0}^{N}\frac{N!}{(v-1)!(N-v)!}\cdot\theta^v\cdot(1-\theta)^{N-v}\\&=\sum_{k=0}^{N}\frac{N!}{(k+\psi-1)!(N-k-\psi)!}\cdot\theta^{k+\psi}\cdot(1-\theta)^{N-k-\psi}\end{aligned}$$

By comparing the expression for $E(P)$, we can observe that $E(P)$ is directly related to the parameter v, which is influenced by the value of k. As k increases, the value of v also increases. Therefore, we can conclude that when the hotness H is larger, corresponding to a higher value of k, it tends to result in a larger persistence P, indicating that higher hotness is associated with larger persistence.

B. No Over-estimation Error

Theorem 1. *Let $P^t(e_i)$ denote the real persistence of item e_i at any given time t and $\hat{P}^t(e_i)$ represent the estimation of $P^t(e_i)$. $\mathbb{E}\left[\hat{P}(e_i)\right]\le P(e_i)$ when $\eta\ge 1$.*

Proof. We resort to mathematical induction [32], [33] to prove this theorem. At the beginning ($t=0$), $\hat{P}(e_i)=0$. We assume this theorem holds at the $(t-1)$-th time window, which is $\mathbb{E}\left[\hat{P}^{t-1}(e_i)\right]\le P^{t-1}(e_i)$.[2] Then at the t-th time window, there exist two cases:

Case 1: If an item ***other than*** e_i arrives, then we get $\mathbb{E}\left[\hat{P}^t(e_i)\right]\le\mathbb{E}\left[\hat{P}^{t-1}(e_i)\right]$ since the persistence estimation for e_i can only decrease or stay the same at the t-th window. Thus, we have

$$\mathbb{E}\left[\hat{P}^t(e_i)\right]\le\mathbb{E}\left[\hat{P}^{t-1}(e_i)\right]\le P^{t-1}(e_i).$$

Because $P^{t-1}(e_i)\le P^t(e_i)$, we obtain that the hypothesis holds at the t-th time window in this case.

[2]Note that here superscript t represents a time window rather than exponentiation.

Case 2: If item e_i arrives at the t-th time window, we employ $\Delta\mathbb{E}^t_{e_i}$ to denote the variation of $\mathbb{E}\left[\hat{P}\left(e_i\right)\right]$ in an adjacent window, which is

$$\Delta\mathbb{E}^t_{e_i} = \mathbb{E}\left[\hat{P}^t\left(e_i\right) - \hat{P}^{t-1}\left(e_i\right)\right].$$

If e_i has been stored in the bucket and the flag of the corresponding bucket is On, we get $\hat{P}^t\left(e_i\right) - \hat{P}^{t-1}\left(e_i\right) = 1$. Otherwise, $\hat{P}^t\left(e_i\right)$ remains the same in this time window.

If e_i is not stored in the bucket, e_i will try to replace the item stored in the bucket with the minimum sum with a probability of $\frac{1}{\eta(B_{i,j}.P+B_{i,j}.H)+1}$. If the replacement is successful, the estimation value for e_i is increased by $(B_{i,j}.P+1)$. Contrarily, the increment of the estimated value for e_i is 0. Thus, the expected increment is $(B_{i,j}.P+1)\cdot\frac{1}{\eta(B_{i,j}.P+B_{i,j}.H)+1}$. Since $B_{i,j}.H \geq 0$, we obtain that the expected increment is no more than 1 when $\eta \geq 1$.

Therefore, the expected incremental persistence of item e_i between two consecutive time windows is not greater than 1 in all scenarios, which is $\mathbb{E}\left[\hat{P}^t\left(e_i\right)\right] - \mathbb{E}\left[\hat{P}^{t-1}\left(e_i\right)\right] \leq 1$ and $\Delta\mathbb{E}^t_{e_i} \leq 1$. Thus, $\mathbb{E}\left[\hat{P}^t\left(e_i\right)\right] \leq \mathbb{E}\left[\hat{P}^{t-1}\left(e_i\right)\right] + 1$. We get

$$\mathbb{E}\left[\hat{P}^t\left(e_i\right)\right] \leq \mathbb{E}\left[\hat{P}^{t-1}\left(e_i\right)\right] + 1 \leq P^{t-1}(e_i) + 1.$$

Since e_i has arrived in the t-th window, we get $P^{t-1}(e_i) + 1 = P^t(e_i)$. Thus, the induction hypothesis also holds in this case. Theorem 1 is proven. □

C. Error Bounds of P-Sketch

Lemma 1. *Let c^t stand for the persistence value of the minimal bucket among the r-hashed buckets at the t-th window. Then $\hat{P}^t\left(e_i\right) \leq P^t\left(e_i\right) + c^t$.*

Proof. We leverage a case analysis [32] to validate this lemma.

First, if item e_i is not in the bucket at the t-th time window during the query process, we get $\hat{P}^t\left(e_i\right) = 0$ and thus the theorem holds. On the contrary, if item e_i stores in the bucket at the query time, there are two cases.

Case 1: e_i enters the bucket without any replacement actions. This case means that e_i holds in its mapped bucket all the time. Thus, the estimated persistence $\hat{P}^t\left(e_i\right)$ is equal to its real persistence $P^t(e_i)$. Thus, the theorem holds.

Case 2: e_i enters the bucket via replacing other items. Consider the item e_i last entered the bucket in the l-th window. At that time, $\hat{P}^l\left(e_i\right) = c^{l-1} + 1$, where $\hat{P}^l\left(e_i\right)$ indicates the estimated persistence of item e_i at time l and c^{l-1} denotes the persistence value of the minimal bucket in the $(l-1)$-th window. Since the persistence counter of the minimal bucket only increases after an item arrives, the counter's value is either incremented by one or remains the same. Thus, $\hat{P}^l\left(e_i\right) \leq c^{l-1} + 1$. Then, assume item e_i arrives q times between the l-th window and the present window (t-th window); we obtain $q \leq P^t(e_i) - 1$ as e_i arrived once at time l. Therefore, we obtain $\hat{P}^t\left(e_i\right) = \hat{P}^l\left(e_i\right) + q \leq c^{l-1} + 1 + P^t(e_i) - 1 = c^{l-1} + P^t(e_i) \leq P^t(e_i) + c^t$.

Since the claim holds in all cases, Lemma 1 is proven. □

(ϵ, δ)-counting is a helpful metric for assessing the error rate of an algorithm [18]. We adopt this to prove that P-Sketch can reach a low underestimation error rate in estimating the persistence of persistent items.

Theorem 2. *Given a small positive number ε that allows εN to be greater than c^t and $\varepsilon N - c^t$ smaller than the number of time windows, for a persistent item entering a bucket at any given time t, when $\eta \geq 1$, $\Pr\left\{P^t\left(e_i\right) - \hat{P}^t\left(e_i\right) \geq \lceil \varepsilon N - c^t \rceil\right\} \leq \frac{N}{w(\varepsilon N - c^t)}$ holds, where w represents the number of buckets in each row, and N stands for the total number of packets in all items.*

Proof. Let $H^t(e_i)$ denote the number of times item e_i arrives before the t-th time window. For an item e_i to replace the item stored in the minimal bucket, the unallocated item requires to arrive $\eta(B_{i,j}.P + B_{i,j}.H) + 1$ times on average. Thus, we use γ to represent the replacement loss when replacing the formerly stored item in bucket $B_{i,j}$, which is $\eta(B_{i,j}.P + B_{i,j}.H) - B_{i,j}.P$. Thus, at any given time t, we have

$$B^t_{i,j}.P = P^t(e_i) - \sum_{x=1}^{H^t(e_i)} \Pr_x\left(e_i\right)\cdot\gamma$$

Given a small positive number ε, the following Markov inequality holds

$$\begin{aligned}
&\Pr\left\{B^t_{i,j}.P \leq P^t\left(e_i\right) + c^t - \varepsilon N\right\} \\
&= \Pr\left\{P^t(e_i) - \sum_{x=1}^{H^t(e_i)} \Pr_x\left(e_i\right)\cdot\gamma \leq P^t\left(e_i\right) + c^t - \varepsilon N\right\} \\
&= \Pr\left\{\sum_{x=1}^{H^t(e_i)} \Pr_x\left(e_i\right)\cdot\gamma \geq \varepsilon N - c^t\right\} \\
&\leq \frac{\mathbb{E}\left[\sum_{x=1}^{H^t(e_i)} Pr_x\left(e_i\right)\cdot\gamma\right]}{\varepsilon N - c^t}
\end{aligned}$$

Assume that all packets from distinct items follow the uniform distribution [18], we obtain

$$\begin{aligned}
\Pr\left\{\Pr_x\left(e_i\right) = \frac{1}{\eta\left(B^t_{i,j}.P + B^t_{i,j}.H\right) + 1}\right\} &= \frac{1}{B^t_{i,j}.P} \\
&= \frac{1}{P^t(e_i) - \mathbb{E}\left[\sum_{x=1}^{H^t(e_i)} \Pr_x\left(e_i\right)\cdot\gamma\right]}
\end{aligned}$$

For ease of understanding, we use δ to denote

$P^t(e_i) - \mathbb{E}\left[\sum_{x=1}^{H^t(e_i)} \Pr_x(e_i)\cdot\gamma\right]$. Then we have

$$\mathbb{E}\left[\sum_{x=1}^{H^t(e_i)} \Pr_x(e_i)\cdot\gamma\right] = \sum_{x=1}^{\mathbb{E}[H^t(e_i)]} \mathbb{E}\left[\Pr_x(e_i)\cdot\gamma\right]$$
$$= \mathbb{E}\left[H^t(e_i)\right] \sum_{P=1}^{\delta} \left[\frac{\gamma}{\eta\left(B^t_{i,j}.P + B^t_{i,j}.H\right)+1}\cdot\frac{1}{\delta}\right]$$
$$= \mathbb{E}\left[H^t(e_i)\right] \sum_{P=1}^{\delta} \left[\frac{\eta(B_{i,j}.P + B_{i,j}.H) - B_{i,j}.P}{\eta\left(B^t_{i,j}.P + B^t_{i,j}.H\right)+1}\cdot\frac{1}{\delta}\right]$$
$$\le \frac{\mathbb{E}\left[H^t(e_i)\right]}{\delta} \sum_{P=1}^{\delta} \frac{\eta}{\eta+1} = \mathbb{E}\left[H^t(e_i)\right]\cdot\frac{\eta}{\eta+1}.$$

Suppose that all packets from different items are hashed evenly to each bucket, we get $\mathbb{E}\left[H^t(e_i)\right] \le \frac{N}{w}$. Thus,

$$\mathbb{E}\left[\sum_{x=1}^{H^t(e_i)} \Pr_x(e_i)\cdot\gamma\right] \le \frac{N}{w}\cdot\frac{\eta}{\eta+1}.$$

Then

$$\Pr\left\{B^t_{i,j}.P \le P^t(e_i) + c^t - \varepsilon N\right\} \le \frac{\mathbb{E}\left[\sum_{x=1}^{H^t(e_i)} \Pr_x(e_i)\cdot\gamma\right]}{\varepsilon N - c^t}$$
$$\le \frac{\frac{N}{w}\cdot\frac{\eta}{\eta+1}}{\varepsilon N - c^t} = \frac{\eta N}{w(\eta+1)(\varepsilon N - c^t)} \le \frac{N}{w(\varepsilon N - c^t)}.$$

Since $B^t_{i,j}.P$ and $\hat{P}^t(e_i)$ both denote the estimated persistence for item e_i at the t-th window. Finally, we obtain

$$\Pr\left\{P^t(e_i) - \hat{P}^t(e_i) \ge \left\lceil \varepsilon N - c^t \right\rceil\right\}$$
$$\le \Pr\left\{\hat{P}^t(e_i) \le P^t(e_i) + c^t - \varepsilon N\right\} \le \frac{N}{w(\varepsilon N - c^t)}.$$

□

To confirm the correctness of the derived error bound, we experiment with a CAIDA16 trace with 0.15M packets and divide it into 600 time windows. We configure ϵ as 0.004 and vary the memory size from 16 to 256KB. The results in Fig.5 show that the empirical value is smaller than the theoretical one, validating the correctness of our theoretical analysis.

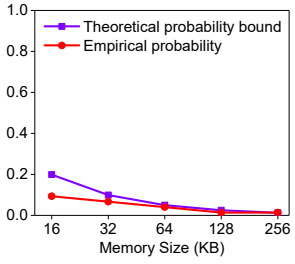


Fig. 5: Theoretical bound and empirical probability of P-Sketch.

V. Evaluation

In this section, we conduct trace-driven experiments to evaluate the performance of P-Sketch. First, we discuss the experimental setup, then analyze the impact of different parameter settings. Afterwards, we compare the performance of P-Sketch with that of prior schemes under different datasets, to demonstrate its superiority. Further, we explore the impact of the persistent item threshold on the detection accuracy and study the contribution of each component to the performance of P-Sketch. Finally, we investigate how to enhance the update throughput of P-Sketch with SIMD instructions.

A. Experimental Setup

Implementation: We implement P-Sketch in C++ and conduct experiments on a physical machine with an eight-core Intel(R) Xeon(R) W-2123 CPU @ 3.60GHz and 32GB DRAM memory, running Ubuntu 16.04 LTS with kernel version 4.15.0-142-generic. We use *MurmurHash* [49] to implement the hash function.

Dataset: We utilize the following datasets to conduct our experiments, each of which is divided into 1,600 time windows [6].

- *IP Trace Dataset* captured by CAIDA on 10 GigE backbone links [42]. Here we leverage four traces (CAIDA15, CAIDA16, CAIDA18, and CAIDA19), collected between 2015–2019. Each trace lasts around 1-hour and contains 0.52M, 0.73M, 0.77M, and respectively 1.53M different items.
- *Data Center (DC) Dataset* collected from the academic data center of the University of Wisconsin in 2010 to analyze network traffic characteristics of data centers in the wild [50]. Here, we use a trace that contains 1.92M distinct items.
- *MAWI Dataset*, whose traffic traces are collected by the MAWI Working Group [51], which investigates the performance of networks in Japanese wide area networks. We pick a trace with a monitoring duration of 15 minutes on Jan 1, 2022, which consists of approximately 8.35M items.

Note that for all traces, we focus on the IPv4 traffic only and utilize the source-destination pairs (8 bytes) as the ID of each item.

Benchmarks: We compare P-Sketch with three existing approaches for persistent item lookup, including Small-Space (SS) [7], WavingSketch [11], and On-Off Sketch [6]. We omit PIE [13] from our evaluation as it does not work with small amounts of memory. WavingSketch provides an unbiased estimation for the frequency of each item via well-designed counters and employs a Bloom filter [12] to remove duplicates in a time window. For a comprehensive comparison, the number of WavingSketch's cells s and On-Off Sketch's key-value pairs q is set as 2, 4 and 16, respectively (s and q marked accordingly in the legend of the following figures next to these approaches). We implemented the On-Off Sketch, Waving Sketch, and SS algorithms using the code provided by their respective authors. By default, we choose the threshold $\alpha = 0.5$ for identifying a persistent item. We also adjust the value of the threshold α to test the robustness of P-Sketch in Section V-D.

When configuring the memory size, all sketches, including our P-Sketch and the baselines considered, are assigned the same memory budget, which ensures a fair comparison among them. Note that the memory allocation approach we take aligns

with many existing works, such as [9], [43], [44], whereby the number of buckets in each row is determined based on the given memory size and the number of rows. To expand on this, in P-Sketch each bucket has a fixed size. By specifying the number of rows r and the total memory size, we can determine the number of buckets in each row based on these parameters. Similarly, in On-Off Sketch, the structure consists of a series of counters, where each counter is associated with g key-value pairs. The size of each counter and key-value pair can be determined in advance based on their respective data structures. Knowing the memory size and the number of key-value pairs associated with each counter, we can calculate the total number of counters in this sketch. The memory allocation method for Waving-Sketch and SS follows the same logic.

Metrics: We consider the following metrics:

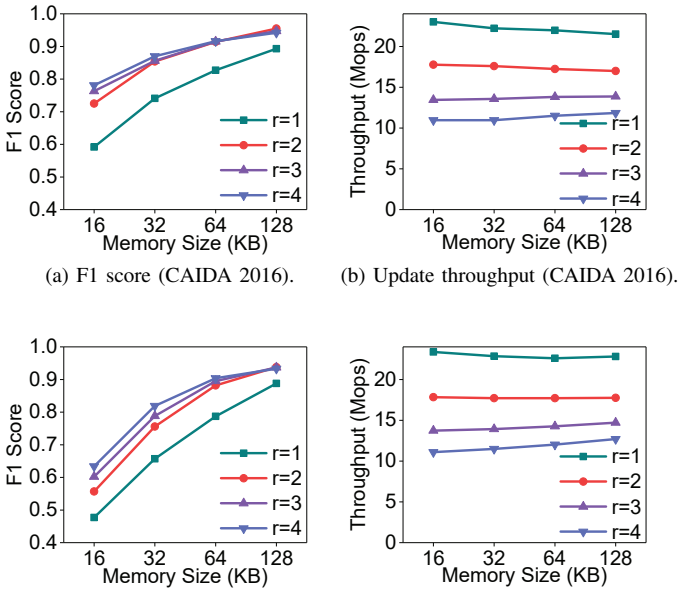
- *Recall*: the ratio of true persistent items reported over all true persistent items.
- *Precision*: the ratio of true persistent items reported over all reported persistent items.
- *F1 score*: $\frac{2\times recall\times precision}{recall+precision}$, which is calculated based to recall and precision, reflecting the accuracy of a method.
- *Average Absolute Error (AAE)*: $\frac{1}{|\Phi|}\sum_{e_i\in\Phi}\left|P(e_i)-\hat{P}(e_i)\right|$, where $P(e_i)$ stands for the real persistence of item e_i, $\hat{P}(e_i)$ is the estimated persistence of that item, and Φ is the query set. Here, the query set encompasses the persistent items reported.
- *Average Relative Error (ARE)*: $\frac{1}{|\Phi|}\sum_{e_i\in\Phi}\frac{\left|P(e_i)-\hat{P}(e_i)\right|}{P(e_i)}$, which evaluates the error rate of the estimated persistence reported by the algorithm.
- *Update Throughput* in million operations (insertions) per second (Mops). We conduct insertions of all packets in all items N and record the total time T. The update throughput is calculated as $\frac{N}{T}$. We repeat all the experiments 5 times and report median values as in [6], [23].

B. Impact of P-Sketch's Parameter Settings

We first measure the influence of different parameter settings on the performance of P-Sketch. The parameter configuration process for P-Sketch is inspired by several existing sketch-based approaches [10], [11], [22], [39]. In contrast to prior methods that entail the specification of multiple parameters [10], P-Sketch simplifies the process by requiring the configuration of only two parameters: the number of rows (r) and the eviction probability (η). Because once the memory size and r have been set, w is also fixed. Therefore we do not consider the parameter w here. We note that existing processors largely have three levels of cache memory (L1 to L3), with the L3 cache being considerably larger than the L1 cache. For instance, the Intel i9-11900K CPU has a 64KB L1 cache memory per core and 16MB of L3 cache. However, unlike the L1 cache with a much smaller memory size, the L3 cache is shared between all cores and is the slowest memory on the CPU. This makes it challenging to handle high-speed items using L3 caches. To attain high-accuracy detection, sketches need to be compact enough to be accommodated in the much faster but tight L1 cache. Thus, similar to recent work [10], [16], we set the memory size between 16 and 128KB for the parameter setting experiments. We apply two CAIDA datasets (CAIDA 2016, 2019) and utilize F1 score and throughput as the evaluation metrics to assess the impact of these parameters.

1) Impact of r: In the ideal scenario, when r is configured as the total number of buckets, indicating each row only contains one bucket, P-Sketch can consistently find the smallest bucket to update. However, as the number of memory access operations for each update is equivalent to the number of buckets in the worst case, the update throughput will drop dramatically, which is unacceptable in practice [23].

Fig. 6 reveals how different r values influence the algorithm's performance under different memory size. η is set to 18 in this experiment (see the next paragraph for results that show this is the optimal value). When the memory budget is small, the detection accuracy increases notably as r increases. The reason is that, items have more chances to be stored in the bucket, mitigating the risk of missing persistent items and thus improving the detection accuracy. We also find that when r increases to 2 and the memory size grows, the larger number of rows improves the performance of P-Sketch only marginally. On the other hand, as shown in Figs 6(b) and (d), the update throughput drops as r increases, since items are likely to undergo more hashing operations on average to find an available bucket. Through selecting the minimum bucket among d buckets, P-Sketch leverages the help of the "power of d choices" paradigm [23], [34], [35]. In this paper, to strike a good balance between detection accuracy and update speed, setting r as 2 is recommended. Alternatively, if prioritizing higher update speed is preferred, the parameter r can be configured as 1 in practice.



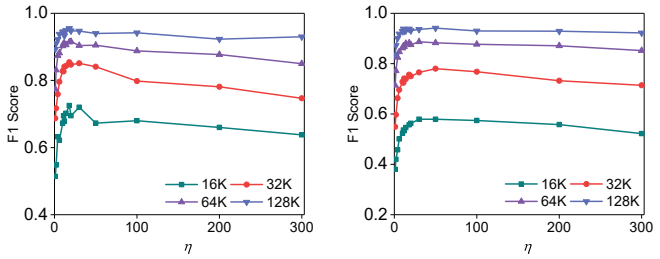
(a) F1 score (CAIDA 2016). (b) Update throughput (CAIDA 2016).

(c) F1 score (CAIDA 2019). (d) Update throughput (CAIDA 2019).

Fig. 6: Detection performance and update throughput under different r settings with two datasets.

2) Impact of η: Recall that η is a user-defined parameter that dictates the replacement probability. If η is configured with a small value, persistent items may be easily replaced by

non-persistent ones. On the contrary, if η is large, even non-persistent items become hard to be evicted from the bucket. Similar to prior work [10], [11], [19], [22], [40], [41], we fix r ($r = 2$ in this case) and vary η to experiment on different datasets to find an appropriate value. Fig. 7 compares P-Sketch's detection performance when varying η and employing different memory sizes. Observe that the F1 score reaches a maximum with different memory sizes when η is in the 10 to 50 range for different traces. Besides, we also conducted additional experiments on various datasets, including MAWI [51]. The results consistently demonstrate a similar trend. Therefore, we choose $\eta = 18$ for P-Sketch, which is the average value that yields the best performance across all the memory budgets considered.



(a) F1 Score (CAIDA 2016). (b) Update Throughput (CAIDA 2019).

Fig. 7: Detection accuracy w/ different η and memory budgets.

C. Persistent Item Lookup Performance

Next, we conduct trace-driven experiments and compare the performance of P-Sketch with that of existing persistent item lookup schemes. We first measure recall, precision, F1 score, AAE, and ARE to evaluate persistent item detection performance. Then, we measure the update throughput of each approach to assess the update speed.

1) Persistent Item Detection: Figs 8–12 illustrate the performance of our P-Sketch and that of the benchmarks considered, with different datasets. Four key takeaways follow from these results:

First, observe that the recall of P-Sketch is higher than that of recent approaches on all traces, with an average improvement over the best-performing sketch-based approach, WavingSketch (with $s = 2$), of 23.52% in the CAIDA 2015 dataset, 37.05% in the CAIDA 2016 dataset, 27.28% in the CAIDA 2018 dataset, 35.11% in the CAIDA 2019 dataset, 99.99% in the DC dataset, and 86.49% in the MAWI dataset (Fig. 8).

Second, considering the features of persistent items and replacing those stored in buckets probabilistically leads to significant gains in lookup precision. Indeed, as shown in Fig. 9, P-Sketch maintains a precision value around 1 even under minimal memory size (e.g., 16KB), which is much higher than that of existing solutions. We also observe that the precision of the benchmarks considered is much lower on the DC and MAWI datasets than when applied on the CAIDA datasets. The reason is that the DC and MAWI traces are less skewed, increasing query difficulty.

Third, as shown in Fig. 10, P-Sketch attains the highest F1 score across various datasets. For instance, under the MAWI dataset, the F1 score of P-Sketch is on average up to 357.16× higher than that of existing methods, confirming the superiority of our approach in terms of detection performance.

Finally, P-Sketch guarantees low estimation error. As reported in Figs 11 and 12, the AAE and ARE of P-Sketch are on average 1.37×/1.80×, 2.08×/2.81×, 1.57×/2.1×, 1.54×/1.99×, 5.37×/5.15×, and 8.14×/10.07× smaller than that of the competing On-Off Sketch solution (with $q = 2$) on the CAIDA 2015, CAIDA 2016, CAIDA 2018, CAIDA 2019, DC, and respectively MAWI dataset.

Analysis: The above results confirm the effectiveness of P-Sketch. We now analyze the reasons behind the performance gains observed over existing algorithms.

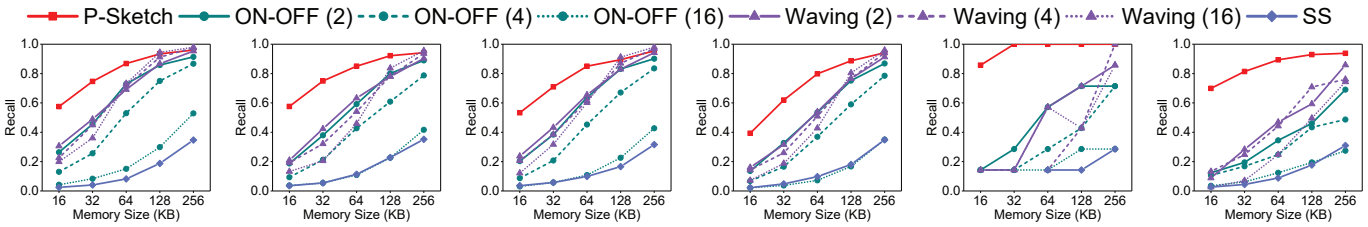
SS keeps track of different items via sampling, so that many non-persistent items are also stored in the hash-based filter, leading to much memory wastage. In addition, the sampling rate of SS needs to be low to keep a small memory, which raises its lookup error. Specifically, from Fig. 9, we observe the precision of SS drops as the memory size increases. The reason is that since the sampling rate is low, increasing space results in more non-persistent items being wrongly identified as persistent, leading to a lower precision.

On-Off Sketch leverages the compact sketch as the data structure, significantly reducing memory usage compared with SS. However, as explained in Section II, On-Off Sketch naïvely replaces the persistent items stored in the key-value pairs, causing many non-persistent items to be mistakenly recognized as persistent, especially under small memory size, thus decreasing detection accuracy.

WavingSketch applies a Bloom filter [12] to vacate duplicates in a time window, which involves larger memory usage as hash tables usually take up much memory [6]. Moreover, the Bloom filter brings significant false positive rates, especially when the memory size is limited (indicating severe hash collisions), yielding low detection accuracy.

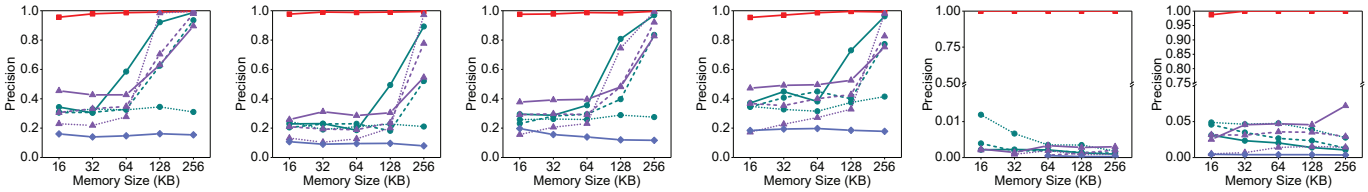
We also observe that when the memory is tight, as the number of cells and key-value pairs increases, the performance of WavingSketch and On-Off Sketch declines. The reason is that when the memory resources are limited, increasing the number of cells and key-value pairs leads to a decreasing number of counters (indicated in Section II), exacerbating hash collisions and thus resulting in low detection accuracy.

Compared with existing schemes, our P-Sketch algorithm incorporates an additional field to capture the hotness of the incumbent item. This enhancement allows us to effectively track and differentiate persistent items from a larger pool of non-persistent ones. Despite the increased memory usage resulting from the inclusion of an additional counter, P-Sketch's probability-based eviction mechanism ensures the preservation of persistent items and significantly reduces the chances of erroneous substitutions by non-persistent items, thereby guaranteeing a high level of detection accuracy. Moreover, considering the substantial disparity between the number of persistent and non-persistent items, the reduced number of buckets in P-Sketch does not overly compromise its ability to maintain high detection accuracy. Specifically, we count



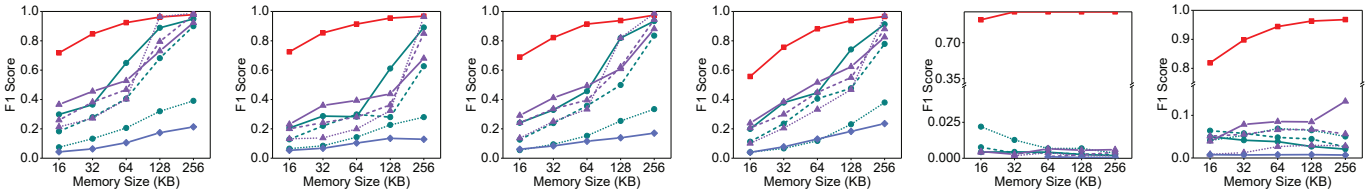
(a) CAIDA 2015. (b) CAIDA 2016. (c) CAIDA 2018. (d) CAIDA 2019. (e) DC. (f) MAWI.

Fig. 8: Persistent item lookup recall with different algorithms, as a function of memory size.



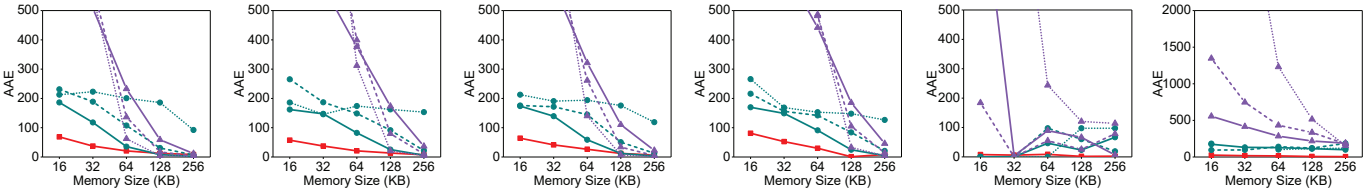
(a) CAIDA 2015. (b) CAIDA 2016. (c) CAIDA 2018. (d) CAIDA 2019. (e) DC. (f) MAWI.

Fig. 9: Persistent item lookup precision with different algorithms, as a function of memory size.



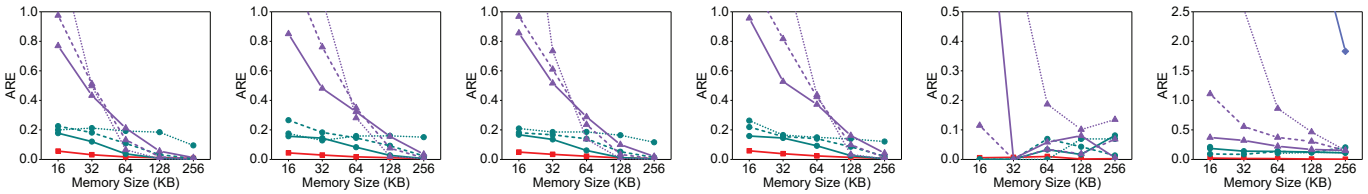
(a) CAIDA 2015. (b) CAIDA 2016. (c) CAIDA 2018. (d) CAIDA 2019. (e) DC. (f) MAWI.

Fig. 10: Persistent item lookup F1 score with different algorithms, as a function of memory size.



(a) CAIDA 2015. (b) CAIDA 2016. (c) CAIDA 2018. (d) CAIDA 2019. (e) DC. (f) MAWI.

Fig. 11: Average Absolute Error (AAE) in persistent item lookup with different algorithms, as a function of memory size.



(a) CAIDA 2015. (b) CAIDA 2016. (c) CAIDA 2018. (d) CAIDA 2019. (e) DC. (f) MAWI.

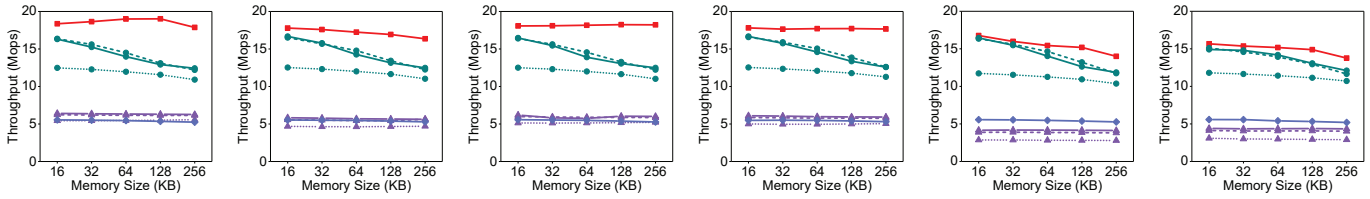
Fig. 12: Average Relative Error (ARE) in persistent item lookup with different algorithms, as a function of memory size.

the number of events where non-persistent items incorrectly replace persistent items during the detection process under the DC and MAWI datasets, respectively. As listed in Table III, we find that our approach effectively prevents persistent items from being effortlessly substituted by non-persistent ones, thus outperforming off-the-shelf schemes in terms of the detection ability.

TABLE III: Number of persistent items being wrongly replaced by non-persistent ones (memory size: 16KB).

Dataset \ Scheme	P-Sketch	ON-OFF (2)	Waving (2)
MAWI	61	10129	8322
DC	1	1314	1238

2) Update Throughput: We now measure the update throughput of P-Sketch and of the benchmarks considered, under different traces and memory settings. As shown in Fig. 13, P-Sketch maintains the highest update speed in all scenarios. Note that all approaches witness a throughput decline as the memory size increases, since they cannot be entirely placed in the cache, and the latency of memory access increases [9]. For instance, for the CAIDA 2015 trace, the



(a) CAIDA 2015. (b) CAIDA 2016. (c) CAIDA 2018. (d) CAIDA 2019. (e) DC. (f) MAWI.

Fig. 13: Update throughput for persistent item lookup with different algorithms, as a function of memory size.

average update throughput of P-Sketch is 31.04%, 193.29%, and 242.67% higher than that of On-Off Sketch (with $q = 2$), WavingSketch (with $s = 2$), and SS, respectively.

Analysis: These results confirm P-Sketch's significant update speed improvements. We briefly investigate the main reasons for the slow update rates of existing solutions.

The data structure of SS consists of many pointers, which is space-inefficient and time-consuming during updates. Also, hash collisions in the hash-based filter increase the number of memory access operations, thus resulting in low update speed [6]. When a new item arrives, On-Off Sketch and WavingSketch perform one hash operation to find a bucket and then iterate through the key-value pairs (cells) associated with that bucket to find an available cell for the new item. Such tedious update processes lower update speeds. Additionally, WavingSketch also relies on a hash-based Bloom filter, which occupies much space and brings excessive memory access operations, further reducing its update speed.

Unlike existing methods, P-Sketch employs a compact data structure without utilizing pointers. Moreover, it stops the hash operations once an item finds an available bucket, which saves much space and limits the number of memory access operations, ensuring a high update speed.

D. Impact of the Persistent Item Threshold

Here we explore the influence of the threshold parameter α on detection performance, varying it between 0.4 and 0.9. We fix the memory size to 32KB, configure the WavingSketch and On-Off Sketch parameter as 4, and adopt the CAIDA 2016 trace for testing. As shown in Fig. 14, P-Sketch maintains its superiority under diverse threshold settings. Specifically, when α is 0.4, the P-Sketch outperforms the state-of-the-art method WavingSketch in terms of recall, with an improvement of 130.87%. In Fig. 14(b), we observe that the precision of P-Sketch is consistently around 1, while that of existing methods decreases noticeably as the threshold increases. The reason is that as α increases, the number of persistent items decreases. Due to the low sampling rate (SS), higher false positive rate (WavingSketch), and rough replacement strategy (On-Off Sketch), more non-persistent items are erroneously labelled as persistent ones by the benchmarks considered, resulting in reduced precision. P-Sketch also obtains the lowest relative error. For example, its ARE is 6.46× less than that of the On-Off Sketch when α is 0.4 (Fig. 14(d)).

E. Impact of Different Number of Time Windows

We investigate the impact of varying the number of windows. For this purpose, we select the CAIDA 2015 trace as the test trace and compare the performance of P-Sketch with that of On-Off Sketch and WavingSketch. The results of our experiments are presented in Table IV. Observe that P-Sketch consistently outperforms the other two algorithms across different numbers of windows. Specifically, when the number of windows is set to 2000, P-Sketch achieves a detection accuracy 2.22 times higher than that of On-Off Sketch and 2.48 times higher than that of WavingSketch.

TABLE IV: F1 score with different number of windows.

F1 Score	400	800	1000	2000	3000
P-Sketch	**0.664**	**0.808**	**0.819**	**0.923**	**0.956**
On-Off Sketch	0.327	0.331	0.325	0.287	0.397
WavingSketch	0.142	0.223	0.231	0.265	0.277

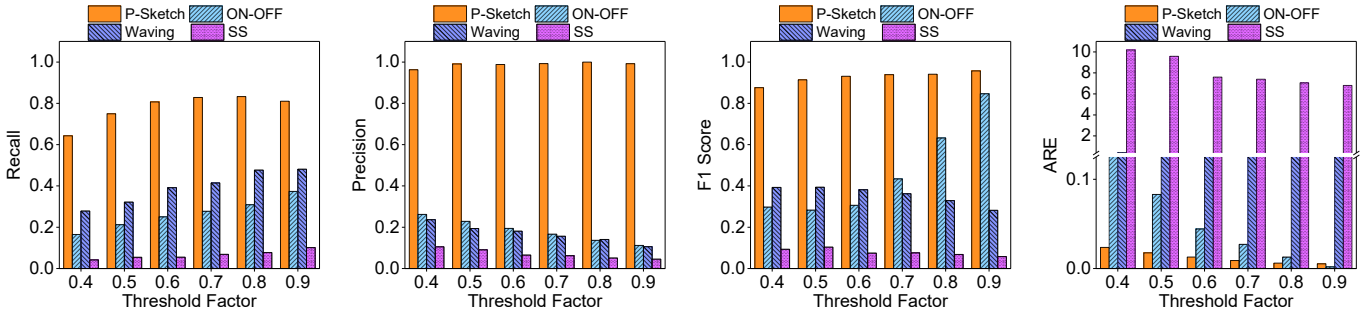
F. Impact of P-Sketch's Design Principles

Recall that P-Sketch incorporates two essential design principles: (i) abandoning hash operations once a new item finds an appropriate bucket rather than hashing it to each row; (ii) refraining from performing replacement actions when the flag of a hashed bucket is *Off* in the first case of Stage 2, and (iii) replacing the item recorded in a bucket based on its persistence and hotness. In the following, we set the memory size to 16KB and employ the CAIDA and MAWI traces to explore the contribution of each principle to the superior performance of P-Sketch.

The results we summarize in Fig. 15 confirm the effectiveness of the first principle. Observe that stopping hashing operations in time saves more space to enable recording more items, resulting in a 3.21% improvement in the F1-Score compared to hashing each item in all rows. Moreover, hashing each item to all rows will result in extra hash operations and thus lead to a longer lookup time. As shown in Figs 15(b) and (c), the number of hash operations and the update throughput are on average 10.77% higher and respectively 16% lower than that of our final P-Sketch design (labeled "Original P-Sketch").

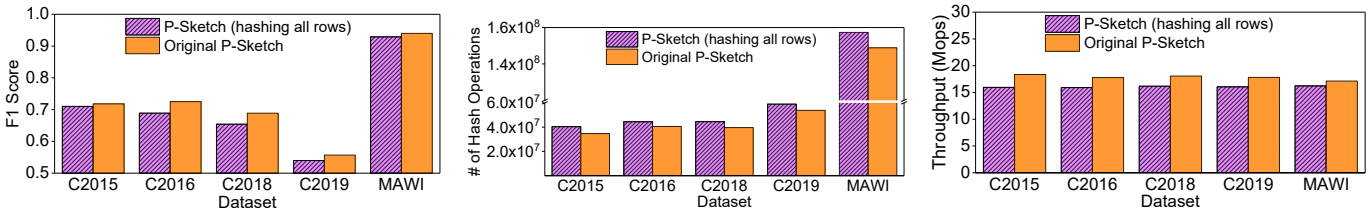
Next, Table V presents the F1 scores of P-Sketch with and without PA, using different CAIDA traces and a memory size of 16KB. The results demonstrate that the default P-Sketch achieves higher accuracy than P-Sketch without PA. Specifically, there is an improvement of 3.61%, 5.69%, and 1.17% in the F1 score across the CAIDA 2015, 2016, and 2018 datasets, respectively. This indicates that the incorporation of PA enhances the detection accuracy of P-Sketch.

Finally, we emphasize the importance of considering the hotness of each item. Fig. 16 reports the performance of P-Sketch with/without taking into account the hotness characteristics of each item. As shown in the figure, the accuracy



(a) Recall. (b) Precision. (c) F1 score. (d) ARE.

Fig. 14: Persistent item lookup performance under different threshold settings on the CAIDA 2016 dataset.

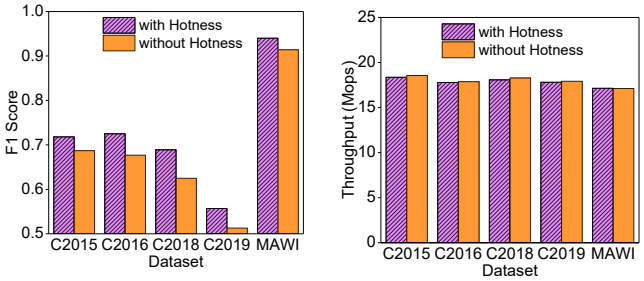


(a) F1 score. (b) Number of Hash Operations (c) Throughput.

Fig. 15: Accuracy and update throughput comparison of P-Sketch with different hash strategies (C stands for CAIDA).

TABLE V: Detection accuracy for P-Sketch with/without the particular abandonment.

F1 Score	CAIDA 2015	CAIDA 2016	CAIDA 2018
P-Sketch	**0.718**	**0.725**	**0.689**
P-Sketch wo PA	0.693	0.686	0.681



(a) F1 score. (b) Throughput.

Fig. 16: Accuracy and update throughput comparison of P-Sketch with/without considering the hotness of each item.

of P-Sketch with hotness is 6.65% higher than that when hotness is not considered. The reason is that hash collisions are frequent under limited memory budgets. With hotness, P-Sketch effectively mitigates the possibility of persistent items being wrongly substituted by non-persistent ones, thus improving the detection accuracy. Fig. 16(b) demonstrates that the update throughput of P-Sketch with/without considering hotness remains almost the same, indicating that incorporating hotness does not introduce significant update overhead.

G. Impact of P-Sketch's Eviction Probability

We compare P-Sketch using various eviction probabilities, including the weighted sum and product approaches. The weighted sum method requires assigning distinct weights to persistence and hotness, introducing additional complexity to the parameter configuration. In our study, we randomly assign weights for persistence and hotness as (1,1), (2,2), (2,5), and (5,2). Alternatively, the product approach involves evicting items tracked in buckets based on a probability of $\frac{1}{(B_{i,j}.P \times B_{i,j}.H)+1}$.

To evaluate the performance of different eviction methods, we randomly select a MAWI trace from 2020 comprising 44.55 million packets. The results in Table VI demonstrate that our default eviction method achieves the highest detection accuracy. Specifically, it outperforms the weighted sum approach with weights (2,2) by 6.29% and the product approach by 853.47%, confirming its effectiveness.

TABLE VI: F1 score with different eviction strategies (Memory size: 16KB).

Eviction Strategy	F1 Score
(1,1)	0.906
(2,2)	0.906
(2,5)	0.906
(5,2)	0.943
Product	0.101
Default	**0.963**

H. Performance comparison of P-Sketch and P-Sketch*

Here, we compare the performance of P-Sketch and P-Sketch*. As seen in Table VII, P-Sketch* yields superior performance, achieving a higher F1 score compared to the original P-Sketch owing to its high memory efficiency. On average, P-Sketch* exhibits an improvement of 7.74% and 6.61% over the CAIDA 2015 and 2016 traces, respectively.

However, generating fingerprints requires additional hash operations, which would arguably slow down the system's update speed. For instance, under the CAIDA 2015 trace, we observe a 4.98% throughput drop when using fingerprints

with a memory size of 32KB. During the query process, we first need to rehash all items to obtain their fingerprints, then determine whether the persistence of each item exceeds the threshold, which increases indeed the query time. Recall however that we present the fingerprint-based P-Sketch* as an alternative to P-Sketch. If the user wishes to prioritize high detection accuracy and is less concerned about update and query times, they can opt for P-Sketch*. Otherwise, we recommend using the default P-Sketch for a good trade-off between detection accuracy and update/query time.

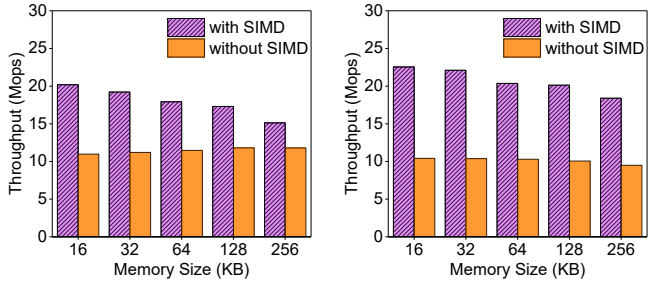
TABLE VII: Performance comparison of P-Sketch and P-Sketch* with different memory sizes.

F1 Score	P-Sketch (CAIDA 15)	P-Sketch* (CAIDA 15)	P-Sketch (CAIDA 16)	P-Sketch* (CAIDA 16)
16KB	0.718	**0.863**	0.725	**0.821**
32KB	0.847	**0.943**	0.854	**0.933**
64KB	0.924	**0.981**	0.914	**0.971**
128KB	0.962	**0.992**	0.955	**0.99**
256KB	0.996	**0.996**	0.968	**0.993**

I. Optimization with SIMD Instructions

Lastly, we boost the update speed of P-Sketch by leveraging SIMD instructions [36]. SIMD instructions can handle sequential access operations in parallel, to accelerate the measurement process in high-speed streams [52]–[55].

Similar to [9], we first use the `MurmurHash3_x64_128` primitive to generate a hash value according to the source/destination address (64-bit) of an item, followed by splitting the hash value into r parts ($r = 4$ in this case; this is because we leverage the primitive `_mm256_set_epi32` to store the hash value, which requires the hash value to be $256 = 64 \times 4$ bits) [56]. Then we apply SIMD instructions to calculate the bucket positions of all r rows and record them into a register array with `_mm256_set_epi64x`. When a new item arrives, P-Sketch with SIMD leverages the primitive `_mm256_cmpeq_epi64` to compare the key of the newly arrived item with the r candidate persistent items' keys in parallel. Compared with the original P-Sketch that takes at most r steps to find an available bucket for a new item, P-Sketch with SIMD only needs 1 step, significantly increasing the comparison speed.



(a) CAIDA 2016. (b) MAWI.

Fig. 17: Update throughput comparison of P-Sketch with/without SIMD Instructions ($r = 4$ in this case).

Fig. 17 compares the throughput of the SIMD-based P-Sketch and the vanilla P-Sketch when employing the CAIDA 2016 and MAWI traces. We vary the memory size from 16KB to 512KB. As illustrated in the figure, with the help of SIMD instructions, the optimized P-Sketch improves the update speed of the vanilla version on average by 57.26% and respectively 104.18%. Note that the throughput is slightly lower than that reported in the previous experiments – recall that we now employ a larger r value. Since the throughput of the SIMD-based P-Sketch is greater than 14.88Mops in all cases, this indicates that the proposed scheme can well match high-speed streams (e.g., 10 Gb/s) [9].

VI. Conclusions

In this paper we introduced P-Sketch, a new sketch-based algorithm for persistent item lookup. P-Sketch replaces persistent items in a probabilistic manner, considering both higher persistence values and stronger hotness features. By this approach P-Sketch alleviates the problem of persistent items being mistakenly substituted by non-persistent ones and thus significantly improves lookup accuracy. We conduct a formal analysis to derive theoretical performance bounds, and trace-driven experiments on multiple datasets to demonstrate that P-Sketch achieves high detection accuracy, high update throughput, as well as high memory efficiency. In essence, P-Sketch substantially outperforms state-of-the-art sketch-based persistent item look up solutions by up to 10.32×, including On-Off Sketch, WavingSketch, and Small-Space. Nonetheless, we show how to exploit SIMD instructions to further enhance the update speed of our P-Sketch solution.

References

[1] Q. Xiao, Y. Qiao, M. Zhen, and S. Chen, "Estimating the Persistent Spreads in High-speed Networks," in Proc. IEEE ICNP, 2014.
[2] N. Immorlica, K. Jain, M. Mahdian, and K. Talwar, "Click Fraud Resistant Methods for Learning Click Through Rates," in Proc. ACM WINE, 2005.
[3] Y.E. Sun, H. Huang, S. Chen, H. Xu, K. Han, and Y. Zhou, "Persistent Traffic Measurement through Vehicle-to-Infrastructure Communications in Cyber-Physical Road Systems," IEEE Transactions on Mobile Computing, vol. 18, no. 7, pp. 1616-1630, 2019.
[4] L. Chen, H. Dai, L. Meng, and J. Yu, "Finding Needles in a Hay Stream: On Persistent Item Lookup in Data Streams," Computer Networks, vol. 181, no. 1, pp. 1-11, 2020.
[5] F. Giroire, J. Chandrashekar, N. Taft, E. Schooler, and D. Papagiannaki, "Exploiting Temporal Persistence to Detect Covert Botnet Channels," in Proc. Springer RAID, 2009.
[6] 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 Proc. VLDB Endowment, 2020.
[7] B. Lahiri, J. Chandrashekar, and S. Tirthapura, "Space-efficient Tracking of Persistent Items in a Massive Data Stream," in ACM DEBS, 2011.
[8] J.S. Vitter, "Random Sampling with a Reservoir," ACM Transactions on Mathematical Software, vol. 11, no. 1, pp. 37-57, 1985.
[9] L. Tang, Q. Huang, and P.P.C. Lee, "MV-Sketch: A Fast and Compact Invertible Sketch for Heavy Flow Detection in Network Data Streams," in Proc. IEEE INFOCOM, 2019.
[10] Z. Zhong, S. Yan, Z. Li, D. Tan, T. Yang, and B. Cui, "BurstSketch: Finding Bursts in Data Streams," in Proc. ACM SIGMOD, 2021.
[11] 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 Proc. ACM KDD, 2020.
[12] B.H. Bloom, "Space/time Trade-offs in Hash Coding with Allowable Errors," Communications of the ACM, vol. 13, no. 7, pp. 422-426, 1970.
[13] H. Dai, M. Shahzad, A.X. Liu, and Y. Zhong, "Finding Persistent Items in Data Streams," in Proc. VLDB Endowment, 2016.
[14] A. Shokrollahi, "Raptor Codes," IEEE Transactions on Information Theory, vol. 52, no. 6, pp. 2551-2567, 2006.

[15] H. Dai, M. Shahzad, A.X. Liu, and Y. Zhong, "PIE: Technical Report," http://cs.nju.edu.cn/daihp/dh/PIE-TR-TON18.pdf, 2018.
[16] J. Huang, W. Zhang, Y. Li, L. Li, Z. Li, J.Ye and, J. Wang, "ChainSketch: An Efficient and Accurate Sketch for Heavy Flow Detection," in IEEE/ACM Transactions on Networking, 2022, doi: 10.1109/TNET.2022.3199506.
[17] M. He, C. Hua, W. Xu, P. Gu, and X.S. Shen, "Delay Optimal Concurrent Transmissions With Raptor Codes in Dual Connectivity Networks," IEEE Transactions on Network Science and Engineering, vol. 8, no. 2, pp. 1478-1491, 2021.
[18] J. Gong, T. Yang, H. Zhang, H. Li, S. Uhlig, S. Chen, L.Uden, and X. Li, "HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows," in Proc. USENIX ATC, 2018.
[19] T. Yang, J. Jing, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic Sketch: Adaptive and Fast Network-wide Measurements," in Proc. ACM SIGCOMM, 2018.
[20] "P-Sketch Code," https://git.ecdf.ed.ac.uk/s2187730/P-Sketch.git
[21] Q. Huang, P.P.C. Lee, and Y. Bao, "SketchLearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference," in Proc. ACM SIGCOMM, 2018.
[22] T. Yang, J. Gong, H. Zhang, L. Zou, L. Shi, and X. Li, "HeavyGuardian: Separate and Guard Hot Items in Data Streams," in Proc. ACM KDD, 2018.
[23] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Mao, P. Liu, R. Zhang, and J. Jiang, "CocoSketch: High-Performance Sketch-based Measurement over Arbitrary Partial Key Query," in Proc. ACM SIGCOMM, 2021.
[24] R.B. Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Designing Heavy-Hitter Detection Algorithms for Programmable Switches," IEEE/ACM Transactions on Networking, vol. 28, no. 3, pp. 1172-1185, 2020.
[25] T. Yang, H. Zhang, H. Wang, M. Shahzad, X. Liu, Q. Xin, and X. Li, "FID-sketch: An Accurate Sketch to Store Frequencies in Data Streams," World Wide Web, vol. 22, no. 1, pp. 2675-2696, 2019.
[26] S. Sheng, Q. Huang, S. Wang, and Y. Bao, "PR-Sketch: Monitoring Per-key Aggregation of Streaming Data with Nearly Full Accuracy," in Proc. VLDB Endowment, 2021.
[27] Z. Liu, R.B. Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "NitroSketch: Robust and General Sketch-based Monitoring in Software Switches," in Proc. ACM SIGCOMM, 2019.
[28] Y. Zhao, K. Yang, Z. Liu, T. Yang, L. Chen, S. Liu, N. Zheng, R. Wang, H. Wu, Y. Wang, and N. Zhang, "LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets," in Proc. USENIX NSDI, 2021.
[29] R. Wang, H. Du, Z. Shen, and Z. Jia, "DAP-Sketch: An Accurate and Effective Network Measurement Sketch with Deterministic Admission Policy," Computer Networks, vol. 194, no. 1, pp. 1-13, 2021.
[30] D. Ting, "Data Sketches for Disaggregated Subset Sum and Frequent Item Estimation," in Proc. ACM SIGMOD, 2018.
[31] Q. Xiao, Z. Tang, and S. Chen, "Universal Online Sketch for Tracking Heavy Hitters and Estimating Moments of Data Streams," in Proc. IEEE INFOCOM, 2020.
[32] R.B. Basat, X. Chen, G. Einziger, R. Friedman, and Y. Kassner, "Randomized Admission Policy for Efficient Top-k, Frequency, and Volume Estimation," IEEE/ACM Transactions on Networking, vol. 27, no. 4, pp. 1432-1445, 2019.
[33] P. Ernest, "Mathematical Induction: A Pedagogical Discussion," Educational Studies in Mathematics, vol. 15, no. 1, pp. 173-189, 1984.
[34] S. Ghorbani, Z. Yang, P.B. Godfrey, Y. Ganjali, A. Firoozshahian, "DRILL: Micro Load Balancing for Low-latency Data Center Networks," in Proc. ACM SIGCOMM, 2017.
[35] M. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing," IEEE Transactions on Parallel and Distributed Systems, vol. 12, no. 10, pp. 1094-1104, 2001.
[36] Intel SSE2 Documentation. https://software.intel.com/en-us/node/683883.
[37] B. Lahiri, S. Tirthapura, and J. Chandrashekar, "Space-efficient Tracking of Persistent Items in a Massive Data Stream," Statistical Analysis and Data Mining, vol. 7, no. 1, pp. 70-92, 2014.
[38] H. Dai, M. Shahzad, M. Li, Y. Zhong, and G. Chen, "Identifying and Estimating Persistent Items in Data Streams," IEEE/ACM Transactions on Networking, vol. 26, no. 6, pp. 2429-2442, 2018.
[39] T. Yang, H. Zhang, D. Yang, Y. Huang, and X. Li, "Finding Significant Items in Data Streams," in Proc. IEEE ICDE, 2019.
[40] 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 Proc. ACM SIGMOD, 2018.
[41] H. Li, Q. Chen, Y. Zhang, T. Yang, B. Cui, "Stingy Sketch: A Sketch Framework for Accurate and Fast Frequency Estimation," in Proc. VLDB Endowment, 2022.
[42] "The CAIDA Anonymized Internet Traces," http://www.caida.org/data/overview/.
[43] L. Tang, Q. Huang and P.P.C. Lee, "SpreadSketch: Toward Invertible and Network-Wide Detection of Superspreaders," in Proc. IEEE INFOCOM, 2020.
[44] L. Tang, Y. Xiao, Q. Huang and P. P. C. Lee, "A High-Performance Invertible Sketch for Network-Wide Superspreader Detection," in IEEE/ACM Transactions on Networking, vol. 31, no. 2, pp. 724-737, 2023.
[45] T. Yang, S. Gao, Z. Sun, Y. Wang, Y. Shen, and X. Li, "Diamond Sketch: Accurate Per-flow Measurement for Big Streaming Data," IEEE Transactions on Parallel and Distributed Systems, vol. 30, no. 12, pp. 2650-2662, 2019.
[46] G. Cormode, M. Hadjieleftheriou, "Finding Frequent Items in Data Streams," in Proc. VLDB Endowment, 2008.
[47] G. Cormode, S. Muthukrishnan, "An Improved Data Stream Summary: The Count-Min Sketch and its Applications," Journal of Algorithms, vol. 55, no. 1, pp. 58-75, 2005.
[48] D. Ting, "Data Sketches for Disaggregated Subset Sum and Frequent Item Estimation," in Proc. ACM SIGMOD, 2018.
[49] A. Appleby, "MurmurHash," https://sites.google.com/site/murmurhash/
[50] T. Benson, A. Akella, and D.A. Maltz, "Network Traffic Characteristics of Data Centers in the Wild," in Proc. ACM IMC, 2010.
[51] "MAWI Working Group Traffic Archive," http://mawi.wide.ad.jp/mawi/.
[52] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units," in VLDB Endowment, 2009.
[53] Q. Huang, X. Jin, P.P.C. Lee, R. Li, L. Tang, Y.C. Chen, and G. Zhang, "SketchVisor: Robust Network Measurement for Software Packet Processing," in Proc. ACM SIGCOMM, 2017.
[54] L. Liu, Y. Shen, Y. Yan, T. Yang, M. Shahzad, B. Cui, and G. Xie, "SF-Sketch: A Two-Stage Sketch for Data Streams," IEEE Transactions on Parallel and Distributed Systems, vol. 31, no. 10, pp. 2263-2276, 2020.
[55] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Adaptive Measurements Using One Elastic Sketch," IEEE/ACM Transactions on Networking, vol. 27, no. 6, pp. 2236-2251, 2019.
[56] L. Tang, Q. Huang, and P.P.C. Lee, "A Fast and Compact Invertible Sketch for Network-Wide Heavy Flow Detection," IEEE/ACM Transactions on Networking, vol. 28, no. 5, pp. 2350-2363, 2020.
[57] Y. Zhou, Y. Zhou, M. Chen, and S. Chen, "Persistent Spread Measurement for Big Network Data Based on Register Intersection," ACM on Measurement and Analysis of Computing Systems, vol. 1, no. 15, pp. 1-29, 2017.

Weihe Li received his Masters's degree from Central South University in 2021. Currently, he is pursuing a Ph.D. degree at the University of Edinburgh. His current research focuses on developing new techniques for accurately detecting specific types of traffic flows in high-speed networks.

Paul Patras is an Associate Professor in the School of Informatics at the University of Edinburgh, where he leads the Mobile Intelligence Lab – a multi-disciplinary team that pursues research at the intersection of network engineering and artificial intelligence, to improve the analysis, resilience, and management of next generation mobile systems. He is also a co-founder and CEO of Net AI, a pioneering university spinout specializing in AI-driven network analytics. He has served on the organizing committee on several conferences and workshops in his field, and advised the ITU-T Focus Group on Machine Learning for Future Networks including 5G. Paul holds M.Sc. and Ph.D. degrees from Universidad Carlos III de Madrid (UC3M) and he was the recipient of a prestigious Chancellor's Fellowship awarded by the University of Edinburgh.