Pontus: Finding Waves in Data Streams

ABSTRACT

The bumps and dips in data streams are valuable patterns for data mining and networking scenarios such as online advertising and botnet detection. In this paper, we define the *wave*, a data stream pattern with a serious deviation from the stable arrival rate for a period of time. We then propose Pontus, an efficient framework for wave detection and estimation. In Pontus, a lightweight data structure is utilized for the preliminary processing of incoming packets in the data plane to take advantage of its high processing speed; then, the powerful control plane carries out computationally intensive wave detection and estimation. In particular, we propose the Multi-Stage Progressive Tracking strategy which detects waves in stages and removes any disqualified items promptly to save memory. Hash collisions are addressed by a Stage Variance Maximization technique to reduce estimation error. Moreover, we prove the theoretical error bound and establish upper bounds of false positive and false negative. Experiment results show that the software version of Pontus can achieve around 97% F1-Score even under scarce memory when baselines fail. Furthermore, the implemented prototype of Pontus based on P4 achieves 842 × higher throughput than the baseline strawman solution.

1 INTRODUCTION

As communication technology advances, a massive amount of data is generated and delivered to users through the high-speed Internet every second. The patterns of the data stream, such as the bumps and dips, can be exploited to provide valuable insights into data and enable many potential applications in both the networking domain and the data mining domain. However, with the overwhelming size of data, accurately recording or querying the characteristics of data streams in real-time becomes a huge challenge. To address this challenge, some flow stream processing algorithms have been proposed [9, 13, 15, 41], which give approximate answers immediately after processing the whole data stream once. These works support the queries of data stream characteristics such as frequency [9, 13, 15], quantile [21, 23, 38] and cardinality [16–18], which are significant for network management.

Recently, several works have been proposed to detect different types of burst [30, 39, 43]. Especially, BurstSketch [43] detects a data stream pattern named the *burst*, defined as a sudden increase immediately followed by a sudden decrease in terms of arrival rate, which is of great significance for network anomaly detection. However, this work lacks universality as the *burst* is only one case of abnormal changes in network traffic. Besides, BurstSketch can only detect but not estimate bursts. More importantly, its detection speed (i.e., around 1 MIPS) is far from sufficient in a real high-throughput network (e.g., 100 Gbps).

In this paper, we first propose a more general data stream pattern, namely, the *wave*. A positive wave is characterized by an increase in the frequency of a flow followed by a decrease, while a negative wave is characterized by a decrease followed by an increase. The wave differs from the burst in two aspects. First, the wave is a more comprehensive concept than the burst. In fact, a burst is a special case of a wave, which has a small increase and decrease window. Second, the wave considers both bumps and dips in data streams, while the burst only considers the former. The above properties of wave allow a much broader range of application scenarios beyond burst detection.

Scenario 1 - Botnet Detection (Network Community). Infected botnet devices are manipulated by attackers to carry out various attacks, e.g., Distributed Denial of Service (DDoS) [3, 40]. In Meris botnet [12], an infected device gradually ramps up and then decreases the number of attacking packets to bypass DDoS detection. Such an attack cannot be identified as a burst due to the absence of sudden changes but can be effectively identified by waves.

Scenario 2 - DNS Queries (Network and Data Mining Community). Domain Name System (DNS) queries can serve as key indicators of a website's service demands [14, 25]. By identifying the wave-like rises and falls in DNS queries, website administrators can improve resource utilization by dynamically adjusting resources to accommodate changes in demands. Another example is online advertising, which is estimated to be a $230 billion industry [10]. Users usually click on different sites in different time periods. For example, users tend to click on news sites at noon and social networks such as Twitter at night. As such, a higher Return on Investment can be achieved by delivering advertisements to customers on different websites upon detecting positive wave patterns in their DNS queries.

Scenario 3 - Burst Detection (Data Mining Community). Burst detection has been utilized in various applications, such as trading volumes monitoring [44], bursty topic mining [39] and text stream mining [26, 41]. As a more general concept than the burst, the wave can also be readily applied in burst detection.

Scenario 4 - Network Failure Detection (Network Community). Optical fibers are widely used on the Internet [19, 29]. However, the fiber may sometimes suffer from degradation or interruption, which is difficult to find and locate. This causes the traffic to decrease and then increase, similar to a negative wave. In this scenario, the optical fiber failure can be detected by wave identification.

Scenario 5 - Financial Market (Data Mining Community). In the financial market, a wave of trading volume may indicate possible financial fraud or illegal market manipulation. By monitoring waves, we will be able to find frequently appearing waves, which can be reported and dealt with individually.

To facilitate these potential applications, 1) wave detection should be able to process high-speed data streams with high accuracy; 2) meanwhile, for more accurate network management, we need to not only detect waves but also shape them to evaluate their types and strengths. Thus, we aim to design an efficient wave detection and estimation scheme supporting more comprehensive capability and higher speed than BurstSketch [43], which is a significant challenge.

Therefore, we propose Pontus, which detects and estimates waves accurately with high throughput in real-time. To enable high-speed preliminary item (packet) processing, we design a memory-efficient data plane, compatible with programmable switches (e.g., P4) [6]. The data plane guarantees the software version of Pontus is faster than BurstSketch. The compatibility enables the deployment of hardware programmable switches, further improving the processing speed hundreds of times. Beyond the data plane's preliminary processing of incoming data, complex but infrequent wave detection and estimation are placed on the control plane. In particular, we propose a **Multi-Stage Progressive Tracking** strategy to detect and estimate waves in Pontus, which consists of three stages that record the initial items, the weak potential waves, and the potential waves, respectively. To reduce the memory overhead, this strategy incorporates early removal of items, which removes items as soon as they are illegal during wave detection. To solve hash collisions in the control plane, we propose a **Stage Variance Maximization** technique to maintain waves with larger variances and reduce the estimation error.

To demonstrate the feasibility of Pontus, we implement the prototypes of both software (i.e., the X86 server) and hardware (i.e., the P4 switch) versions[1]. To evaluate the performance of Pontus, we conduct comprehensive experiments based on the prototypes, with the real-world traffic traces of CAIDA [8], Data Center [1], WIDE [24] and Synthetic dataset. The results confirm the high efficiency (i.e., throughput and accuracy) of Pontus.

In summary, the advantages of Pontus are four-fold. First, Pontus is expressive, which can not only detect the wave but also estimate the wave for efficient network management. Second, Pontus is memory-efficient, which requires only $O(1)$ memory, small enough to fit in CPU caches or P4 switches. Third, Pontus is fast. Our implemented software and hardware (P4 switch) versions of Pontus respectively show the 1.3 × and 572.6 × throughput compared to BurstSketch. Fourth, Pontus is accurate. Our experiments show that it achieves 99% F1-Score using only 0.5 MB memory.

2 RELATED WORK

Currently, there is no work focusing on the wave pattern. In this section, we introduce several algorithms of sketch and burst detection which are related to the wave pattern. We also discuss the opportunities provided by the programmable switches.

Sketch Algorithm. Sketch approaches are well-established for data stream measurement tasks. Classical sketch solutions include the Count-Min (CM) sketch [13] and the Count (C) sketch [9]. For example, the CM sketch consists of d arrays, each with l counters. The d arrays are associated with d pairwise independent hash functions. To insert an incoming item e, the CM sketch calculates d hash functions and inserts e to the mapping counter in each array. To query an item e, the CM sketch returns the minimum estimated frequency in the d mapping counters. The C sketch is similar to the CM sketch. The difference lies in that the C sketch uses additional d hash functions to get an unbiased estimation of an item and return the medium estimated frequency in the d mapping counters.

Burst Detection. In prior arts, detect bursts mainly focus on Wavelet Tree (WT) and Aggregation Tree (AT) [11, 32, 33], which will not be elaborated on due to limited space. Recently, several works have proposed different definitions of the burst [30, 39, 43]. [30, 39] define the burst as an item with a surge in arrival rate. [30] proposes CM-PBE, which detects bursts from history without storing or querying the whole data stream. [39] proposes TopicSketch which uses a sketch to store the velocities of an item and incrementally maintains velocities in two windows to obtain the acceleration of an item. [43] defines the burst as a sudden increase of an item in terms of arrival rate followed by a sudden decrease. [43] proposes BurstSketch, which uses a sketch to store the frequency of an item in a window and filter out infrequent items. BurstSketch maintains frequent items in a hash table. At the end of a window, BurstSketch looks up the hash table, filters out illegal items, and reports bursts.

Compared with the burst, the wave is a more comprehensive definition of the fluctuation of data streams. Therefore, tracking waves is more complicated than bursts, requiring more memory and computing power. There are four reasons why existing algorithms cannot extend to wave detection: 1) The structures in previous algorithms can not scale to track the increase or decrease trends of frequencies over multiple windows, which is required for wave detection. For example, BurstSketch only supports the comparison of an item's frequencies in two adjacent windows to determine whether the item is increasing. To track the frequency of an element in multiple windows, BurstSketch has to add additional hash tables, which unavoidably cause major changes in algorithm logic and render their theoretical proofs invalid. 2) Existing algorithms face throughput bottlenecks and are unable to support high-speed processing. For example, BurstSketch traverses all buckets in the second stage for each insertion, resulting in poor throughput. Moreover, TopicSketch and BurstSketch couple their insertion and update together, and cannot be deployed on hardware devices (e.g., programmable switches, smart network cards), hindering their potential in high-speed data stream scenarios. 3) The designs of existing algorithms have some drawbacks, affecting their performance. BurstSketch uses a fixed threshold to remove items with low frequency and records items with high frequency in the second stage, which may cause false negatives. Experiments in Section 9.7 will verify the limitations of existing solutions.

Programmable Switches. Programmable switches (e.g., Barefoot Tofino [4]) are an emerging networking technology that provides hardware programmability and flexibility without compromising performance. A representative programmable switch architecture is Protocol Independent Switch Architecture (PISA) [7], where the ASIC chip consists of a programmable parser and a number of reconfigurable match-action tables. Operators can implement custom programs in the switch using domain-specific languages (e.g., P4 [6]), allowing the switch to process data traffic at terabits per second. With a high line-rate guarantee and flexibility, programmable switches are ideal for wave detection and estimation.

3 THE WAVE

3.1 Wave Definition

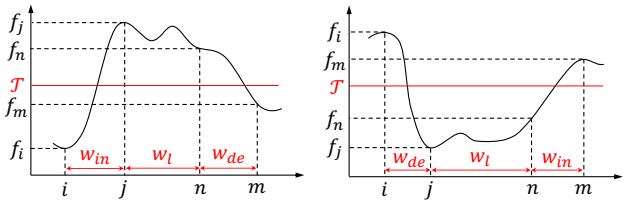
For a time series data stream $\mathcal{S} = \{e_1, e_2, ...\}$, suppose that the data stream is divided into fixed windows $\{W_1, W_2, ...\}$ and the frequencies of item e in the windows are $\{f_1, f_2, ...\}$. Consider the wave amplitude threshold $\mathcal{T}$, and the increase, steady, decrease

[1]The related codes are anonymously available at Github [2].

Notation	Meaning
S_i	Stage i, $i \in \{1, 2, 3\}$
K, V	the key and the value field in each bucket
$\mathcal{B}_{S_i}[j]$	the j^{th} bucket in Stage i
k	the wave shape threshold
$\mathcal{T}$	wave amplitude threshold to filter tiny fluctuations
T_{in}, T_l, T_{de}	increase, steady, decrease window size threshold
$\lambda_{in}, \lambda_{de}$	the weak potential pos./neg. wave threshold
W_t	the t-th window

Table 1: Important Notations.

window size thresholds T_{in}, T_l, T_{de}. Some important notations are shown in Table 1.



(a) A positive wave (b) A negative wave

Figure 1: Wave illustration. The horizontal and vertical axes are frequency and window index, respectively.

Definition 1. *For item e, a **positive wave**, as Figure 1(a) shows, is identified if there exist four non-overlapping windows W_i, W_j, W_n, W_m, $i < j < n < m$, such that the following conditions are met:*

- *Shape condition:* $f_j \geq k \cdot f_i, f_m \leq \frac{1}{k} \cdot f_n, k \in \{Z_+ | k > 1\}$.
- *Duration condition:* $j - i \leq T_{in}, n - j \leq T_l, m - n \leq T_{de}$.
- *Amplitude condition:* $f_l \geq \mathcal{T}, \forall l \in \{j, j+1, ..., n\}$.

In the positive wave, $W_{i \to j}$ is the *increase* phase, consisting of $w_{in} = j - i$ windows; $W_{j \to n}$ is the *steady* phase, consisting of $w_l = n - j$ windows; and $W_{n \to m}$ is the *decrease* phase, consisting of $w_{de} = m - n$ windows.

Definition 2. *For item e, a **negative wave**, as Figure 1(b) shows, is identified if there exist four non-overlapping windows W_i, W_j, W_n, W_m, $i < j < n < m$, such that the following conditions are met:*

- *Shape condition:* $f_j \leq \frac{1}{k} \cdot f_i, f_m \geq k \cdot f_n, k \in \{Z_+ | k > 1\}$.
- *Duration condition:* $j - i \leq T_{de}, n - j \leq T_l, m - n \leq T_{in}$.
- *Amplitude condition:* $f_i \geq \mathcal{T}, f_m \geq \mathcal{T}$.

The *decrease*, *steady* and *increase* phases of a negative wave can be defined similarly to a positive wave.

The *wave* is a more comprehensive and prevailing concept in data streams, compared with the *burst* in [43], which is a special case of a positive wave with small T_{in} and T_{de}.

3.2 Wave Detection and Estimation

Wave detection and estimation are needed for different applications.

Wave Detection. Wave detection, which detects all waves in data streams, is the fundamental task. Specifically, given an item e in data stream $\mathcal{S}$, if e is a wave, it is reported, along with its type (i.e. positive or negative), its estimated time stamp t, increase window size w_{in}, steady window size w_{de} and decrease window size w_{de}.

Wave Estimation. For each detected wave, the wave curve is estimated by its frequencies during $\{t, t+1, ..., t+(w_{in}+w_l+w_{de})\}$ windows, which is hard to realize within limited memory. Thus, we use the curve points f_i, f_j, f_n, and f_m as shown in Figure 1 to approximately estimate a wave curve instead. Note that wave estimation can be turned off if it is unnecessary for specific scenarios or if the memory is insufficient.

4 SYSTEM OVERVIEW

In this section, we present the system overview of Pontus. We first provide a strawman solution and then we propose Pontus to address the drawbacks of the strawman solution.

4.1 The Strawman Solution

To detect waves, a straightforward approach is based on multiple CM sketches. $N = T_{in} + T_l + T_{de}$ CM sketches record the frequencies of the latest N windows and a hash table stores the potential waves for wave detection. We insert each incoming item e into the CM sketch of the current window. If the frequency of e is larger than $\mathcal{T}$, we put it in the hash table. In this way, the small flows with fluctuating frequencies can be filtered to avoid meaningless wave detection. At the end of each window, we update each bucket in the hash table. We examine whether the bucket inserted in the current window is a wave by querying the latest T_{in} CM sketches. An item is set to a positive wave candidate if its estimated frequency in the current window is k times larger than the estimated frequency in any of the latest T_{in} windows, and to a negative wave candidate otherwise. For other buckets in the hash table, we query their estimated frequencies to check if any of the wave candidates fulfill the criteria of a positive or negative wave.

The strawman solution has four main drawbacks. 1) When the model parameters (e.g., T_{in}, T_l, T_{de}) change, the number of sketches in the strawman needs to be changed, necessitating the redeployment of the model. As such, the strawman has poor flexibility. 2) The throughput of the strawman is very limited since each insertion of the strawman needs to traverse the entire hash table. Meanwhile, the strawman couples its insertion and update together, and cannot be deployed on hardware devices to improve its throughput. 3) The strawman is memory-intensive because it requires $T_{in} + T_l + T_{de}$ sketches to store the frequencies. 4) The strawman stores a large amount of useless item information in each window, which cannot efficiently utilize the memory, resulting in excessive hash collisions and poor performance.

4.2 The Pontus Framework

To address the limitations of our strawman solution, we should first reduce the memory overhead. We observe that most items do not have the wave pattern most of the time. Therefore, in Pontus, we first only record the item's frequencies of the current and previous windows. Based on the comparison between them, we then sift out the items with fluctuating and start to record their frequencies

during the full wave cycle (i.e., $T_{in} + T_l + T_{de}$ windows). In this way, the excessive amount of redundant information maintained in the strawman solution is avoided, thereby saving memory space.

To this end, we propose a **Multi-Stage Progressive Tracking** strategy in Pontus, which consists of three stages. The **Multi-Stage Progressive Tracking** is from the core idea of "filtering", where we filter out useless items, i.e., non-fluctuating items, as early as possible, and only maintain and track useful items (fluctuating items) to save memory and improve throughput. We classify the items in the data stream into three types: 1) useless items, which violate wave conditions already; 2) weak potential waves, items that have started to show the increase or decrease trends but have not satisfied the left shape condition, that is, the change of frequency is not strong enough to meet the left shape condition yet ($f_j \geq k \cdot f_i$ or $f_j \leq \frac{1}{k} \cdot f_i$). These waves can be regarded as a possible start of the left shape of a true wave. A *weak potential wave* is identified by the thresholds λ_{in} for a positive wave, i.e., $f_j \geq \lambda_{in} \cdot f_i$, and λ_{de} for a negative wave, i.e., $f_j \leq \lambda_{de} \cdot f_i$, where $1 \leq \lambda_{in} < k$ and $1/k < \lambda_{de} \leq 1$ (e.g., $\lambda_{in} = 1.3, \lambda_{de} = 0.8$); 3) potential waves that have already satisfied the left shape condition.

Particularly, the *Stage 1* (initial stage) is responsible for recording the frequencies of current and previous windows for the useless items. The *Stage 2* (weak potential stage) and the *Stage 3* (potential stage) are responsible for recording the frequencies of the weak potential waves and potential waves, respectively. The *Stages 1* filters out useless items and sends weak potential waves to the *Stages 2* and potential waves to the *Stages 3*. The *Stages 2* filters out illegal weak potential waves among the recorded weak potential waves and sends potential waves to the *Stages 3*. The *Stages 3* filters out illegal potential waves and reports waves. The items in the *Stages 2,3* are removed immediately to save memory if they are identified as illegal ones, e.g., one of the wave conditions is obviously violated. The *Stages 1* uses approximate count. The *Stages 2,3* use exact count. The **Multi-Stage Progressive Tracking** strategy can significantly save memory overhead. As the space complexity of Pontus is $O(1) << O(N)$, hash collisions are inevitable in Pontus. Though Pontus cannot achieve absolute correctness, we prove the upper bounds on the probability of false positives and false negatives for Pontus in Section 8 and experiments in Section 9.2 further verify that these error bounds are guaranteed to be small with properly selected parameters.

Based on the **Multi-Stage Progressive Tracking** strategy, we further decouple the item-by-item frequency recording of the current window and the window-by-window wave detection or estimation (hundreds of thousands of times less frequent than the former) by separating Pontus into the data and control planes. The **data plane** is only responsible for simply recording item frequencies in the current window. The **control plane** is responsible for recording all the other information and conducting more complex operations of the three stages, including identifying waves, removing illegal ones, and moving items between stages (e.g., $S_1 \rightarrow S_2$, $S_1 \rightarrow S_3$ and $S_2 \rightarrow S_3$). In this way, the data plane structure is simplified and shrunk to an extremely small size, which can be cached by CPU multi-level caches (X86-based software version) or deployed to programmable switches (P4-based hardware version) for an acceleration of the highly frequent item-by-item recording.

Data plane: The data plane maintains minimal information required to support the control plane in its three-stage structure. During each window, the data plane processes incoming data with high speed and records relevant frequency information into corresponding stages. To solve the hash collisions in the data plane, we use a probability replacement strategy. At the end of each window, the data plane sends the data collected in the current window to the control plane. After the control plane's further processing of the data, the data plane updates its stages according to the latest information sent back from the control plane.

Control plane: At the end of each window, the control plane processes the latest information received from the data plane following the Multi-Stage Progressive Tracking strategy and reports any detected waves. To solve hash collisions in the control plane, we use the **Stage Variance Maximization** technique to maintain waves with larger variances and reduce the estimation error. The purpose of **Stage Variance Maximization** is that the waves with large fluctuations are strong indicators for anomalies of the items in the data streams. After updating, the control plane sends the latest information back to the data plane.

Pontus can detect and estimate waves accurately with limited memory in real-time. The details of the data and control planes of Pontus are presented next.

5 THE PONTUS DATA PLANE

The data plane consists of three stages, *Stages 1-3* ($S_{1,2,3}$). S_i consists of l_i buckets. Each bucket stores a K-V pair, where the item ID and its estimated frequency are stored as the key and the value, respectively. There are d hash functions $h_1(\cdot), h_2(\cdot), ..., h_d(\cdot)$ associated with S_1. The same set of $\hat{d}$ hash functions, i.e., $g_1(\cdot), g_2(\cdot), ..., g_{\hat{d}}(\cdot)$ are associated with both S_2 and S_3 to save memory overhead. The data plane is responsible for **insertion** and **update**.

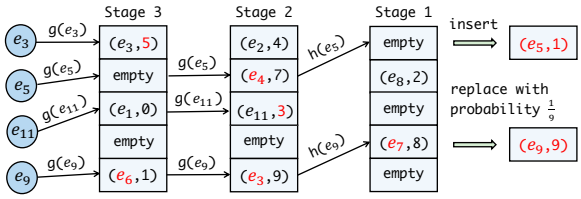
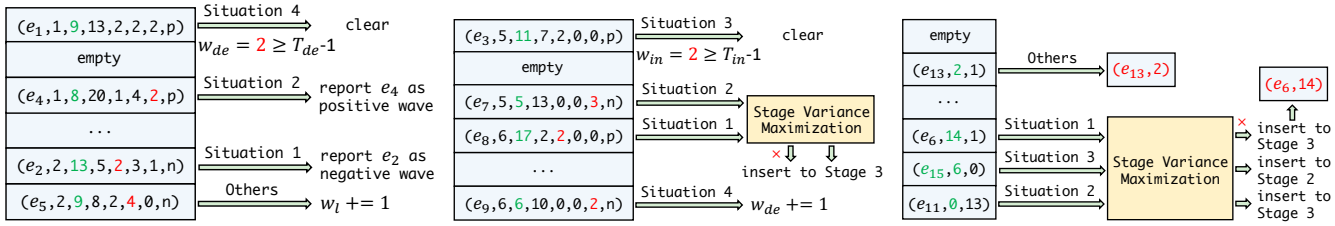


Figure 2: The Pontus data plane.

Insertion: Each incoming item is inserted into the data plane. The pseudo-code of insertion in the data plane is shown in Algorithm 1. Given an incoming item e with frequency f, we first hash e into $\hat{d}$ mapping buckets $\mathcal{B}_{S_3}[g_1(e), ..., g_{\hat{d}}(e)]$ to check whether it is in S_3. If e is in S_3, we increment its estimated frequency by f. Otherwise, we examine $\hat{d}$ mapping buckets $\mathcal{B}_{S_2}[g_1(e), ..., g_{\hat{d}}(e)]$ to check whether it is in S_2. If e is in S_2, we increment its estimated frequency by f. Otherwise, we insert it into S_1 by hashing e into d mapping buckets $\mathcal{B}_{S_1}[h_1(e), ..., h_d(e)]$, and consider the following three cases.

Case 1: $e \in S_1$. We increment its estimated frequency by f.

Case 2: $e \notin S_1$, and there exists an empty bucket $\mathcal{B}_{S_1}[h_i(e)]$ in the d mapping buckets. In this case, we insert e in $\mathcal{B}_{S_1}[h_i(e)]$ by setting $K = e$ and $V = f$.



(a) S_3 (b) S_2 (c) S_1

Figure 3: The Pontus control plane (the data received from the data plane is highlighted in green).

Algorithm 1: Insertion in the data plane

Input: item e; frequency f;
if ***Insert_MidStage***(S_3, e, f) **then**
  return ;
if ***Insert_MidStage***(S_2, e, f) **then**
  return ;
Insert_FirstStage(e, f) ;
return ;
Function *Insert_MidStage*(S_j, e, f)**:**
  for $i \in [1, \hat{d}]$ **do**
    if *e is in* $\mathcal{B}_{S_j}[g_i(e)]$ **then**
      $\mathcal{B}_{S_j}[g_i(e)].V += f$;
      return **true** ;
  return **false** ;
Function *Insert_FirstStage*(e, f)**:**
  for *each* $i \in [1, d]$ **do**
    if *e is in* $\mathcal{B}_{S_1}[h_i(e)]$ **then**
      $\mathcal{B}_{S_1}[h_i(e)].V += f$;
      return ;
  if *there exists empty bucket* $\mathcal{B}_{S_1}[h_i(e)]$ **then**
    insert e into $\mathcal{B}_{S_1}[h_i(e)]$;
    $\mathcal{B}_{S_1}[h_i(e)].V += f$;
    return ;
  // Hash Collision
  use e to replace the smallest bucket $\mathcal{B}_{S_1}[h_i(e)]$ and $\mathcal{B}_{S_1}[h_i(e)].V += f$ w.p. $\frac{f}{\mathcal{B}_{S_1}[h_i(e)].V+f}$;
  return ;

Case 3: e is not in S_1 and there is no empty bucket in the d mapping buckets. In this case, we try to replace the smallest bucket $\mathcal{B}_{S_1}[h_i(e)]$ among the d mapping buckets. To solve the hash collisions, there are several strategies: probabilistic replacement [5, 42], probabilistic decay [20] and frequency decay [43]. In Pontus, the probabilistic replacement is adopted to solve the hash collisions in S_1, since it achieves the best performance, as will be shown in the experiments in Section 9.5. In particular, we replace the key K by e and increment the value V by f with the probability of $\frac{f}{V+f}$. Note that our probabilistic replacement is different from [5, 42]. [42] increments V by f while replacing K by e with a probability of $\frac{f}{V}$. [5] finds the smallest bucket b among all buckets in S_1. Then, the key K of b is replaced by e and the value V is incremented by f with a probability of $\frac{f}{b.V+f}$.

Example: Figure 2 shows an example of insertion. We set $d = 1, \hat{d} = 1, \mathcal{T} = 10, k = 2$ and item frequency $f = 1$ for simplicity. 1) To insert e_3, we use hash function $g(\cdot)$ to map it to bucket $(e_3, 4)$ in S_3 and increment V by 1 (from 4 to 5). 2) To insert e_9, we map it to bucket $(e_6, 1), (e_3, 9)$ in S_3, S_2, sequentially. Both of the K are not equal to e_9. Then we map it to bucket $(e_7, 8)$ in S_1. The K is also not equal to e_9. We replace this bucket to $(e_9, 9)$ with probability $\frac{1}{9}$.

Update: At the end of each window, the data plane sends the estimated frequencies in S_2, S_3 and K-V pairs in S_1 to the control plane. The control plane updates its stages, which will be elaborated in Section 6, and sends update information back to the data plane. The data plane updates its buckets accordingly as follows. In stage S_1, all buckets are cleared. In S_2 and S_3, the keys are updated according to the latest information received from the data plane while the values are reset to 0 for all buckets. For example, if we update e_3 and insert it from S_2 to S_3 in the control plane, then we do the same operations in the data plane.

6 THE PONTUS CONTROL PLANE

The control plane uses a **Multi-Stage Progressive Tracking** strategy consisting of three stages $S_{1,2,3}$ to filter the illegal wave patterns as early as possible to save memory. S_i consists of l_i buckets. S_1 (initial stage) is responsible for recording the frequencies of current and previous windows for the initial items. In S_1, each bucket stores a K-V pair. S_2 (weak potential stage) stores weak potential positive and negative wave patterns that are in the increase or decrease phase. If the weak potential positive or negative wave pattern in S_2 starts to decrease or increase, then it becomes illegal and is evicted immediately. S_3 (potential stage) is responsible for recording the frequencies of the potential waves. Each bucket in S_2 and S_3 stores $(K, t, V, w_{in}, w_l, w_{de}, type)$ where t is current window timestamp and $type$ is the potential type of wave.

At the end of each window, the control plane receives the estimated frequencies in S_2, S_3 and K-V pairs in S_1 from the data plane. The control plane updates each bucket correspondingly to filter out illegal items and report waves. After updating, the control plane sends update information back to the data plane. We define four update situations: $Situation1$: $V_d \geq k \cdot V_c \wedge V_d \geq \mathcal{T}$, indicating that an item increases k times. $Situation2$: $V_d \leq 1/k \cdot V_c \wedge V_c \geq \mathcal{T}$,

Algorithm 2: Update S_3

```
for each bucket i ∈ B_{S_3} ∧ B_{S_3}[i] is not empty do
    if Situation1(V_d, V_c) then
        if B_{S_3}[i].type == n then
            B_{S_3}[i].w_in+ = 1 ;
            Report e as negative wave ;
        clear B_{S_3}[i] to empty ;
    else if Situation2(V_d, V_c) then
        if B_{S_3}[i].type == p then
            B_{S_3}[i].w_de+ = 1 ;
            Report e as positive wave ;
        clear B_{S_3}[i] to empty ;
    else if Situation3(V_d, V_c) then
        if B_{S_3}[i].type == n ∧ B_{S_3}[i].w_in < T_in − 1 then
            B_{S_3}[i].w_in+ = 1 ;
        else
            clear B_{S_3}[i] to empty ;
    else if Situation4(V_d, V_c) then
        if B_{S_3}[i].type == p ∧ B_{S_3}[i].w_de < T_de − 1 then
            B_{S_3}[i].w_de+ = 1 ;
        else
            clear B_{S_3}[i] to empty ;
    else
        if B_{S_3}[i].w_l < T_l then
            t_l+ = 1 ;
        else
            clear B_{S_3}[i] to empty ;
return ;
```

Algorithm 3: Update S_2

```
for each bucket i ∈ B_{S_2} ∧ B_{S_2}[i] is not empty do
    if Situation1(V_d, V_c) then
        if B_{S_2}[i].type == p then
            B_{S_2}[i].w_in+ = 1, B_{S_2}[i].V = V_d ;
            insert B_{S_2}[i] into S_3 ;
        clear B_{S_2}[i] to empty ;
    else if Situation2(V_d, V_c) then
        if B_{S_2}[i].type == n then
            B_{S_2}[i].w_de+ = 1, B_{S_2}[i].V = V_d ;
            insert B_{S_2}[i] into S_3 ;
        clear B_{S_2}[i] to empty ;
    else if Situation3(V_d, V_c) then
        if B_{S_2}[i].type == p ∧ B_{S_2}[i].w_in < T_in − 1 then
            B_{S_2}[i].w_in+ = 1 ;
        else
            clear B_{S_2}[i] to empty ;
    else if Situation4(V_d, V_c) then
        if B_{S_2}[i].type == n ∧ B_{S_2}[i].w_de < T_de − 1 then
            B_{S_2}[i].w_de+ = 1 ;
        else
            clear B_{S_2}[i] to empty ;
    else
        clear B_{S_2}[i] to empty ;
return ;
```

indicating that an item decreases k times. *Situation*3: $V_d > \lambda_{in} \cdot V_c \wedge V_d \geq \mathcal{T}/T_{in}$, indicating that an item starts to increase. *Situation*4: $V_d < \lambda_{de} \cdot V_c \wedge V_c \geq \mathcal{T}$, indicating that an item starts to decrease.

*Situation*1, 2 indicate a strong increase or decrease that satisfies left shape conditions in the wave definition. *Situation*3, 4 indicate a weak increase or decrease that is not strong enough to satisfy the left shape condition but still worth further monitoring. $\lambda_{in}, \lambda_{de}$ indicate the threshold that we further monitor the weak potential positive or negative wave. Note that V_d is the estimated frequency collected by the data plane in the current window and V_c is the estimated frequency in the previous window stored in the control plane.

Update: We update the control plane in the order of S_3, S_2 , and S_1 so as to reduce the hash collisions when the item updates. The current window is set to W_t. Non-empty buckets in the three stages are updated as follows. Pseudo-codes of update S_3, S_2, S_1 are shown in Algorithm 2,3,4, respectively.

In S_3, if a bucket b matches 1) *Situation*1: $w_{in}+ = 1$, report K as a negative wave if $type = n$, and clear b; 2) *Situation*2: $w_{de}+ = 1$, report K as a positive wave if $type = p$, and clear b; 3) *Situation*3: $w_{in}+ = 1$ if $type = n$ and $w_{in} < T_{in} - 1$, which implies the increase phase of a potential negative wave, otherwise, an illegal wave is identified by clearing b; 4) *Situation*4: $w_{de}+ = 1$ if $type = p$ and $w_{de} < T_{de} - 1$, which implies the decrease phase of a potential positive wave, otherwise, an illegal wave is identified by clearing b; 5) *other situations*: $w_l+ = 1$ if $w_l < T_l$, otherwise clear b.

In S_2, if a bucket b matches 1) *Situation*1: $w_{in}+ = 1, V = V_d$, insert b into S_3 if $type = p$, which implies an existing weak potential positive wave experiences a strong enough increase so that it can be promoted to S_3 as a potential positive wave, and clear b; 2) *Situation*2: $w_{de}+ = 1, V = V_d$, insert b into S_3 if $type = n$, which implies an existing weak potential negative wave experiences a strong enough decrease so that it can be promoted to S_3 as a potential negative wave, and clear b; 3) *Situation*3: $w_{in}+ = 1$ if $type = p$ and $w_{in} < T_{in} - 1$, meaning it remains as a weak potential positive wave, otherwise, an illegal wave is identified, and clear b; 4) *Situation*4: $w_{de}+ = 1$ if $type = n$ and $w_{de} < T_{de} - 1$, implying a weak potential negative wave remains in its status, otherwise, an illegal wave is identified by clearing b; 5) *other Situations*: an illegal item is identified by clearing b.

In S_1, set $V_c = 0$ if a bucket b is not in the control plane and set $V_d = 0$ if b is not in the data plane. We regard the above buckets as special buckets in S_1. If b matches 1) *Situation*1: an increase experienced which is strong enough to directly identify the increase phase of a potential positive wave, set $t = W_t, w_{in} = 1, V = V_d, type = p$ in a temporary bucket tmp and insert tmp into S_3; 2) *Situation*2: a

Algorithm 4: Update S_1

```
for each bucket i ∈ B_{S_1} ∧ B_{S_1}.[i] is not empty do
    set e = B_{S_1}[i].K ;
    if Situation1(V_d, V_c) then
        insert b = (e, W_t, V_d, 1, 0, 0, p) into S_3 ;
    else if Situation2(V_d, V_c) then
        insert b = (e, W_t, V_d, 0, 0, 1, n) into S_3 ;
    else if Situation3(V_d, V_c) then
        insert b = (e, W_t, V_d, 1, 0, 0, p) into S_2 ;
    else if Situation4(V_d, V_c) then
        insert b = (e, W_t, V_d, 0, 0, 1, n) into S_2 ;
    else
        B_{S_1}[i].V = V_d ;
    if match one situation then
        if fail to insert then
            B_{S_1}[i].V = V_d ;
        else
            clear B_{S_1}[i] to empty ;
return ;
```

decrease experienced which is strong enough to directly identify the decrease phase of a potential negative wave, set $t = W_t, w_{de} = 1, V = V_d, type = n$ in tmp and insert tmp into S_3; 3) $Situation3$: a weak increase experienced which is not strong enough to justify a potential positive wave, so we set $t = W_t, w_{in} = 1, V = V_c, type = p$ in tmp and insert tmp into S_2 as a weak potential positive wave for further monitoring. 4) $Situation4$: a weak decrease experienced which is not strong enough to justify a potential negative wave, so we set $t = W_t, w_{de} = 1, V = V_c, type = n$ in tmp and insert tmp into S_2 as a weak potential negative wave for further monitoring. If b matches any of the above situations and the insertion is successful, clear b. Otherwise, clear b if $V_d = 0$, or set $V = V_d$ if $V_d \neq 0$.

Example: Figure 3 shows an example of control plane update. Estimated frequencies in S_2, S_3 and K-V pairs in S_1 collected by the data plane are highlighted in green. We set $T_{in} = T_{de} = 3, T_l = 10, d = 1, \hat{d} = 1, \mathcal{T} = 10, k = 2, W_t = 8, \lambda_{in} = (k-1)/T_{in} + 1$ and $\lambda_{de} = 1 - (1 - 1/k)/T_{de}$. For S_3, to update e_1, since it matches $Situation4$, we check $w_{de} = 2 \geq T_{de} - 1$, thus e_1 is an illegal wave and we clear it to empty; to update e_4, since it matches $Situation2$, we do $w_{de}+ = 1$ (from 1 to 2) and report it as a positive wave. For S_2, to update e_7, since it matches $Situation2$, we do $w_{de}+ = 1$. We successfully insert it into S_3 using Stage Variance Minimization. For S_1, to update e_{15}, since it matches $Situation3$, we use a temporary bucket tmp and set $K = e_{15}, t = 8, V = 6, w_{in} = 1, type = p$. We successfully insert tmp into S_2 using Stage Variance Minimization. Then we clear the K-V pair in S_1 to empty.

Wave estimation: To estimate the curve points of a wave, we add two additional counters V_{st} and V_{vr} for each bucket in S_2 and S_3. V_{st} indicates f_i which is the start curve point of a wave. V_{vr} indicates $|f_j - f_i|$, which is the variance of a wave. It is worth noting that, when updating a bucket b in S_1, a temporary bucket is inserted to either S_2 or S_3 depending on its matching situation, we set $V_{st} = V_c$ (i.e. f_i) and $V_{vr} = |V_d - V_c|$ (i.e. $|f_j - f_i|$) in the corresponding bucket in S_2 or S_3 as part of the insertion operation. When updating a bucket b in S_2, under $Situation1$ and $Situation2$, we set $V_{vr} = V_d - V_c$, $V_{st} = V_c$ in the corresponding bucket in S_3 when b is inserted to S_3; under $Situation3$ and $Situation4$, we update $V_{vr} = V_d - V_c$ in bucket b itself. In S_3, when a wave is detected, $V_{st}, V_{st} + V_{vr}, V_c, V_d$ are reported as the estimated curve points f_i, f_j, f_n, f_m along with the wave.

Insertion: As discussed above, insertion may be triggered by moving from one stage to another. We hash its key K into $\hat{d}$ mapping buckets $\mathcal{B}_{s_i}[g_1(K), ..., g_{\hat{d}}(K)]$. Then, we consider two cases.

Case 1: there exists an empty bucket $\mathcal{B}_{S_i}[g_j(K)]$ in $\hat{d}$ mapping buckets, we insert b in $\mathcal{B}_{S_i}[g_j(K)]$.

Case 2: there is no empty bucket. When wave estimation is disabled, i.e., the counters V_{st} and V_{vr} are not available, we randomly select one bucket to replace with b. When wave estimation is enabled, we propose a **Stage Variance Maximization** technique to solve hash collisions. In Stage Variance Maximization, we select one bucket $\mathcal{B}_{S_i}[g_j(K)]$ with the smallest V_{vr} among $\hat{d}$ mapping buckets. If the $b.V_{vr} > \mathcal{B}_{S_i}[g_j(K)].V_{vr}$, we replace it with b. Our experiments show that Stage Variance Maximization reduces estimation error.

Compared with the strawman solution, the advantages of Pontus using Multi-Stage Progressive Tracking are two-fold. First, Pontus only requires $O(1)$ memory to detect waves, while strawman requires $O(T_{in} + T_l + T_{de})$. Second, the throughput of Pontus is only related to the number of hash functions (d+$\hat{d}$), while the throughput of strawman is associated with the number of hash functions in CM sketches and the number of buckets in the hash table. Therefore, Pontus can achieve higher throughput than the strawman.

7 OPTIMIZATION

We implement the data plane of Pontus in programmable switches to improve its throughput. With the Stateful Algorithm and Logical Unit (Stateful ALU) in each stage of the switch pipeline, we look up and update the entries in the corresponding register array. There are three differences between the P4 hardware and software versions of Pontus. 1) Due to the resource limitation of Stateful ALUs, we only store the key and value fields in physical registers. For insertion, we need to go back to the register that has the smallest value and reset the key and value when there are hash collisions in S_1. However, this process is not allowed in the P4 language. To solve this problem, we use the resubmit primitive. When a packet is resubmitted to the beginning of the pipeline, it can maintain up to eight bytes of metadata in the resubmit header. We use the resubmit metadata to record the necessary information (e.g., the counted packet number) for the replacement operation. 2) As multiplications, divisions, and floating-point operations are not supported in P4, it is difficult to calculate the probability. In P4, to approximate probabilistic replacement, we generate a 32-bit random number r and replace the smallest bucket in S_1 if $(r << L_V) < 2^{32}$, in which L_V represents the bit furthest to the left in V. 3) The control plane is deployed on the local CPU of the programmable switch or a remote server. At the end of each window, the programmable data plane sends values in registers of different stages to the control plane. After updating, the control plane sends the update information back to the data plane by rewriting physical registers in the switch pipeline.

8 MATHEMATICAL ANALYSIS

In this section, we formally establish the error bound of S_1 and the upper bounds of false positive and false negative.

8.1 Error Bound of S_1

Lemma 8.1. *Given a data stream $\mathcal{S}$, which obeys an arbitrary distribution, assume that S_1 has d hash functions and m is the value field of the minimum bucket among d mapping buckets. Let $\hat{f}$ and f be the estimated and actual frequency of an item e, respectively. Then, we have $\hat{f} \leq f + m$.*

Proof. Similarly as in [5], at time t in current window: Case 1: e is not in S_1. In this case, $\hat{f} = 0$ and the claim obviously holds.

Case 2: e is in S_1. In this case, consider the last time l (l and t are both in the current window), e is admitted in S_1. So we have $V_e^l = m_{l-1} + 1$, where V_e^l is the value field (V) of e's bucket at time l and m_{l-1} is the minimum V among d mapping buckets at time $l-1$ in S_1. Note that we only increase the minimum V among d mapping buckets due to an item insertion. At that point, either no bucket changes or the minimum V is incremented. Hence, $V_e^l \leq m_{l-1} + 1$. Suppose that e arrives n times between time l and time t. $n \leq f - 1$ since e arrived once at time $l-1$. Therefore, it follows that $\hat{f} = V_e^t = V_e^l + n \leq m_{l-1} + 1 + f - 1 = f + m$. □

Theorem 8.1. *Assume that each window is fixed in $\mathcal{S}$ and has N items. Let w be the number of buckets in S_1 and ϵ be a small positive number, we have $\mathcal{P}(\hat{f} - f \geq \epsilon N) \leq \frac{1}{\epsilon\omega}$.*

Proof. Suppose that d hash functions are uniformly distributed, according to Lemma 8.1, we have $E(\hat{f} - f) \leq m \leq \frac{N}{\omega}$. By Markov inequality, we have $\mathcal{P}(\hat{f} - f \geq \epsilon N) \leq \frac{E(\hat{f}-f)}{\epsilon N} \leq \frac{m}{\epsilon N} \leq \frac{1}{\epsilon\omega}$. □

8.2 Upper Bounds of False Positives

Lemma 8.2. *Let N be the window size, $\omega_1, \omega_2, \omega_3$ be the numbers of buckets in S_1, S_2, S_3, respectively. d is the number of hash functions in S_1. $\hat{d}$ is the number of hash functions in S_2, S_3. The probability of hash collisions in S_1, S_2, S_3 are given as*

$$\mathcal{P}_{S_1} \leq \left\{1 - (\frac{N}{\omega_1} + 1)e^{\frac{N}{\omega_1}}\right\}^d, \tag{1}$$

$$\mathcal{P}_{S_2} \leq \left\{1 - \left[\frac{N\,(T_{in}+1)}{\mathcal{T}\omega_2} + 1\right] e^{-\frac{N(T_{in}+1)}{\mathcal{T}\omega_2}}\right\}^{\hat{d}}, \tag{2}$$

$$\mathcal{P}_{S_3} \leq \left[1 - \left(\frac{2N}{\mathcal{T}\omega_3} + 1\right) e^{-\frac{2N}{\mathcal{T}\omega_3}}\right]^{\hat{d}}. \tag{3}$$

Proof. S_1 inserts a weak potential positive wave into S_2 if it matches *Situation*3. So there are at most $\frac{NT_{in}}{\mathcal{T}}$ weak potential positive waves in a window. Similarly, S_1 inserts a weak potential negative wave into S_2 if it matches *Situation*4. There are at most $\frac{N}{\mathcal{T}}$ weak potential negative waves. Totally, there are at most $\frac{N(T_{in}+1)}{\mathcal{T}}$ weak potential waves. Each weak potential wave is randomly mapped to a bucket in S_2 by hash functions. The probability that a weak potential wave is mapped to an arbitrary bucket is $\frac{1}{\omega_2}$. Therefore, for any bucket, the number of weak potential waves mapped to bucket Y follows a Binomial distribution $B\left(\frac{N(T_{in}+1)}{\mathcal{T}}, \frac{1}{\omega_2}\right)$. Usually, $\frac{N(T_{in}+1)}{\mathcal{T}}$ is large and $\frac{1}{\omega_2}$ is a small probability. Thus, Y approximately follows a Poisson distribution $\pi\left(\frac{N(T_{in}+1)}{\mathcal{T}\omega_2}\right)$, that is,

$$\mathcal{P}(Y = i) = \frac{\left(\frac{N(T_{in}+1)}{\mathcal{T}\omega_2}\right)^i}{i!} e^{-\frac{N(T_{in}+1)}{\mathcal{T}\omega_2}}. \tag{4}$$

Hash collisions happen when two or more weak potential waves are mapped to one bucket. The probability of hash collision can be given as:

$$\begin{aligned}\mathcal{P}(Y \geq 2) &= 1 - \mathcal{P}(Y = 0) - \mathcal{P}(Y = 1) \\ &\leq 1 - \left[\frac{N\,(T_{in}+1)}{\mathcal{T}\omega_2} + 1\right] e^{-\frac{N(T_{in}+1)}{\mathcal{T}\omega_2}}.\end{aligned}$$

Since we use $\hat{d}$ hash functions in S_2, we have

$$\mathcal{P}_{S_2} \leq \left\{1 - \left[\frac{N\,(T_{in}+1)}{\mathcal{T}\omega_2} + 1\right] e^{-\frac{N(T_{in}+1)}{\mathcal{T}\omega_2}}\right\}^{\hat{d}}. \tag{5}$$

□

Similarly, we can prove $\mathcal{P}_{S_1}$ and $\mathcal{P}_{S_3}$. We omit the detailed proofs here due to limited space.

Lemma 8.3. *Suppose that X follows a Poisson distribution with parameter λ. When λ is large (i.e., $\lambda \geq 20$), its cumulative distribution function approximately obeys*

$$P(X \leq x) = \sum_{i=0}^{x} p(X = i) \approx \left(1 + e^{\frac{\lambda - x}{\sqrt{\lambda}}}\right)^{-1}. \tag{6}$$

According to Equation (6)*, we have*

$$P(a < X \leq b) = P(X \leq b) - P(X \leq a) \leq 1 - e^{-\frac{b-a}{\sqrt{\lambda}}}, \tag{7}$$

$$P(a < X \leq b) \geq \frac{e^{\sqrt{\lambda}}}{(1 + e^{\sqrt{\lambda}})^2}(e^{-\frac{a}{\sqrt{\lambda}}} - e^{-\frac{b}{\sqrt{\lambda}}}). \tag{8}$$

Theorem 8.2. *Let N be the window size. The frequency f_c of item e in the current window follows a Poisson distribution [45] with parameter λ. The estimated frequency $\hat{f}_p$ of item e in the previous window is n. Items are independent and we cut off the tail of the distribution. The upper bounds of the false positives of the positive wave P_{FPP} and negative wave P_{FPN} are*

$$P_{FPP} \leq \{(1 - e^{-\frac{k \cdot n + 1}{\sqrt{\lambda}}}) + (1 - e^{-\frac{\lambda_{in} \cdot n + 1}{\sqrt{\lambda}}}) \tag{9}$$

$$\cdot (1 - \left[\frac{e^{\sqrt{\lambda}}}{(1 + e^{\sqrt{\lambda}})^2}(e^{-\frac{\lambda_{in} \cdot n}{\sqrt{\lambda}}} - e^{-\frac{k \cdot n}{\sqrt{\lambda}}})\right]^{T_{in}-1})\} \cdot \frac{\mathcal{P}_{S_1}}{d}, \tag{10}$$

$$P_{FPN} \leq \{(1 - e^{-\frac{n/k + 1}{\sqrt{\lambda}}}) + (1 - e^{-\frac{(\lambda_{de}{}^{-1/k}) \cdot n + 1}{\sqrt{\lambda}}}) \tag{11}$$

$$\cdot (1 - \left[\frac{e^{\sqrt{\lambda}}}{(1 + e^{\sqrt{\lambda}})^2}(e^{-\frac{n/k}{\sqrt{\lambda}}} - e^{-\frac{\lambda_{de} \cdot n}{\sqrt{\lambda}}})\right]^{T_{de}-1})\} \cdot \frac{\mathcal{P}_{S_1}}{d}. \tag{12}$$

Proof. We first prove the upper bound of the false positive of the positive wave. Here, we only consider the part where Pontus causes errors. The approximate count we use in S_1 causes errors. Note that for P_{FPP}, we consider $\hat{f}_p = f_p$, since $\hat{f}_p > f_p$ leads to smaller P_{FPP}. For P_{FPN}, we consider $\hat{f}_c = f_c$, since $\hat{f}_c > f_c$ leads

to smaller P_{FPN} where $\hat{f_c}$ is the estimated frequency of item e in the current window. In S_2 and S_3, we use exact counts, which do not cause errors. Thus we consider the probability that an item is mistakenly inserted into S_3 (i.e., through $S_1 \to S_2 \to S_3$ or $S_1 \to S_3$) due to hash collisions. So, P_{FPP} satisfies

$$P_{FPP} \le P\{S_1 \to S_2 \to S_3\} + P\{S_1 \to S_3\}, \tag{13}$$

where $P\{S_1 \to S_2 \to S_3\}$ ($P_{1,2,3}$ for short) is the probability that an item is mistakenly inserted into S_3 via S_2 and $P\{S_1 \to S_3\}$ ($P_{1,3}$ for short) is the probability that an item is mistakenly inserted into S_3 directly. $P_{1,2,3}$ is calculated by $P\{S_1 \to S_2\} \cdot P\{S_2 \to S_3\}$ ($P_{1,2}, P_{2,3}$ for short, respectively), so we have $P_{1,2,3} = P_{1,2} \cdot P_{2,3}$.

First, consider $P_{1,2}$, we have

$$P_{1,2} = P\{f_c < \lambda_{in} \cdot f_p \wedge \hat{f_c} \ge \lambda_{in} \cdot f_p | f_p = n\} \tag{14}$$

$$= \sum_{i=1}^{\lceil \lambda_{in} \cdot n \rceil} P(f_c = \lfloor \lambda_{in} \cdot n \rfloor - i) \cdot P(\hat{f_c} - f_c \ge i). \tag{15}$$

According to Lemma 8.3, we have

$$P_{1,2} \le (1 - e^{-\frac{\lambda_{in} \cdot n+1}{\sqrt{\lambda}}}) \cdot P(\hat{f_c} - f_c \ge 1). \tag{16}$$

Note that necessary conditions of $P(\hat{f_c} - f_c \ge 1)$ are that a hash collision occurs in S_1 and the bucket of the hash collision happens to be the bucket recorded f_c. Therefore, we have

$$P_{1,2} \le \left(1 - e^{-\frac{\lambda_{in} \cdot n+1}{\sqrt{\lambda}}}\right) \cdot \frac{\mathcal{P}_{S_1}}{d}. \tag{17}$$

Let P_x and P_y represent $P(\lambda_{in} \cdot f_P \le f_c \le k \cdot f_p)$ and $P(f_c \ge k \cdot f_p)$ for short, respectively. Note that $P_x + P_y \le 1$. So $P_{2,3}$ is

$$P_{2,3} = \sum_{i=0}^{T_{in}-1} P_x^{i-1} \cdot P_y = \frac{1 - P_x^{T_{in}-1}}{1 - P_x} \cdot P_y \le \left(1 - P_x^{T_{in}-1}\right). \tag{18}$$

According to Lemma 8.3, we have

$$P_{2,3} \le 1 - \left[\frac{e^{\sqrt{\lambda}}}{\left(1 + e^{\sqrt{\lambda}}\right)^2}\left(e^{-\frac{\lambda_{in} \cdot n}{\sqrt{\lambda}}} - e^{-\frac{k \cdot n}{\sqrt{\lambda}}}\right)\right]^{T_{in}-1}. \tag{19}$$

Similar with $P_{1,2}$, for $P_{1,3}$, we have

$$P_{1,3} \le \left(1 - e^{-\frac{k \cdot n+1}{\sqrt{\lambda}}}\right) \cdot \frac{\mathcal{P}_{S_1}}{d}. \tag{20}$$

Combine Equation (17), (19), and (20), we have

$$P_{FPP} \le \{(1 - e^{-\frac{k \cdot n+1}{\sqrt{\lambda}}}) + (1 - e^{-\frac{\lambda_{in} \cdot n+1}{\sqrt{\lambda}}}) \tag{21}$$

$$\cdot (1 - \left[\frac{e^{\sqrt{\lambda}}}{(1 + e^{\sqrt{\lambda}})^2}(e^{-\frac{\lambda_{in} \cdot n}{\sqrt{\lambda}}} - e^{-\frac{k \cdot n}{\sqrt{\lambda}}})\right]^{T_{in}-1})\} \cdot \frac{\mathcal{P}_{S_1}}{d}. \tag{22}$$

Next, we prove the upper bound of the P_{FPN}. For $P_{1,2}$, we have

$$P_{1,2} = P\left\{\frac{1}{k} \cdot \hat{f_p} < f_c \le \lambda_{de} \cdot \hat{f_p} \wedge f_c > \lambda_{de} \cdot f_p | \hat{f_p} = n\right\}. \tag{23}$$

Notice that $f_p = \hat{f_p} - (\hat{f_p} - f_p)$, so we have

$$P_{1,2} = \sum_{i=\lceil \frac{n}{k} \rceil}^{\lfloor \lambda_{de} \cdot n \rfloor} P(f_c = i) \cdot P(\hat{f_p} - f_p > n - \frac{i}{\lambda_{de}}) \tag{24}$$

$$\le \left(1 - e^{-\frac{(\lambda_{de}{}^{-1}/k) \cdot n+1}{\sqrt{\lambda}}}\right) \cdot \frac{\mathcal{P}_{S_1}}{d}. \tag{25}$$

The proof of $P_{2,3}$ and $P_{1,3}$ of P_{FPN} are similar to P_{FPP}. □

The proof of upper bounds of false negatives can be obtained similarly. Due to the space limitation, we omit them here.

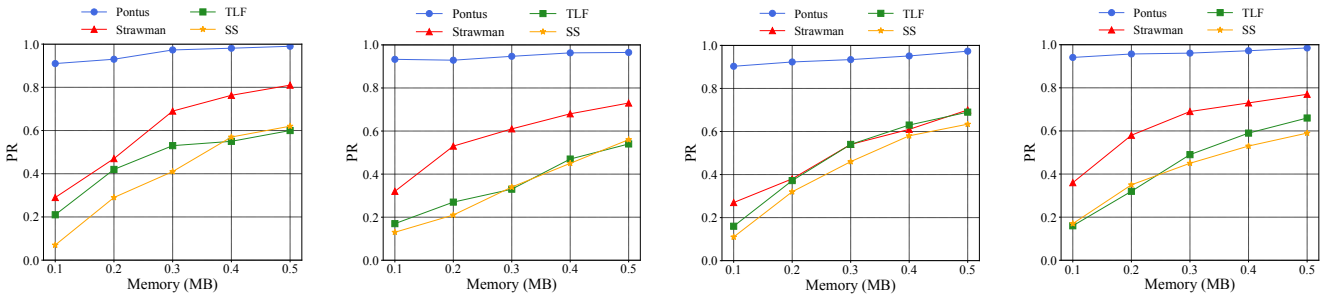
9 EXPERIMENTAL RESULTS

9.1 Experiment Setup and Metrics

Datasets: We use four datasets in our experiment. For each dataset, we divide it into windows that have fixed size $N = 10000$. Note that the ground truth of the wave is different when we set different T_{in}, T_{de}, k. An example of the number of ground truths in the following dataset is shown in Table 2. **1) IP Trace dataset:** The IP Trace dataset contains data streams of anonymized IP trace collected by CAIDA in 2019 [8]. Each item contains a source IP address (4 bytes) and a destination IP address (4 bytes). There are about 36M packages and 7.9M distinct items in the IP Trace dataset. **2) Data Center dataset:** The data center dataset[1] contains traces collected from the data centers studied in [36]. Each item (4 bytes) represents the ID of the trace. There are about 20M packages and 4.7M distinct items in the Data Center dataset. **3) MAWI dataset:** The MAWI dataset contains traffic traces collected by MAWI [24]. Each item contains a source IP address (4 bytes) and a destination IP address (4 bytes). There are about 20M packages and 5.3M distinct items in the MAWI dataset. **4) Synthetic dataset:** We use Web Polygraph [34] to generate synthetic dataset which follows Zipf [31] distribution. The length of each item is 4 bytes. We use the dataset with a skewness of 1.5 as the Synthetic dataset in the following experiments. There are about 20M packages and the number of distinct items is according to the skewness.

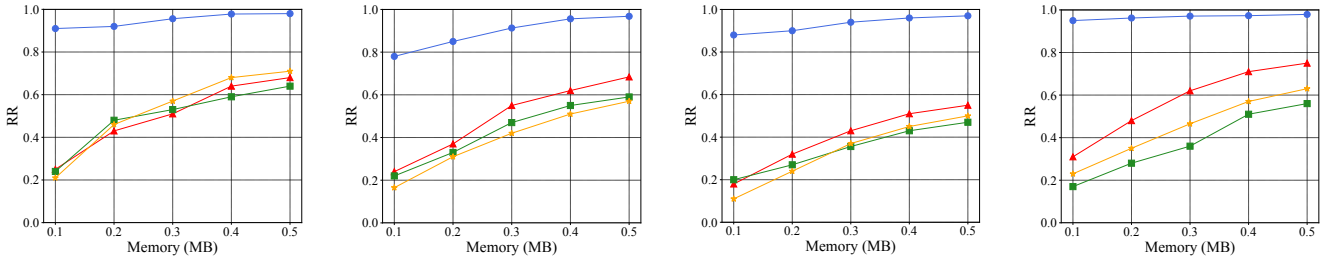
Implementation: Our software version of Pontus is implemented in C++. We conduct experiments on a server with 16-core CPUs (32 threads, Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz), and 64GB DRAM memory. In our software version of Pontus, we use BOB Hash [22] to implement the hash functions. We recommend a ratio of $7:2:1$ for the memory of S_1, S_2 and S_3 in practice to reduce the hash collisions since the number of waves is much smaller than the number of the potential waves and non-fluctuating items. We build our Pontus hardware prototype based on an EdgeCore wedge 100BF-65X Tofino switch with a local CPU of Intel(R) Xeon(R) CPU D-1527 @ 2.20GHz. The P4 code is compiled by Barefoot P4 Studio Software Development Environment(SDE). We use the traffic generator KEYSIGHT XGS12-SDL to generate high-speed traffic. We enable the Intel DPDK library on the server for high-performance traffic replay.

Metrics: **1) Recall Rate (RR):** Ratio of the number of correctly reported instances to the number of reported instances. **2) Precision Rate (PR):** Ratio of the number of correctly reported instances to the number of correct instances. **3) F1-Score:** $\frac{2 \cdot RR \cdot PR}{RR+PR}$. **4) Average Relative Error (ARE):** $\frac{1}{\Psi}\sum_{e \in \Psi}\sum_{a \in C}\frac{|a-\hat{a}|}{a}$, where



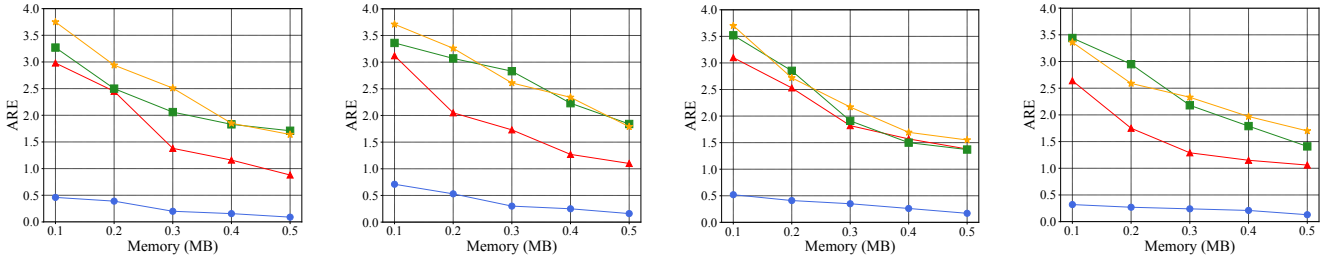
(a) Data Center (b) CAIDA (c) WIDE (d) Synthetic

Figure 4: Wave detection Precision Rate of Pontus and baseline solutions.



(a) Data Center (b) CAIDA (c) WIDE (d) Synthetic

Figure 5: Wave detection Recall Rate of Pontus and baseline solutions.



(a) Data Center (b) CAIDA (c) WIDE (d) Synthetic

Figure 6: Wave estimation Average Relative Error of Pontus and baseline solutions.

a is the true curve point of item e, $\hat{a}$ is the estimated curve point, $C = \{a_i, a_j, a_n, a_m\}$ is true curve points set of item e, and Ψ is the ground truth set. ARE evaluates errors in wave estimation. **5) Throughput:** Million insertions per second (MIPS). Throughput experiments are repeated 10 times to ensure statistical significance.

Ground Truth: To generate the ground truth label, we assign a bucket to each incoming item, which contains $T_{de} + T_l + T_{in}$ counters to record the frequencies of items across windows. At the end of each window, we iterate through all buckets and check if the frequency change of the counters in the buckets satisfies the definition of the wave. If so, we report it as a wave. During experiments, a wave detected by the algorithm is considered to be correct when its type (i.e., positive or negative), t (timestamp), w_{in}, w_{de}, and w_l equal those of the ground truth wave, which are used to calculate PR and RR. We also assign each bucket four counters to record curve points, which are used to calculate ARE.

9.2 Wave Detection/Estimation Experiments

We set $d = 3, \hat{d} = 4, T_{in} = T_{de} = 4, T_l = 10$, $k = 2$ and vary the memory from 0.1 MB to 0.5 MB. SpaceSaving [28] (SS) maintains a hash table with k buckets, and each bucket contains a K-V pair. Two-Level Filtering [37] (TLF) uses two hash tables: the first-level filter and the second-level filter, which contain k_1 and k_2 buckets, respectively. We extend the bucket in SS and TLF to maintain more information for tracing waves and construct baseline methods based on them for comparison.

The results of wave detection precision rate on different datasets are illustrated in Figure 4. The results show that Pontus obtains over 90% PR on all datasets even when the memory is very small, and even achieves near 98% PR when the memory is increased to 0.5 MB. The strawman solution, however, can barely detect waves when memory is small since it requires excessive memory to build CM

Dataset \ Metrics	# waves		F1-Score		ARE	
	pos.	neg.	pos.	neg.	pos.	neg.
Data Center	8942	2531	97%	99%	0.12	0.10
CAIDA	16894	6530	97%	98%	0.25	0.08
WIDE	7441	1657	98%	99%	0.21	0.05
Synthetic	10786	5688	97%	99%	0.17	0.11

Table 2: Performance regarding each type of wave.

sketches to reduce the hash collisions. It only starts to function after the memory is more than 0.2 MB, but still struggles to reach 75% PR even with 0.5 MB memory. The PRs of SS and TLF are even worse than the strawman solution. The results of wave detection recall rate on different datasets are illustrated in Figure 5. The results show that Pontus performs well even under very scarce memory of 0.1 MB, achieving about 80% RR on the CAIDA dataset and over 95% RR on other datasets. When memory is raised to 0.5 MB, Pontus successfully achieves nearly 99% RR. Meanwhile, the strawman solution again could barely operate under little memory and only gets near 50% and 75% RRs even with 0.5 MB memory. SS and TLF are incompetent under small memory, achieving about 19% and 15% RR with 0.1 MB memory, respectively. Though the RRs of SS and TLF increase as the memory increases, they merely achieve about 60% RR with 0.5 MB memory.

The results of wave estimation AREs on different datasets are illustrated in Figure 6. Pontus achieves much lower ARE than the strawman on all datasets. When memory is 0.5 MB, Pontus achieves only 0.1 ARE, which is about 10 × lower than the strawman solution. The AREs of SS and TLF are as high as 3.6 and 3.3 when memory is small, respectively, which are over 2 × higher than the strawman.

9.3 Performance for Each Type of Wave

Pontus is designed to detect and estimate both positive and negative waves simultaneously, as evaluated in previous experiments. To understand its performance in detecting either positive or negative waves alone, we present the F1-score and ARE for positive and negative waves obtained in the previous experiments separately (with the memory of 0.5 MB) in Table 2. As can be seen, the F1-Scores of detecting the negative waves are slightly (about 0.02) higher than those of positive waves for all four datasets. The AREs of detecting the negative waves are generally smaller than those of positive waves, though to varying degrees, across the four datasets.

$\lambda_{in}, \lambda_{de}$ are hyper-parameters related to T_{in}, T_{de}, k and the number of positive and negative wave. In our experiment, we set $\lambda_{in} = 1 + \frac{k-1}{T_{de}}, \lambda_{de} = 1 - \frac{k-1}{kT_{de}}$. Since the numbers of the positive waves are much higher than those of negative ones in all four datasets, we slightly increase λ_{de} to maintain more potential negative waves to reduce the probability of hash collision of negative waves. In practice, administrators can adjust λ_{in} and λ_{de} to achieve good performance in their specific data mining or networking scenarios.

We count the numbers of positive and negative waves in the experimental datasets in Table 3. Next, we offer more insights into their potential applications in real-life scenarios. For the purpose of user privacy protection, public datasets do not contain the payload of packets. Therefore, we classify applications based on port numbers. As shown in Table 3, for all experimental datasets, we observe waves mainly on ports 80, 23, 53, 22, 123, and others. Waves

Port \ Dataset		Data Center	CAIDA	WIDE	Synthetic
80	pos.	2773	4395	2683	1325
	neg.	1240	3569	1411	2958
23	pos.	3427	6763	3632	3711
	neg.	14	30	7	78
53	pos.	1411	4187	209	1774
	neg.	998	2841	154	2426
22	pos.	682	924	637	2449
	neg.	6	13	0	22
123	pos.	137	531	123	1301
	neg.	2	8	4	16
Others	pos.	512	94	157	226
	neg.	271	69	81	188

Table 3: Numbers of positive and negative waves on different ports in each experimental dataset.

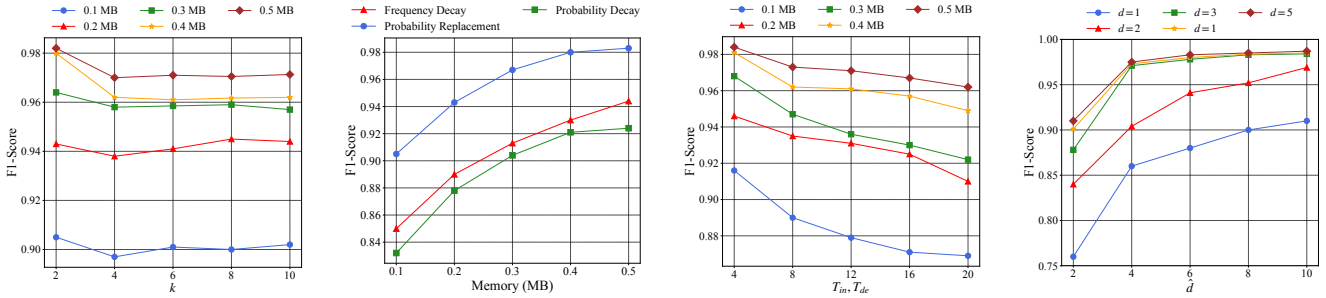
detected on port 80 can be adopted in online advertising-related applications, since this port is particularly reserved by the HTTP protocol for web browsing. Port 53 is designated for DNS nameservers. As such, waves detected on this port can be used for DNS query-related scenarios. Waves on ports 23, 123, and 22 [27] may be caused by attacks in the networks. For example, Mirai [3] attacks at port 23, and Worm [35] attacks at port 123. There are significantly more positive waves than negative waves in all datasets, except for the Synthetic dataset, where we intentionally generate more negative waves to see its impacts on the results. In particular, the vast majority of waves that appear at ports 23, 22, and 123 are positive waves. This is because attackers usually gradually ramp up the numbers of attacking packets and then decrease to bypass DDoS detection, which is consistent with the definition of the positive wave. When deploying Pontus in the real world, a detailed analysis can be done through the payloads of packets.

9.4 Update Time and Packet Loss

We set $d = 3, \hat{d} = 4, T_{in} = T_{de} = 4, T_l = 10, k = 2$ and vary the memory from 0.1 MB to 10 MB. Each window is set to contain 20K packets (items) and each packet length is 1500 Byte. The update time and packet loss per window are illustrated in Figure 8(a). As the memory increases, the update times of the control plane and data plane increase slowly in the beginning until the memory reaches 1 MB, after which the growth escalates rapidly. The increases in update times cause the packet loss rate to rise in a similar trend: it increases slightly to just above 1% at 1 MB memory and then rises rapidly to 4.7% at 10 MB memory. The results in Figure 8(b) show that the accuracy first increases sharply to around 99% at 0.5 MB but decreases as the memory approaches 10 MB. This is because overly small memory leads to frequent hash collisions while overly large memory causes prolonged update time, which induces an increased packet loss rate and thus decreased accuracy. In our experiments, the memory of Pontus is set to 0.5 MB.

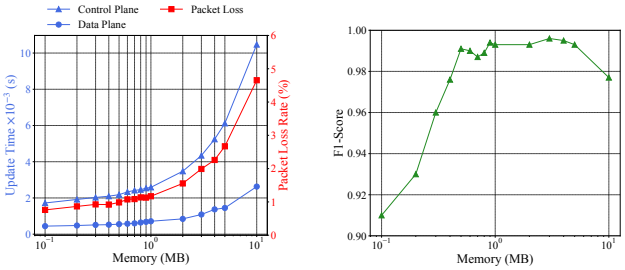
9.5 Parameters Effects Analysis

Effects of k: In this experiment, we vary the k from 2 to 10. The results in Figure 7(a) show that Pontus performs well, achieving over 92% F1-Score regardless of the value of k. When k varies from



(a) k (b) Replacement strategy (c) T_{in}, T_{de} (d) $d, \hat{d}$

Figure 7: Analysis of different parameters and replacement strategies' effects.



(a) Update time and Packet Loss (b) F1-Score

Figure 8: Update Time and Packet Loss.

2 to 4, the F1-Score of Pontus drops slightly. The reason is that the number of items inserted into S_2 or S_3 decreases, so hash collisions in S_1 increase. When k varies from 4 to 10, the number of items inserted into S_2 or S_3 is stable. Thus, the F1-Score of Pontus is also stable. In general, Pontus is insensitive to k when k varies.

Effects of replacement strategy: In Figure 7(b), we compare three replacement strategies in S_1 as mentioned in Section 5. The results show that probability replacement performs the best among the three strategies while frequency decay performs worst. Therefore, we use probability replacement in our implementation.

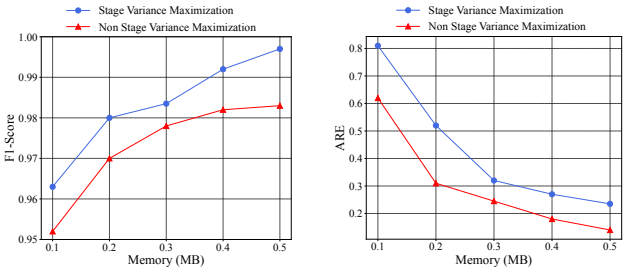
Effects of T_{in}, T_{de}: We vary T_{in}, T_{de} from 4 to 20. The results in Figure 7(c) show that the F1-Score of Pontus slightly decreases as T_{in} and T_{de} increase. The reason the number of items inserted into S_2 and S_3 increases as T_{in} and T_{de} increase. Thus, the hash collisions in S_2 and S_3 increase.

Effects of $d, \hat{d}$: We vary d from 1 to 5 and $\hat{d}$ from 2 to 10. The results in Figure 7(d) show that the F1-Score of Pontus increases as $d, \hat{d}$ increase. However, the insertion throughput will decrease as $d, \hat{d}$ increase. Therefore, we set $d = 3$ and $\hat{d} = 4$ to make a trade-off between accuracy and throughput.

Effects of Stage Variance Maximization : This experiment evaluates the effects of Stage Variance Maximization. We vary the memory from 0.1 MB to 0.5 MB. Figure 9(a) shows the F1-Scores achieved with and without using Stage Variance Maximization. As can be seen, the Stage Variance Maximization improves the F1-Score of Pontus for about 0.1 when the memory is small. When the memory is up to 0.5 MB, using Stage Variance Maximization improves about 0.15. Figure 9(a) shows the AREs achieved with and without using Stage Variance Maximization. Our experiments show that using Stage Variance Maximization reduces about 0.2 ARE of Pontus when the memory is 0.1 MB. When the memory is up to 0.5 MB, using Stage Variance Maximization reduces about 0.1 ARE.

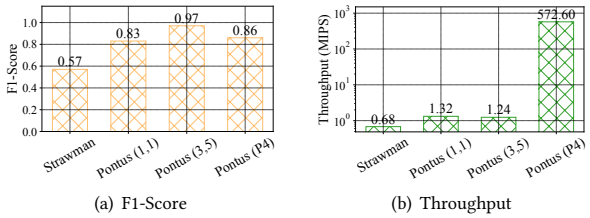
9.6 Experiments on P4 version of Pontus

We compare the throughput and F1-Score of the strawman, software version, and P4 version of Pontus. We set the memory of Pontus and strawman to 0.3 MB and $T_{in} = T_{de} = 4, T_l = 10, k = 2$. Pontus (1,1) and Pontus (3,5) refer to $d = 1, \hat{d} = 1$ and $d = 3, \hat{d} = 5$, respectively.



(a) F1-Score (b) ARE

Figure 9: Effects of Stage Variance Maximization.



(a) F1-Score (b) Throughput

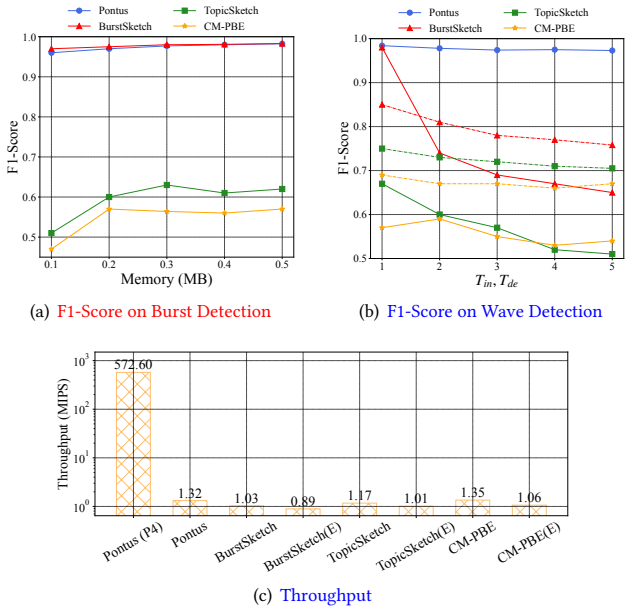
Figure 10: Experiments on P4 version of Pontus.

Figure 10(a) shows that Pontus (P4) achieves 86% F1-Score which is higher than those of Pontus (1,1) and strawman. The F1-Score

of Pontus (P4) is about 0.1 less than that of Pontus (3,5) due to the approximate probabilistic replacement used in P4. Figure 10(b) shows that the throughput of Pontus (P4) is about 842, 433, and 462 × higher than those of strawman, Pontus (1,1) and Pontus (3,5), respectively. In general, the P4 version of Pontus achieves outstanding processing speed with high accuracy.

9.7 Comparison with Prior Arts

In this part, we compare Pontus with BurstSketch [43], Topic-Sketch [39] and CM-PBE [30] on burst detection and wave detection.



(a) F1-Score on Burst Detection (b) F1-Score on Wave Detection

(c) Throughput

Figure 11: Performance comparison with prior arts. In sub-figure (b), the dotted lines indicate the enhanced versions of the corresponding solutions.

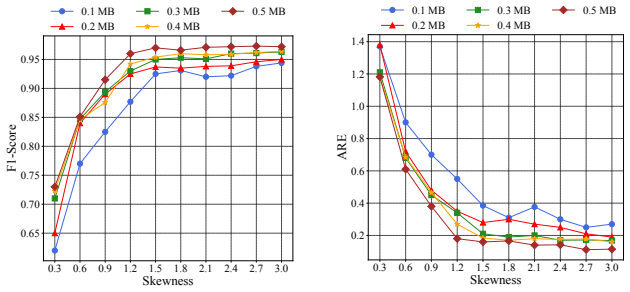
In burst detection, we set $d = 3, \hat{d} = 4, T_{in} = T_{de} = 1, T_l = 10, k = 2$ and vary the memory from 0.1 MB to 0.5 MB. The F1-Scores of Pontus and the prior arts in burst detection are illustrated in Figure 11(a). The results show that BurstSketch and Pontus achieve comparable F1-Score (above 95%) in detecting bursts most of the time and BurstSketch only outperforms Pontus slightly when the memory is small. The F1-Scores of TopicSketch and CM-PBE are only around half of that of Pontus since they only consider the increase of a burst and do not consider the decrease of a burst.

In wave detection, we vary T_{in}, T_{de} from 1 to 5. The bigger T_{in}, T_{de} is, the more complicated the waves in data streams are. To evaluate the wave detection performance of Pontus, we implement enhanced versions of prior arts, which are called as BurstSketch(E), TopicSketch(E), and CM-PBE(E), respectively. Specifically, to support the tracking of waves, we add additional hash tables to the original schemes to track item frequency in multiple windows and extend each bucket to maintain necessary information. As shown in Figure 11(b), Pontus retains about 97% F1-Score, regardless of the variation of T_{in}, T_{de}. Though the enhanced versions of prior arts outperform their original versions, their F1-Scores, ranging from 0.65 to 0.85, are still significantly inferior to that of Pontus.

As shown in Figure 11(c), the throughput of P4 version Pontus is over 500 × higher than others. The throughput of software version Pontus is about 30% and 13% higher than BurstSketch and TopicSketch, respectively, though slightly lower than CM-PBE. The enhanced versions of prior arts suffer from degraded throughput than their original versions due to insertions in additional hash tables.

Overall, the results show that Pontus significantly outperforms prior arts in detecting waves, while still achieving one of the best performances in detecting bursts.

9.8 Experiments on Distributions



(a) F1-Score (b) ARE

Figure 12: Experiments on dataset skewness.

To evaluate the effects of dataset skewness, we generate 10 synthetic datasets with skewness from 0.3 to 3. We set $d = 3, \hat{d} = 4, T_{in} = T_{de} = 4, T_l = 10, k = 2$ and the memory of Pontus to 0.3 MB. Figure 12(a) shows the F1 scores under different dataset skewness. The results show that the F1-Score under a skewness of 0.3 is about 1.35 × and 1.4 × lower than those under a skewness of 0.9 and 3, respectively. Figure 12(b) shows the AREs achieved under different dataset skewness. The ARE under a skewness of 0.3 is about 2.3 × and 7.5 × higher than those under a skewness of 0.9 and 3, respectively. Our results reveal that Pontus performs well under the high skewness of the dataset and is robust against dataset skewness.

10 CONCLUSION

In this paper, we propose the *wave*, a new data pattern in data streams, and propose Pontus, a lightweight wave detection and estimation framework. Pontus utilizes the data plane to process the incoming data stream and the control plane to detect waves. Pontus uses the Multi-Stage Progressive Tracking strategy to filter the illegal items in advance and the Stage Variance Maximization to minimize the estimation error. The P4 version of Pontus further improves the throughput. Moreover, we prove the theoretical error bound and establish upper bounds of false positive and false negative. Experiment results show that our Pontus is significantly faster and more accurate than the strawman.

REFERENCES

[1] Aditya Akella, Theophilus Benson. 2010. Data Center dataset. https://pages.cs.wisc.edu/~tbenson/IMC10_Data.html. (2010).

[2] Anonymous. 2022. Source code of Pontus. https://anonymous.4open.science/r/Pontus-08B6/. (2022).

[3] Manos Antonakakis and Tim April. 2017. Understanding the Mirai Botnet. In *USENIX Security*. 1093–1110.

[4] Barefoot Networks. 2021. Tofino Switch. https://www.barefootnetworks.com/products/brief-tofino. (2021).

[5] Ran Ben-Basat and Gil Einziger. 2017. Randomized admission policy for efficient top-k and frequency estimation. In *INFOCOM*. 1–9.

[6] Pat Bosshart and Dan Daly. 2014. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.* 44, 3 (2014), 87–95.

[7] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando A. Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM 2013 Conference*. 99–110.

[8] Caida. 2019. Anonymized 2019 internet traces. http://www.caida.org/data/overview/. (2019).

[9] Moses Charikar and Kevin C. Chen. 2002. Finding Frequent Items in Data Streams. In *ICALP*. 693–703.

[10] Peiqing Chen and Chen. 2021. Out of Many We are One: Measuring Item Batch with Clock-Sketch. In *SIGMOD/PODS*. 261–273.

[11] Tingting Chen and Yi Wang. 2006. Detecting lasting and abrupt bursts in data streams using two-layered wavelet tree. In *AICT-ICIW*. 30–36.

[12] Cloudflare DDoS Team. 2021. Meris Botnet. https://radar.cloudflare.com/notebooks/meris-botnet#meris_attacks_over_time. (2021).

[13] G. Cormode and S. Muthukrishnan. 2004. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. In *LATIN*. 29–38.

[14] Sean Patrick Donovan and Nick Feamster. 2014. Intentional Network Monitoring: Finding the Needle without Capturing the Haystack. In *HotNets*. 5:1–5:7.

[15] Cristian Estan and George Varghese. 2002. New directions in traffic measurement and accounting. In *SIGCOMM*. 323–336.

[16] Cristian Estan and George Varghese. 2003. Bitmap algorithms for counting active flows on high speed links. In *IMC*. 925–937.

[17] Flajolet and Philippe. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AOFA*. 127–146.

[18] P. Flajolet and G. N. Martin. 1985. Probabilistic counting algorithms for data base applications. *Journal of Computer & System Sciences* 31, 2 (1985), 182–209.

[19] Monia Ghobadi and Ratul Mahajan. 2016. Optical Layer Failures in a Large Backbone. In *IMC*. 461–467.

[20] Lukasz Golab and David DeHaan. 2003. Identifying frequent items in sliding windows over on-line packet streams. In *IMC*. 173–178.

[21] Nikita Ivkin and Zhuolong Yu. 2019. QPipe: quantiles sketch fully in the data plane. In *CoNEXT*. 285–291.

[22] Robert J. Jenkins Jr. 1995. BOB Hash website. http://burtleburtle.net/bob/hash/evahash.html. (1995).

[23] Z. Karnin and K. Lang. 2016. Optimal Quantile Approximation in Streams. In *FOCS 2016*. 71–78.

[24] Koushirou Mitsuya Kenjiro Cho. 2000. Traffic Data Repository at the WIDE Project. In *USENIX FREENIX Track*. 51–51.

[25] Jason Kim and Hyojoon Kim. 2021. Analyzing Traffic by Domain Name in the Data Plane. In *SOSR*. 1–12.

[26] Jon M. Kleinberg. 2002. Bursty and hierarchical structure in streams. In *SIGKDD*. 91–101.

[27] Yeheskel Lapid and Niv Ahituv. 1986. Approaches to handling "Trojan Horse" threats. *Comput. Secur.* 5, 3 (1986), 251–256.

[28] Ahmed Metwally and Divyakant Agrawal. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *ICDT*. 398–412.

[29] Congcong Miao and Jingyu Xiao. 2022. Detecting Ephemeral Optical Events with OpTel. In *NSDI*. 1–12.

[30] Debjyoti Paul and Yanqing Peng. 2019. Bursty Event Detection Throughout Histories. In *ICDE*. 1370–1381.

[31] David M. W. Powers. 1998. Applications and Explanations of Zipf's Law. In *NeMLaP/CoNLL*. 151–160.

[32] Shouke Qin and Weining Qian. 2005. Adaptively detecting aggregation bursts in data streams. In *DASFAA*. 435–446.

[33] Shouke Qin and Weining Qian. 2006. Approximately processing multi-granularity aggregate queries over data streams. In *ICDE*. 67–67.

[34] Alex Rousskov and Duane Wessels. 2004. High-performance benchmarking with Web Polygraph. *Softw. Pract. Exp.* 34, 2 (2004), 187–211.

[35] Eugene H. Spafford. 1989. The Internet Worm Incident. In *ESEC '89*. 446–468.

[36] Aditya Akella Theophilus Benson. 2010. Network traffic characteristics of data centers in the wild. In *IMC*. 267–280.

[37] Shobha Venkataraman and Dawn Xiaodong Song. 2005. New Streaming Algorithms for Fast Detection of Superspreaders. In *NDSS*.

[38] L. Wang and G. Luo. 2013. Quantiles over data streams: an experimental study. *VLDB* 25, 4 (2013), 1–24.

[39] Wei Xie and Feida Zhu. 2013. TopicSketch: Real-Time Bursty Topic Detection from Twitter. In *ICDM*. 837–846.

[40] Dai Yumei and Liang Yu. 2014. POSTER: A Hybrid Botnet Ecological Environment. In *SIGSAC*. 1421–1423.

[41] Y. Zhang, J. Li, and Y. Lei. 2020. On-off sketch: a fast and accurate sketch on persistence. *VLDB* 14, 2 (2020), 128–140.

[42] Yinda Zhang and Zaoxing Liu. 2021. CocoSketch: high-performance sketch-based measurement over arbitrary partial key query. In *SIGCOMM*. 207–222.

[43] Zheng Zhong and Shen Yan. 2021. BurstSketch: Finding Bursts in Data Streams. In *SIGMOD*. 2375–2383.

[44] Yunyue Zhu. 2003. Efficient elastic burst detection in data streams. In *SIGKDD*. 336–345.

[45] Moshe Zukerman, Timothy D. Neame, and Ron Addie. 2003. Internet Traffic Modeling and Future Technology Implications. In *INFOCOM 2003*. 587–596.