A Unified Framework for Mining Batch and Periodic Batch in Data Streams

Zirui Liu*, Xiangyuan Wang*, Yuhan Wu*, Tong Yang*†, Kaicheng Yang*, Hailin Zhang*, Yaofeng Tu‡, and Bin Cui*

Abstract—*Batch* is an important pattern in data streams, which refers to a group of identical items that arrive closely. We find that some special batches that arrive periodically are of great value. In this paper, we formally define a new pattern, namely *periodic batches*. A group of periodic batches refers to several batches of the same item, where these batches arrive periodically. Studying periodic batches is important in many applications, such as caches, financial markets, online advertisements, networks, *etc.* This paper proposes a unified framework, namely the HyperCalm sketch, to detect batch and periodic batch in data streams. HyperCalm sketch takes two phases to detect periodic batches. In phase 1, we propose a time-aware Bloom filter, called HyperBloomFilter (*HyperBF*), to detect batches. In phase 2, we propose an enhanced top-k algorithm, called Calm Space-Saving (*CalmSS*), to report top-k periodic batches. Extensive experiments show HyperCalm outperforms the strawman solutions 4$\times$ in term of average relative error and 98.1$\times$ in term of speed. All related codes are open-sourced.

Index Terms—Data Mining, Data Stream, Sketch, Periodic Batch

1 Introduction

1.1 Background and Motivation

Batch is an important pattern in data streams [2], which is a group of identical items that arrive closely. Two adjacent batches of the same item are spaced by a minimum interval $\mathcal{T}$, where $\mathcal{T}$ is a predefined threshold. Although batches can make a difference in various applications, such as cache [2], networks [3], and machine learning [4], [5], it is not enough to just study batches. For instance, in cache systems, with just the measurement results of batches, we are still not able to devise any prefetching method and replacement policy. Further mining some special patterns of batches is of great importance. On the basis of batches, we propose a new pattern, namely periodic batch. A group of periodic batches refers to α consecutive batches of the same item, where these batches arrive periodically. We call α the periodicity. Finding top-k periodic batches refers to reporting k groups of periodic batches with the k largest periodicities.

Studying top-k periodic batches is important in practice. For example, consider a cache stream formed by many memory access requests where each request is an item, periodic batches provide insights to improve the cache hit rate. With the historical information of periodic batches, we can forecast the arrival time of new batches, and prefetch the item into cache just before its arrival. For another example, in financial transaction streams, periodic transaction batches could be an indicator of illegal market manipulation [6]. By detecting periodic batches in real time, we can quickly find those suspicious clients that might be laundering money. Periodic batches are also helpful in recommendation systems and online advertisements, where the data stream is generated when users click or purchase different commodities. A batch forms when users continuously click or purchase the same type of commodities. In this scenario, periodic batches imply users' seasonal and periodic browsing or buying behaviors [7] (*e.g.*, Christmas buying patterns that repeat yearly, or seasonal promotion-related user behaviors). Studying periodic batches can help us to better understand customer behavior, so that we can deliver appropriate advertisements promptly to customers. In addition, periodic batches are also important in networks. In network stream, most TCP senders tend to send packets in periodic batches [8]. If we can forecast the arrival time of future batches, we can pre-allocate resources to them, or devise better strategies for load balancing. To our knowledge, there is no existing work studying periodic batches, and we are the first to formulate and address this problem.

Finding periodic batches is a challenging issue. First, finding batches is already a challenging issue. Until now, the state-of-the-art solution to detect batches is Clock-Sketch [2], which records the last arrival time of recent items in a cyclic array, and uses another thread to clean the outdated information using CLOCK [9] algorithm. However, to achieve high accuracy, it needs to scan the cyclic array very fast, which consumes a lot of CPU resources. Second, periodic batch is a more fine-grained definition, and thus finding periodic batches is more challenging than just finding batches. The goal of this paper is to design a compact sketch algorithm that can accurately find periodic batches with small space- and time- overhead.

1.2 Our Proposed Solution

This paper proposes a unified framework, called HyperCalm sketch, to detect batch and periodic batch in data

**National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, China.*
†Peng Cheng Laboratory, Shenzhen, China.
‡ZTE Corporation, China.
The preliminary version of this paper titled "HyperCalm Sketch: One-Pass Mining Periodic Batches in Data Streams" is published in the 39th IEEE International Conference on Data Engineering (IEEE ICDE 2023) [1], Anaheim, California, USA, April 3-7, 2023

streams in real time. HyperCalm takes two phases to find top-k periodic batches. In phase 1, for each item e arriving at time t, we check whether it is the start of a batch. If so, we query a TimeRecorder queue to get the arrival time $\hat{t}$ of the last batch of e, and calculate the batch interval $V = t - \hat{t}$. Then we send this batch and its interval $\langle e, V \rangle$ to the second phase. In phase 2, we check periodicity and manage to record top-k periodic batches, *i.e.*, top-k $\langle e, V \rangle$ pairs. In phase 1, we devise a better algorithm than the state-of-the-art algorithm for detecting batches, Clock-Sketch [2]. In phase 2, we propose an enhanced top-k algorithm, which naturally suits our periodic batch detection scenario.

In phase 1, we propose a time-aware version of Bloom filter, namely HyperBloomFilter (HyperBF for short), to detect batches. For each incoming item, phase 1 should report whether the item is the start of a batch. In other words, this is an existence detection algorithm. In addition, phase 1 should be aware of the item arrival time to divide a series of the same item into many batches. Bloom filter [10] is the most well-known memory-efficient data structure used for existence detection. However, the existence detection of Bloom filter is only low-dimensional, *i.e.*, it is agnostic to time dimension. Typical work aware of time dimension is *Persistent Bloom filter* (PBF) [11]. It is an elegant variant of Bloom filter, which uses a set of carefully constructed Bloom filters to support *membership testing for temporal queries* (MTTQ) (*e.g.*, has a person visited a website between 8:30pm and 8:40pm?). MTTQ and batch detection are different ways to be aware of time dimension. To enable Bloom filter to be aware of time, our HyperBF extends each bit in Bloom filter into a 2-bit cell, doubling the memory usage. Compared to the standard Bloom filter, HyperBF has the same number of hash computations and memory accesses for each insertion and query. The only overhead for time awareness is doubling the memory usage, which is reasonable and acceptable.

In phase 2, we propose an enhanced top-k algorithm, called Calm Space-Saving (CalmSS for short), to report top-k periodic batches. For each incoming batch and its interval, *i.e.*, $\langle e, V \rangle$, phase 2 should keep periodic batches with large periodicities, and evict periodic batches with small periodicities. In other words, phase 2 keeps frequent $\langle e, V \rangle$ pairs, and evicts infrequent $\langle e, V \rangle$ pairs, which is is a top-k algorithm. Typical top-k algorithms include Space-Saving [12], Unbiased Space-Saving [13], and Frequent [14]. However, their accuracy is significantly harmed by cold items [1]. This problem is more serious in our scenario of periodic batch detection. This is because one infrequent item may have multiple batches, and one frequent item may also have multiple batches without periodicity. Both the two cases above increase the number of cold $\langle e, V \rangle$ pairs. To identify and discard cold items, Cold Filter [17] and LogLogFilter [18] record the frequencies of all items in a compact data structure. However, considering the large volume of data stream, this structure will be filled up very quickly, and needs to be cleaned up periodically. To ensure the one-pass property of our solution, it is highly desired to devise a data structure which will never be filled up. Instead of recording all items, our solution is to just record the frequencies of some items in the sliding window. Rather than using existing sliding window algorithms [19], [20], [21], this paper designs an LRU queue working together with Space-Saving because of the following reasons. First, our LRU queue is elastic: users can dynamically tune its memory usage to maintain a satisfactory accuracy. Second, our LRU queue has elegant theoretical guarantees (see details in § 3.3). Third, our LRU queue can be naturally integrated into the data structure of Space-Saving (see details in § 3.4): such combination achieves higher accuracy and higher speed. Our combination is faster because the LRU queue efficiently filters most cold items, and thus the complicated replacement operations incurred by cold items are avoided (see Figure 15c). Actually, besides the application of periodic batch detection, our LRU queue can improve the accuracy/speed of any streaming algorithms. We can handle any case that Cold filter can handle, and we are both time- and space- more efficient than Cold filter (§ 5.2). All related codes are open-sourced [22].

1. Cold items refer to items with small frequencies (*i.e.*, infrequent items), and hot items refer to items with large frequencies (*i.e.*, frequent items). In practice, most items are cold items, which appear just several times [15], [16]

1.3 Key Contributions

- We formulate the problem of finding periodic batches in data streams. We believe this is an important problem.
- We propose an accurate, fast, and memory efficient HyperCalm sketch to detect periodic batches in real time. Both the two components of HyperCalm, HyperBF and CalmSS, significantly outperform the SOTA in detecting batches and finding top-k items, respectively.
- We derive theoretical guarantees for our HyperBF and CalmSS, and validate our theories using experiments.
- We conduct extensive experiments, and the results show that HyperCalm outperforms the strawman solutions $4\times$ in term of error and $98.1\times$ in term of speed.
- We apply HyperCalm to the scenarios of cache and network measurement, showing that periodic batches can benefit real-world application. We also integrate HyperCalm into Apache Flink [23] and Redis [24].

2 Background and Related Work

2.1 Problem Statement

Batches: A data stream is an infinite sequence of items where each item is associated with a timestamp. A batch is defined as a group of identical items in the data stream, where the time gap between two adjacent batches of the same item must exceed a predefined threshold $\mathcal{T}$. For convenience, in this paper, two adjacent batches mean two batches belong to the same item by default. The arrival time of a batch is defined as the timestamp of the first item of this batch. We define the interval/time gap between two adjacent batches as the interval between their arrival times.

Periodic batches: A group of periodic batches refers to α consecutive batches of the same item, where these batches arrive with a fixed time interval. We call α the *periodicity*. Here, the "fixed time interval" is not the exact time, but the approximate (noise-tolerant) time rounded up to the nearest time unit (*e.g.*, one millisecond). Finding top-k periodic batches refers to reporting k groups of periodic batches with the k largest periodicities. Note that one item may have more than one group of periodic batches, and thus can be reported more than once.

Example (Figure 1): We use an example to further clarify our problem definition. We focus on two distinct items e_1 and e_2 in the data stream. For e_1, its 6 batches form a group of periodic batches. For e_2, it has two groups of periodic batches, with the periodicities of 4 and 5, respectively. Note that some batches of e_2 just have one item.

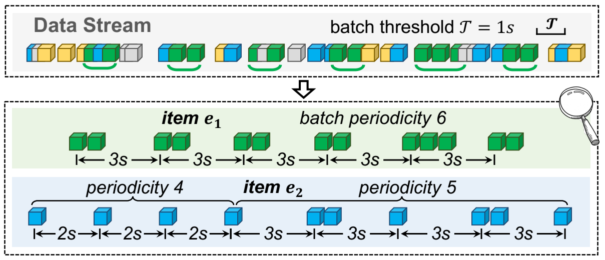


Fig. 1: Example of periodic batches.

Discussion: The definition of periodic batches is a design choice related to final application. We think our definition of periodic batches is most general, which can benefit many real-world applications (see § 5.8 as an example). However, certain application may also care about other aspects of periodic batches, such as batch size and distance. For example, some application may just want to detect those periodic batches that are large enough in size. It is not hard to detect those variants of periodic batches by adding small modification to our solution. For example, we will discuss how to detect periodic large batches in § 3.7 and demonstrate its application in § 5.9.

2.2 Related Work

Related work is divided into three parts: 1) algorithms for batch detection; 2) algorithms for finding top-k frequent items; and 3) algorithms for mining periodic patterns.

Batch detection: Item batch is defined very recently in [2], which proposes Clock-Sketch to find batches. Clock-Sketch consists of an array of s-bits cells. For each incoming item, it sets the d hashed cells as $2^s - 1$. For query, if one of the d hashed cells is zero, it reports a batch. Clock-Sketch uses an extra thread to cyclically sweep the cell array at a constant speed and decreases the swept non-zero cells by one. The sweeping speed is carefully selected to avoid false-positive errors. Besides Clock-Sketch, some sliding window algorithms can be applied to find batches, including *Time-Out Bloom Filter (TOBF)* [25] and *SWAMP* [26].

Finding top-k frequent items: To find top-k frequent items in data streams, existing approaches maintain a synopsis data structure. There are two kinds of synopses: *sketches* and *KV tables*. **1)** Sketches usually consist of multiple arrays, each of which consists of multiple counters. These counters are used to record the frequencies of the inserted items. Typical sketches include CM [27], CU [28], Count [29], and more [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43]. However, sketches are memory inefficient because they record the frequencies of all items, which is actually unnecessary. **2)** KV tables record only the frequent items. Typical KV table based approaches include Space-Saving [12], Unbiased Space-Saving [13], Frequent [14], and more [44]. Space-Saving and Unbiased Space-Saving record the approximate top-k items in a data structure called Stream-Summary. However, their accuracy is significantly degraded by cold items. To address this issue, Cold Filter [17] uses a two-layer CU sketch to filter cold items. However, as aforementioned, the structure of Cold Filter will be filled up very quickly, and cleaning the full Cold Filter will inevitably incur error and time overhead.

Mining periodic patterns: Although there have been some algorithms aiming at mining periodicity in time sequence data [45], [46], [47], [48], [49], [50], [51], their problem definitions are different from ours. More importantly, most of them do not meet the requirements of data stream model processing: 1) each item can only be processed once; 2) the processing time of each item should be $O(1)$ complexity and fast enough to catch up with the high speed of data streams. For example, TiCom [48] defines a periodical problem in an incomplete sequence data, and develops an iterative algorithm with time complexity of $O(n^2)$. RobustPeriod [46] proposes an algorithm based on discrete wavelet transform with time complexity of $O(n \log n)$. Further, there are some works which elegantly use Fast Fourier Transform (FFT) or Auto Correlation Function (ACF) to address different definitions of periodic items, such as SAZED [49]. These algorithms need to process one item multiple times, and thus cannot meet the above two requirements.

TABLE 1: Symbols frequently used in this paper.

Symbol	Meaning
e	ID of an **item** in a data stream
$\mathcal{T}$	Batch threshold spacing two adjacent batches
d	Number of arrays in HyperBF
$\mathcal{B}_i$	The i^{th} array of HyperBF
m	Number of 2-bit cells in each array $\mathcal{B}_i$
l	Number of 2-bit cells in each block
$h_i(\cdot)$	Hash function mapping an item into a cell in $\mathcal{B}_i$
V	Time interval of two adjacent batches of an item
c	Length of the TimeRecorder queue
$E = \langle e, V \rangle$	An **entry** in phase 2, which is the concatenation of an item e and its batch interval V, *i.e.*, $\langle e, V \rangle$
w	Length of the LRU queue in CalmSS
$\mathcal{P}$	Promotion threshold of the LRU queue
b	Number of slots in each bucket of bucketized TimeRecorder/LRU-Queue/Space-Saving
z	Number of buckets in bucketized TimeRecorder/ LRU-Queue/Space-Saving
$g(\cdot)$	Hash function mapping items/entries into buckets in bucketized TimeRecorder/ LRU-Queue/Space-Saving
$\mathcal{L}$	Batch size threshold of periodic large batch
$\mathcal{C}_i$	The i^{th} array of CM sketch used to report batch size

3 The HyperCalm Sketch

Overview (Figure 2): The workflow of the HyperCalm sketch consists of two phases: **1)** A HyperBloomFilter (HyperBF) detecting the start of batches; and **2)** A Calm Space-Saving (CalmSS) recording and reporting top-k periodic batches. In addition, we design a TimeRecorder queue to record the last batch arrival time for potential periodic batches. Given an incoming item e arriving at time t, we first propose HyperBF to check whether it is the start of a batch. If so, we query the TimeRecorder queue to get the arrival time $\hat{t}$ of the last batch of e and calculate the batch interval $V = t - \hat{t}$. [2] Then we update the arrival time of last batch of e in the TimeRecorder queue to t. Next, we

2. To tolerate noise in batch interval, V is rounded up according to the regulations described in the parameter setting part of § 5.3.

send e and its batch interval V to CalmSS to detect top-k periodic batches. We combine the ID of item e and its interval V to form an ***entry*** $E = \langle e, V\rangle$, and insert the entry into CalmSS. CalmSS reports k groups of periodic batches with the k largest periodicities, *i.e.*, reports top-k entries with the k largest frequencies, where each entry is an $\langle e, V\rangle$ pair. We will discuss that the insertion time complexity of each component of the HyperCalm sketch is $O(1)$, and thus its overall time complexity is also $O(1)$. The main symbols used in this paper are listed in Table 1.

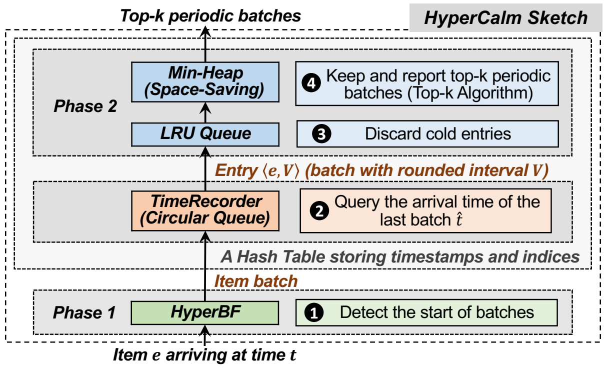

Top-k periodic batches
HyperCalm Sketch
Phase 2
Min-Heap (Space-Saving)
4 Keep and report top-k periodic batches (Top-k Algorithm)
LRU Queue
3 Discard cold entries
Entry ⟨e,V⟩ (batch with rounded interval V)
TimeRecorder (Circular Queue)
2 Query the arrival time of the last batch t̂
A Hash Table storing timestamps and indices
Item batch
Phase 1
HyperBF
1 Detect the start of batches
Item e arriving at time t

Fig. 2: HyperCalm sketch workflow.

3.1 The HyperBF Algorithm

Rationale: To enable Bloom filter to be time-aware, the key technique of HyperBF is to extend every bit in Bloom filter into a 2-bit cell, and use these cells to compactly record the approximate last arrival time of recent items. Although we can also use 3-bit or 4-bit cells, we find that under fixed memory, using 2-bit cells achieves the best accuracy. Since 2-bit cell can represent 4 states (0∼3), HyperBF cyclically divides the timeline into three kinds of time slices (1∼3), and the length of each time slice is $\mathcal{T}$, where $\mathcal{T}$ is the predefined *batch threshold* (see Figure 4a). These time slices are recorded in the 2-bit cells of HyperBF. HyperBF needs to clean all outdated time slices efficiently. Rather than using an extra thread like Clock-Sketch [2], HyperBF incidentally cleans the outdated cells during each insertion. Compared to standard Bloom filter, HyperBF has the same number of hash computations and memory accesses for each insertion and query. Further, we propose a novel *Asynchronous Timeline* technique to significantly reduce the error of HyperBF. Theoretical guarantees of HyperBF are provided in § 4.

Data structure: HyperBF consists of d arrays $\mathcal{B}_1, \cdots, \mathcal{B}_d$. Each array $\mathcal{B}_i$ has m 2-bits cells $\mathcal{B}_i[1], \cdots, \mathcal{B}_i[m]$, which are evenly divided into $\frac{m}{l}$ blocks with l 2-bit cells. Each block can fit into the size of a cache line, and thus could be read or write through one memory access. When checking one cell, we can incidentally access the other cells in its block, which does not incur extra memory accesses. Each array $\mathcal{B}_i$ is associated with a hash function $h_i(\cdot)$ that maps an item into a cell in it. As mentioned above, HyperBF divides the timeline into three kinds of time slices (1∼3). Each cell stores a time slice (1∼3) or a zero flag (0). We preserve the zero value of cells as the batch flag: once an incoming is mapped into a cell with batch flag, a new batch starts. For example in Figure 3, HyperBF has 2 cell arrays, each of which has 4 2-bit cells which are divided into 2 blocks. For simplicity, each block has $l = 2$ cells here. In practice, we can set the *block* size to any value no more than 64B ($l \leqslant 256$), *i.e.*, no more than the cache line size. All cells are initialized to 0.

Insert: For each incoming item e with timestamp t, we first calculate the current time slice $s = \lfloor \frac{t}{\mathcal{T}} \rfloor \mod 3 + 1$. We calculate the d hash functions to locate the d *hashed cells* of e: $\mathcal{B}_1[h_1(e)], \cdots, \mathcal{B}_d[h_d(e)]$. For each hashed cell, we check the block which the cell resides, and incidentally clean *outdated cells* to zero flag. Specifically, if the current time slice is 1, time slice 2 is outdated; if the current time slice is 2, time slice 3 is outdated; if the current time slice is 3, time slice 1 is outdated. Due to the high speed of the data stream, all outdated cells will be cleaned in time (see theoretical results in § 4). After cleaning, if any one of the d hashed cells is zero flag, HyperBF reports the start of a batch. Finally, we update all d hashed cells to the current time slice s.

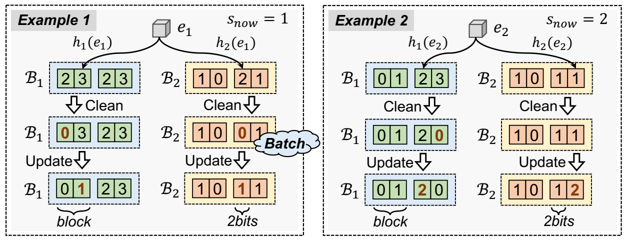


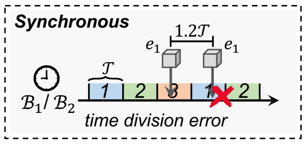
Fig. 3: Two examples of HyperBF ($d = 2$, $m = 4$, $l = 2$).

Example 1 (left of Figure 3): For item e_1 arriving at time slice $s_{now} = 1$, we first locate its two hashed cells $\mathcal{B}_1[2]$ and $\mathcal{B}_2[3]$ by calculating $h_1(e_1)$ and $h_2(e_1)$. Next, we clean the outdated cells with value 2. For $\mathcal{B}_1[2]$, we check all cells in its block (*i.e.*, $\mathcal{B}_1[1]$ and $\mathcal{B}_1[2]$), and clean the outdated cell $\mathcal{B}_1[1]$ to *zero*. For $\mathcal{B}_2[3]$, we clean the outdated cell $\mathcal{B}_2[3]$ to *zero*. After cleaning, since the second hashed cell $\mathcal{B}_2[3]$ is *zero*, we report the start of a batch. Finally, we update the two hashed cells to s_{now}.

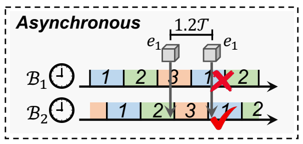
Example 2 (right of Figure 3): For item e_2 arriving at time slice $s_{now} = 2$, we first locate its two hashed cells $\mathcal{B}_1[3]$ and $\mathcal{B}_2[4]$. Next, we check the blocks which the two hashed cells reside, and clean the outdated cells with value 3, *i.e.*, clean $\mathcal{B}_1[4]$ to zero. Since after cleaning, both the two hashed cells are not *zero*, we do not report a batch. Finally, we update $\mathcal{B}_1[3]$ and $\mathcal{B}_2[4]$ to s_{now}.

Error analysis: HyperBF might miss some batches, but the reported batches are always correct. The error of HyperBF comes from three aspects. **1)** The error incurred by hash collision, which is the cause of false positive error of Bloom filters. **2)** The error incurred by outdated cells that are not cleaned in time. **3)** The error incurred by coarse-grained timeline division. We provide the theoretical analysis of the three kinds of error in § 4, proving that the impact of the first two kinds of error are negligible. For the third error, essentially, our 2-bit time slice is a coarse-grained timeline division: the gain is extremely high memory efficiency, and the cost is the fuzzy perception of time. Fortunately, the error incurred by fuzzy perception of time can be significantly reduced by the following *Asynchronous Timeline* technique.

Asynchronous Timeline: HyperBF perceives time in a fuzzy way. When the interval between two adjacent batches is among $\mathcal{T} \sim 2\mathcal{T}$, HyperBF might not be able to report the second batch correctly, depending on the relative offset of the timeline. Specifically, only when the interval span three time slices can HyperBF be able to report the second batch. This issue is illustrated in Figure 4a. Although the time interval between the two occurrences of e_1 exceeds $\mathcal{T}$, HyperBF cannot correctly divide them into two batches because the interval span just two time slices. Therefore, when the current time slice is 1, time slice 3 is not outdated. To address this issue, we propose the *Asynchronous Timeline* technique. Our key idea is to use d different timeline offsets for the d arrays to enhance the ability of batch perception. In this way, as long as the interval spans three time slices in any one of the d timelines, HyperBF can perceive the second batch correctly. As shown in Figure 4b, after using the *Asynchronous Timeline* technique, the interval spans three time slices in the second array. In this example, HyperBF can correctly perceive the second batch. We derive theoretical guarantees for *Asynchronous Timeline* using linear programming model in Theorem 4.3 in § 4, proving that the time division error can be reduced by d times when using d evenly distributed timelines.



(a) Synchronous.



(b) Asynchronous.

Fig. 4: Optimization using *Asynchronous Timeline*

Analysis on computational cost: The insertion operation of HyperBF requires d hash calculations ($h_1(e), \cdots, h_d(e)$) and d memory accesses ($\mathcal{B}_1[h_1(e)], \cdots, \mathcal{B}_d[h_d(e)]$). For each memory access, HyperBF checks l cells within a block and cleans all outdated cells therein. We will see that this procedure can be accomplished with SIMD AVX-512 instructions without the need for looping (§ 3.6). Therefore, the time complexity of HyperBF is $O(d)$. In practice, we usually set d to a small value ($d = 8$ by default). Thus, the time complexity of HyperBF can be approximate to $O(1)$.

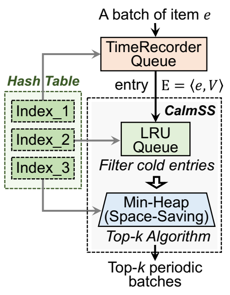
3.2 The TimeRecoder Algorithm

To record the last arrival time of batches, a strawman solution is to use a huge hash table to store the arrival time of the last batches for all items. This is memory inefficient, because most batches are not periodic. To address this issue, we propose TimeRecorder aiming to only store the time for those batches that are potential top-k periodic batches. The data structure of TimeRecorder is essentially a circular queue, which is implemented as a doubly linked list of c nodes. Each node records an item ID. We build the first hash table index (`Index_1`) for TimeRecorder. For each item e in the TimeRecorder queue, we store the arrival time $\hat{t}$ of its last batch in `Index_1`.

For each incoming batch of item e at time t, we first query `Index_1` to check whether the arrival time of its last batch is recorded. **1)** If so, we calculate the batch interval $V = t - \hat{t}$. Then we combine the item ID e and its batch interval V to form an entry $E = \langle e, V \rangle$, and send the entry to CalmSS. Finally, we update the timestamp of e to the current time t and move it to the front of the circular queue. **2)** If not, we insert e into the TimeRecorder queue, and store the arrival time t of its last batch in `Index_1`. If the TimeRecorder queue is already full before insertion, we evict the oldest (least recently accessed) item e_0 to make room for e. Note that if e_0 has periodic batches (*i.e.*, it is maintained in CalmSS), we still preserve the arrival time of its last batch in `Index_1`. For the implementation details, please see § 3.4.

Our TimeRecorder evicts the following items: **1)** Items that are old and do not show periodicity; and **2)** Items whose batches have long periods, which have little potential to become top-k periodic batches. The TimeRecorder keeps the items that are highly likely to have top-k periodic batches, and discard other items which are the major part of the data stream. Therefore, our TimeRecoder queue is much more memory efficient than the above strawman solution.

Analysis on computational cost: In the insertion operation of TimeRecorder, we first query `Index_1` (implemented as a hash table with $O(1)$ time complexity). If item e exists in `Index_1`, we update its last arrival time in `Index_1` and move it to the front of the circular queue. Notice that we can acquire the position of e in the circular queue from `Index_1`. If e is not in `Index_1`, we insert it into `Index_1` and the front of the circular queue, and evict the oldest item from the tail of the circular queue. All these operations can be accomplished with $O(1)$ time complexity, and thus the overall time complexity of TimeRecorder is also $O(1)$.



(a) Data structure.

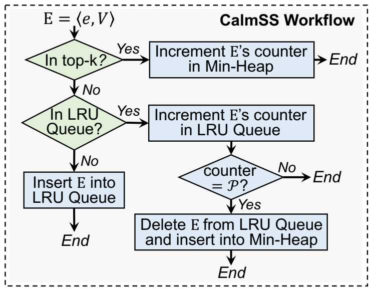
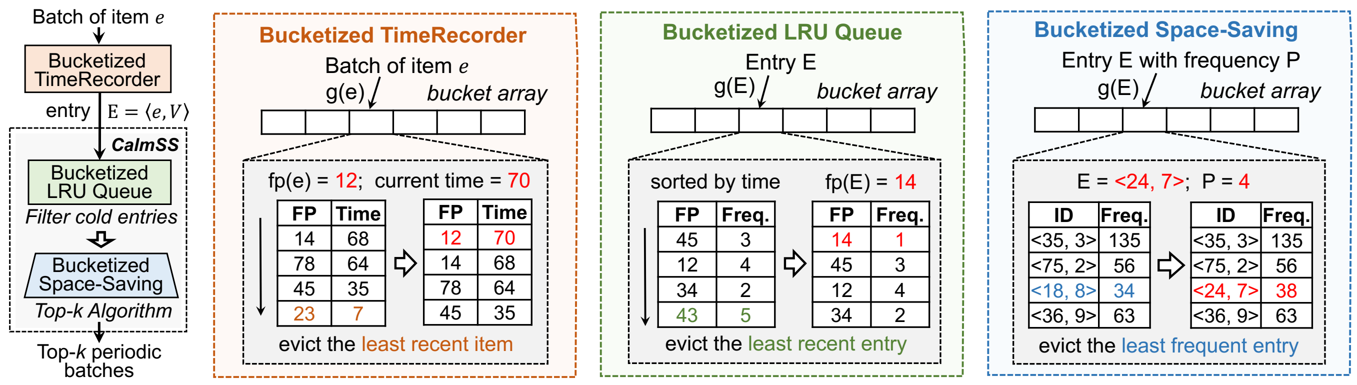

$\mathrm{E} = \langle e, V \rangle$
CalmSS Workflow
In top-k?
Yes
Increment E's counter in Min-Heap
End
No
In LRU Queue?
Yes
Increment E's counter in LRU Queue
No
Insert E into LRU Queue
End
counter $= \mathcal{P}$?
No
End
Yes
Delete E from LRU Queue and insert into Min-Heap
End
(b) Insertion workflow.

Fig. 5: Data structure and workflow of CalmSS.

3.3 The CalmSS Algorithm

Rationale: Phase 2 uses a top-k algorithm to report top-k periodic batches. The most well-known top-k algorithm is Space-Saving [12], which works by maintaining a Min-Heap of m bins. For each incoming entry $E_1 = \langle e_1, V_1 \rangle$, if it is in the heap, it increments its counter by one; otherwise, it updates one of the smallest bins (E_{min}, f_{min}) to ($E_1, f_{min} + 1$). In this way, each incoming entry increments a counter in Space-Saving. Recall that in phase 2, most entries are cold entries, which appear just several times. All increments by cold entries are unnecessary, and significantly increase the overestimation error. Therefore, we propose CalmSS to minimize the influence of cold entries. The key idea of CalmSS is to use a queue to discard cold entries. The queue records the frequency of entries in the sliding window. This queue follows the LRU strategy: the least recently visited cold entry will be discarded, and hot entries will be moved to Space-Saving. Specifically, each incoming entry is first inserted into the queue: if it appears too few times in the sliding window, it will be discarded; otherwise, it will be moved to Space-Saving. This *LRU Queue* can be considered as a guardian of Space-Saving to keep cold entries outside.

Data structure (Figure 5a): CalmSS consists of an LRU queue and a Space-Saving (it is essentially an Min-Heap): **1)** The LRU queue uses a sliding window of w bins to keep the recent w distinct entries. Each bin stores a key-value pair (E, f), where the key is an entry ID and the value is a small



(a) Overview. (b) TimeRecorder. (c) LRU Queue. (d) Space-Saving.

Fig. 6: Bucketized Partition Optimization ($b = 4$).

counter recording the frequency of E. The LRU queue uses a predefined threshold $\mathcal{P}$ (called promotion threshold) to filter out cold entries: Once the counter of an entry E reaches $\mathcal{P}$, it means it is not a cold entry, and thus we remove E from the LRU queue and insert it into the Space-Saving. **2)** The Space-Saving uses a Stream-Summary [12] and a hash table to achieve O(1) time complexity to locate and update the entries. Similar to the TimeRecoder, we build the second and third hash table indices (`Index_2` and `Index_3`) for the LRU queue and the Space-Saving.

Insert (Figure 5b): For each incoming entry $E = \langle e, V \rangle$, we first query it in the hash indices: **1)** If E is in the Space-Saving, we just increment its counter by one. **2)** If E is in the LRU queue, we increment the small counter of E in the LRU queue by one. After increment, if the small counter reaches the predefined promotion threshold $\mathcal{P}$, we remove E from the LRU queue and insert $(E, \mathcal{P})$ into the Space-Saving. Specifically, if the Min-Heap is already full before inserting E, we update the smallest node (E_{min}, f_{min}) in the Space-Saving to $(E, f_{min} + \mathcal{P})$. **3)** If E is not in the LRU queue, we insert $(E, 1)$ into the LRU queue. If the LRU queue is already full before inserting E, we evict the least recently accessed entry to make room for E.

Analysis on computational cost: In the insertion operation of CalmSS, we first query the hash indices (with $O(1)$ time complexity) to obtain the positions of entry E in Space-Saving and LRU queue. Subsequently, we update the frequency of E in either Space-Saving or LRU queue. If the frequency in the LRU queue exceeds threshold $\mathcal{P}$, we insert $(E, \mathcal{P})$ into Space-Saving. Notice that the update operation of Space-Saving has a time complexity of $O(1)$ [12]. Therefore, the time complexity of CalmSS is also $O(1)$.

Report: To report top-k periodic batches, CalmSS reports the k entries with the k largest frequencies in the Min-Heap. The time complexity of this procedure is $O(k)$. Note that one item could have multiple groups of periodic batches, and thus could be reported more than once.

3.4 Implementation

In our implementation, we combine the three hash indices (`Index_1`, `Index_2`, and `Index_3`) into one hash table index `Index_all`. For each key-value pair in the hash table `Index_all`, it includes one key (item ID) e and three values: **1)** A timestamp $\hat{t}$, which is the arrival time of the last batch of e; **2)** Two entry lists `List_1` and `List_2`, which record the corresponding entries of e (they are essentially some batch intervals of e) that are in the LRU queue and the Space-Saving (Min-Heap), respectively. Each node in the two entry lists uses a pointer to index the location of the LRU queue or the Space-Saving. **3)** A counter recording the sum of several parts: the number of appearances of e in the TimeRecorder, and the lengths of the two lists. We delete e from the hash table once its counter is decremented to zero. In this way, for all items that have periodic batches, their last batch arrival time is maintained in `Index_all` even if they are not in the TimeRecorder queue.

3.5 Optimization: Bucketized Partition

In the TimeRecorder and CalmSS algorithm, for each incoming item/entry, we need to first check whether the item/entry is recorded in TimeRecorder/CalmSS. In the basic version of HyperCalm, to accelerate the checking process, we build a hash table to index each item/entry with $O(1)$ time complexity. However, this data structure doubles the memory usage, and it is also time inefficient because of many pointer operations. To address this issue, we propose the Bucketized Partition optimization, which removes the hash table from our HyperCalm sketch while maintaining its $O(1)$ time complexity.

Data structure (Figure 6): As shown in Figure 6a, we remove the hash table from HyperCalm sketch. As shown in Figure 6b-6d, the data structure of optimized TimeRecorder/LRU-Queue/Space-Saving is an array of z buckets. We use a hash function $g(\cdot)$ to map each item/entry into one bucket in the array. Each bucket has b slots. In bucketized TimeRecorder (Figure 6b), each slot stores a 16-bit fingerprint of an item and a timestamp recording its last batch arrival time. In bucketized LRU Queue (Figure 6c), each slot stores a 16-bit fingerprint of an entry and a counter recording its frequency. In bucketized Space-Saving (Figure 6d), each slot stores an entry ID and a counter recording its frequency.

Insertion: As shown in Figure 6a, we process each item batch reported by HyperBF in a one-pass manner. For each batch of item e at time t, we first query the bucketized TimeRecorder to acquire its last batch arrival time $\hat{t}$ and calculate its batch interval $V = t - \hat{t}$. Then we send the entry $E = \langle e, V \rangle$ to bucketized CalmSS, which reports top-k frequent entries, *i.e.*, top-k periodic batches. Specifically, we first check whether E is recorded in bucketized Space-Saving. If so, we increment its frequency by one. Otherwise, we insert E into bucketized LRU Queue. Below we describe the insertion operation of the three algorithms.

1) Bucketized TimeRecorder: Each bucket in bucketized TimeRecorder works independently as a small circular queue. The items in each bucket are sorted in time order, *i.e.*, the first item is the most recent item and the last item is the least recent item. For each incoming batch of item e at time t, we first calculate hash function $g(e)$ to locate the $g(e)_{th}$ bucket (called the hashed bucket of e). Then we calculate a hash function to acquire the fingerprint $fp(e)$. If e is recorded in the hashed bucket (*i.e.*, $fp(e)$ is recorded in this bucket), we update its timestamp to t and move this item to the first slot. Otherwise, we evict the least recent item (*i.e.*, the last item in the bucket) if the bucket is full, and then insert $(fp(e), t)$.

Example (Figure 6b): We first calculate hash functions to locate the hashed bucket and acquire the fingerprint $fp(e) = 12$. Since $fp(e)$ is not recorded in the hashed bucket, we evict the least recent item $(23, 7)$ and then insert $(fp(e), t)$.

2) Bucketized LRU Queue: Similar to bucketized TimeRecorder, each bucket in bucketized LRU Queue works independently as a small LRU queue. The entries in each bucket are sorted in time order. For each incoming entry E, we first calculate hash function $g(E)$ to locate its hashed bucket and calculate hash function $fp(E)$ to acquire its fingerprint. If E is recorded in the hashed bucket, we increment its counter by one and move E to the first slot. If the counter reaches the promotion threshold $\mathcal{P}$ after increment, we remove E from the bucket and insert $(E, \mathcal{P})$ into Space-Saving. If E is not recorded in the hashed bucket, we evict the least recent entry, and insert $(fp(E), 1)$ into this bucket.

Example (Figure 6c): To insert entry E, we first calculate hash functions to locate its hashed bucket and acquire its fingerprint $fp(E) = 14$. Since $fp(E)$ is not recorded in the hashed bucket, we evict the least recent entry (*i.e.*, the last entry) and insert $(fp(E), 1)$ into this bucket. After insertion, the entries in this bucket are also kept in time order.

3) Bucketized Space-Saving: Each bucket in bucketized Space-Saving works independently as a small Space-Saving. For each incoming entry E and its frequency $\mathcal{P}$, we first calculate hash function to locate the hashed bucket. If E is recorded in this bucket, we increment its frequency by $\mathcal{P}$. Otherwise, we check the hashed bucket to find the slot recording the entry with the smallest frequency, *i.e.*, (E_{min}, f_{min}), and then update this slot to $(E, f_{min} + \mathcal{P})$.

Example (Figure 6d): To insert entry E with frequency $\mathcal{P}$, we first calculate hash function to locate its hashed bucket. Since E is not recorded in the hashed bucket, we evict the least frequent entry $E_{min} = \langle 18, 8 \rangle$ and insert $(E, 34 + \mathcal{P})$.

Report: To report top-k periodic batches, we traverse all buckets in bucketized Space-Saving, and reports the entries with top-k largest frequencies. The time complexity of this traversal procedure is $O(z \times b)$.

Discussion: Bucketized Partition optimization has the following advantages. First, bucketized HyperCalm sketch is more memory efficient because it removes the hash table index. In addition, it replaces the IDs in TimeRecorder and LRU Queue with compact fingerprints, which further saves memory. Second, bucketized HyperCalm sketch has faster insertion speed. It processes each item in a one-pass manner and has $O(1)$ time complexity. Third, bucketized HyperCalm sketch is cache-friendly and can be further accelerated using SIMD instructions [52] to achieve better data parallelism, which will be described in § 3.6.

Analysis on computational cost: In the insertion operation of Bucketized TimeRecorder/LRU-Queue/Space-Saving, we first calculate hash function to locate one hashed bucket. Then we update the timestamp or frequency of the item/entry within its hashed bucket. The entire insertion operation requires one hash calculation and one memory access. Therefore, the time complexity of Bucketized TimeRecorder/LRU-Queue/Space-Saving is $O(1)$.

3.6 Optimization: SIMD Acceleration

Single instruction, multiple data (SIMD) [52] is a widely-used data parallel processing technology that can perform the same operation on multiple data simultaneously. This technology well suits the data structure of the HyperCalm sketch using Bucketized Partition. Below we describe how to use SIMD instructions to accelerate HyperCalm.

HyperBF acceleration: In HyperBF, for each incoming item, we first locate d hashed blocks and clean all cells in these blocks. To ensure that outdated cells can be cleaned timely, the block size is set to 64B (cache line size). In our basic implementation, we use a loop to clean these cells, which can be accelerated using SIMD. Currently, AVX-512 instruction set provides 512-bit wide SIMD registers (ZMM). We can load a block into one 512-bit register and use 512-bit bit operations to efficiently clean this block, eliminating the complicated loops and improving efficiency.

Bucketized TimeRecorder/CalmSS acceleration: In bucketized TimeRecorder/LRU-Queue/Space-Saving, we treat the b slots as uniform data points and uses SIMD instructions to simultaneously process them in parallel. To ensure memory continuity, in each bucket, we record IDs (or fingerprints) and frequencies (or timestamps) in two arrays separately, which are called the ID array and the information array. In insertion process, we first check the ID array. If we find a matched ID/fingerprint, we update the corresponding frequency/timestamp. Otherwise, we check the information array to find the item/entry to be evicted (*e.g.*, the least recent item in TimeRecorder). We can use SIMD instructions to accelerate three processes: 1) finding mathed ID/fingerprint; 2) sorting items according to time order; 3) finding the minimum counter. The implementation details can be found in our supplementary materials [53].

3.7 Extension: Mining Periodic Large Batches

Motivation: In § 2.1, we define periodic batches without considering the size of batches. Under such definition, a group of periodic items is also treated as a group of periodic batches, even if each batch only has one item. However, in many applications, users are more interested in periodic batches with large sizes. For example, in the scenario of streaming database, many data processing requests arrive with periodicity. It is possible to identify periodic batches and predict the arrival time of each batch, so as to pre-allocate resources (*e.g.*, CPU and I/O) to handle the increased load more efficiently. In this scenario, it's more beneficial to prioritize pre-allocating resources for larger batches as opposed to smaller ones due to their higher demand for resources and their potential to impact system performance more significantly. Another example can be seen in the context of network traffic load balancing. Here, the network operator can identify periodic batches and schedule these batches to balance the load (*e.g.*, schedule each batch to the least loaded path). In this case, the operator only needs to schedule large batches and can neglect small batches, because only large batches have the potential to cause load imbalance [54].

In this subsection, we present the definition of periodic large batches and extend our HyperCalm sketch to identify them. Specifically, we extend HyperBF to report the size of each batch, and we only send the batches with large estimated sizes to the next phase. In this way, we maintain the information of periodic large batches in CalmSS.

Problem statement: A group of periodic large batches refers to α consecutive batches of the same item, where these batches arrive with a fixed time interval, and the size of each batch is larger than a predefined threshold $\mathcal{L}$. Similar to the definitions in § 2.1, we define α as the periodicity and define top-k periodic large batches as the k groups of periodic large batches with the k largest periodicities.

Data structure: We extend HyperBF to report batch size by combining it with a Count-Min (CM) sketch [27]. Consider a HyperBF with d arrays $\mathcal{B}_1, \cdots, \mathcal{B}_d$, each of which has m 2-bit cells. We build a CM sketch with d arrays $\mathcal{C}_1, \cdots, \mathcal{C}_d$, each of which has m counters. In this way, each 2-bit cell is associated with one counter in CM sketch, and we call this counter as the associated counter of the cell.

Insert: For each incoming item e arriving at time t, we insert it into the modified HyperBF as follows. First, we calculate hash functions to locate the d hashed cells $\mathcal{B}_1[h_1(e)], \cdots, \mathcal{B}_d[h_d(e)]$ and their d associated counters $\mathcal{C}_1[h_1(e)], \cdots, \mathcal{C}_d[h_d(e)]$. For each hashed cell, we check its block and clean the outdated cells according to the rule of basic HyperBF described in § 3.1. In this procedure, if a cell is cleaned to zero, we additionally clean its associated counter in the CM sketch to zero. Afterwards, we update all d hashed cells to the current time slice, and increment each of the d associated counters by one. We estimate the size of the current batch of e as the minimum values among the d counters $\mathcal{C}_1[h_1(e)], \cdots, \mathcal{C}_d[h_d(e)]$, and if the current estimated size exceeds the predefined large batch threshold $\mathcal{L}$ (*i.e.*, its estimated size reaches $\mathcal{L}+1$), we report a large batch of e to the next phase of HyperCalm sketch.

Report: We record periodic large batches and their periodicities in CalmSS. To report top-k periodic large batches, we reports the k entries with the k largest frequencies.

Discussion: Recall that in § 2.1, we define the arrival time of each batch as the arrival time of its first item, and we use this time to calculate the time interval between two adjacent batches. However, the extended HyperBF reports a batch to the next phase only when its estimated frequency reaches $\mathcal{L}+1$. This introduces a time lag in the estimation of the batch's arrival time. As different batches might have different item arrival speed, and the CM sketch has overestimation errors, the calculated batch interval might not be very accurate. However, our experimental results show that even with the time lag of batch arrival time and overestimation error of CM sketch, HyperCalm still has high accuracy in finding periodic large batches (>0.96 F1 score).

3.8 Extension: Dynamic Memory Adjustment

In practice, the density of data streams and the available memory resources might vary dynamically [55]. It is desirable to perform on-the-fly reconfiguration on the sketch size to adapt to these dynamic variations. Towards this goal, we propose the dynamic memory adjustment operations for HyperBF and Bucketized TimeRecorder/LRU-Queue/Space-Saving, by which we can dynamically compress and expand their sizes by any integer factor. These operations allow us to dynamically adjust the memory usage for HyperCalm sketch without losing the previously recorded information. The time complexity of the memory adjustment operations for HyperBF and Bucketized TimeRecorder/LRU-Queue/Space-Saving are $O(m)$ and $O(z)$ respectively.

Dynamic memory adjustment on HyperBF: We introduce the dynamic memory adjustment operations of HyperBF to expand/compress its size by any integer factor. 1) To **expand** the size of HyperBF by r times, for each of its array $\mathcal{B}_i$, we perform the memory copy operation to copy its m cells by r times and get $\mathcal{B}'_i$ (with $m' = m \times r$ cells). Then we modify hash function $h_i(\cdot) = \mathcal{H}(\cdot)\%m$ to $h'_i(\cdot) = \mathcal{H}(\cdot)\%m'$. Notice that $h'_i(\cdot) \in \{h_i(\cdot), h_i(\cdot)+m, \cdots, h_i(\cdot)+(r-1)\times m\}$. Therefore, the time information recorded in $\mathcal{B}_i[h_i(e)]$ can still be retrieved in $\mathcal{B}'_i[h'_i(e)]$ after expansion. 2) To **compress** the size of HyperBF by r times, for each of its array $\mathcal{B}_i$, we first split its cells into r groups, each of which has m/r cells. Then we merge every r cells with the same index in the r groups into one cell and get $\mathcal{B}'_i$ (with $m' = m/r$ cells). For example, we merge $\mathcal{B}_i[0], \mathcal{B}_i[\frac{m}{r}], \cdots, \mathcal{B}_i[(r-1)\times\frac{m}{r}]$ to $\mathcal{B}'_i[0]$, merge $\mathcal{B}_i[1], \mathcal{B}_i[\frac{m}{r}+1], \cdots, \mathcal{B}_i[(r-1)\times\frac{m}{r}+1]$ to $\mathcal{B}'_i[1]$, *etc.* The merging operation is performed by taking the most recent state among the r cells. We set the resulting cell to 0 if the r cells are all outdated. Finally, we modify hash function $h_i(\cdot) = \mathcal{H}(\cdot)\%m$ to $h'_i(\cdot) = \mathcal{H}(\cdot)\%m'$. As m is divisible by m', we have $h_i(\cdot)\%m' = h'_i(\cdot)$. Therefore, the time information recorded in $\mathcal{B}_i[h_i(e)]$ will be merged to $\mathcal{B}'_i[h'_i(e)]$ after compression.

Examples (Figure 7): We take the first array $\mathcal{B}_1$ as an example to further illustrate the procedure of expanding/compressing a HyperBF by $r=2$ times. 1) In the expansion operation, we just copy $\mathcal{B}_1$ by $r=2$ times to get $\mathcal{B}'_1$, where $\mathcal{B}'_1[0] = \mathcal{B}'_1[4]$, $\mathcal{B}'_1[1] = \mathcal{B}'_1[5]$, *etc.* 2) In the compression operation, we first split $\mathcal{B}_1$ into $r=2$ groups. Then we merge every $r=2$ cells with the same index in the two groups to get $\mathcal{B}'_1$. Specifically, we merge $\mathcal{B}_1[0]$ and $\mathcal{B}_1[4]$ to get $\mathcal{B}'_1[0]$, which is set to the more recent state $\mathcal{B}_1[4]=2$. Similarly, we merge $\mathcal{B}_1[1]$ and $\mathcal{B}_1[5]$ to get $\mathcal{B}'_1[1]$, *etc.* For $\mathcal{B}'_1[3]$, as both $\mathcal{B}_1[3]$ and $\mathcal{B}_1[7]$ are outdated, we set it to 0.

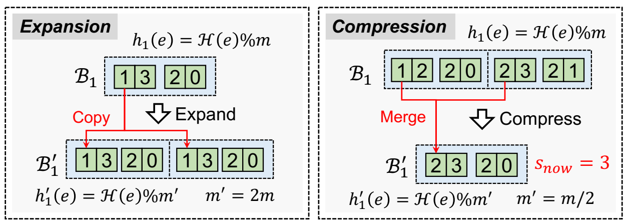


Fig. 7: Dynamic memory expansion/compression on HyperBF ($m = 4/8$, $l = 2$, $r = 2$, current state $s_{now} = 3$).

Dynamic memory adjustment on Bucketized TimeRecorder/ LRU-Queue/Space-Saving: Similarly, we introduce the dynamic memory adjustment operations of Bucketized TimeRecorder/LRU-Queue/Space-Saving. Below we take a bucket array $\mathcal{A}$ with z buckets to explain the expansion/compression operations. 1) To **expand** the memory by r times, we perform the memory copy operation to copy the z buckets by r times and get $\mathcal{A}'$ (with $z' = z \times r$ buckets). We modify hash function $g(\cdot) = \mathcal{G}(\cdot)\%z$ to $g'(\cdot) = \mathcal{G}(\cdot)\%z'$. For bucketized Space-Saving, when executing the memory copy operation, for each entry E in bucket $\mathcal{A}'[i]$, we compute hash function to check whether it should be retained in $\mathcal{A}'[i]$. Specifically, we check whether $g'(E) = i$, and if not, we remove E from $\mathcal{A}'[i]$. As discussed above, this design guarantees that the information of item e (or entry) in $\mathcal{A}[g(e)]$ can be retrieved in $\mathcal{A}'[g'(e)]$ after expansion. 2) To **compress** the memory by r times, we also split $\mathcal{A}$ into r groups, each of which has z/r buckets. We merge every r buckets with the same index in the r groups into one bucket and get $\mathcal{A}'$ (with $z' = z/r$ buckets). In the merging operation, we preserve the following b fingerprints/entries. For bucketized TimeRecorder, we preserve the most recent b fingerprints. For bucketized LRU-Queue, we approximately select the most recent b fingerprints in a round-robin fashion among the r bucket and preserve them. For bucketized Space-Saving, we preserve the most frequent b entries. This design guarantees that the information of item e (or entry) in $\mathcal{A}[g(e)]$ can only be stored in $\mathcal{A}'[g'(e)]$ after compression.

Examples (Figure 8): We take bucketized Space-Saving as an example to illustrate the procedure of expanding/compressing a bucket array $\mathcal{A}$ by $r = 2$ times. 1) In the expansion operation, we copy $\mathcal{A}$ by $r = 2$ times to get $\mathcal{A}'$. For each entry E, we calculate hash function $g'(E)$ to check whether it should be retained in its bucket. As $g'(\langle 35, 3\rangle) = 3$, we delete $\langle 35, 3\rangle$ from $\mathcal{A}'[0]$. Similarly, as $g'(\langle 75, 2\rangle) = 0$, we delete $\langle 75, 2\rangle$ from $\mathcal{A}'[3]$. 2) In the compression operation, we split $\mathcal{A}$ into $r = 2$ groups and merge every two buckets with the same index to get $\mathcal{A}'$. For example, when merging $\mathcal{A}[0]$ and $\mathcal{A}[3]$ to get $\mathcal{A}'[0]$, we preserve the $b = 2$ entries with the largest frequencies.

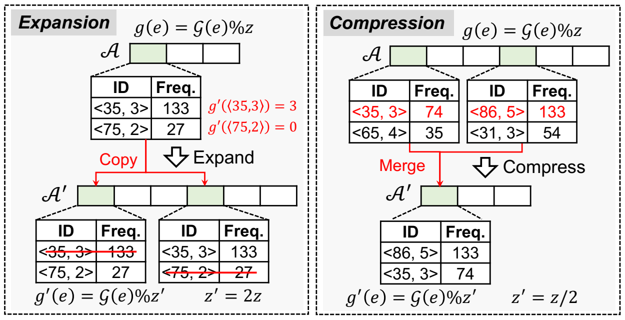


Fig. 8: Dynamic memory expansion/compression on Bucketized Space-Saving ($z = 3/6$, $b = 2$, $r = 2$).

4 Mathematical Analysis

In this section, we provide theoretical analysis for HyperCalm sketch, and validate our theoretical analyses using experiments. We focus on the following four issues.

- **How accurate can HyperBF detect batches?** We derive the error bound of HyperBF in Lemma 4.4 and Theorem 4.1, and conduct experiments to validate our bound in Figure 10b. The results show that both theoretical and experimental error are smaller than 0.01 in common cases.
- **How accurate can CalmSS detect top-k periodic batches?** We derive the error bound of CalmSS in Theorem 4.2, and conduct experiments to validate our bound in Figure 11. The results show that both theoretical and experimental error rate are smaller than 0.01 in common cases.
- **Is the Asynchronous Timeline technique of HyperBF effective?** We theoretically analyze the accuracy gain of *Asynchronous Timeline* technique in Theorem 4.3, and conduct experiments to validate it in Figure 12. Both theoretical and experimental results show that *Asynchronous Timeline* technique improves the accuracy of HyperBF.
- **How accurate can bucketized Space-Saving report the periodicity of periodic batches?** We derive the error bound of bucketized Space-Saving in Theorem 4.4 and Theorem 4.6, and conduct experiments to validate the theoretical bounds.

4.1 Error Rate of HyperBF

We first prove the error rate of HyperBF in Theorem 4.1. A data stream can be formulated by two variables: density α and activity β, where density α is the number of distinct items observed at each moment, and activity β is the number of distinct items emerging/dying per unit time. Consider two consecutive time interval T_1 and T_2. The numbers of distinct items observed in T_1 and T_2 are $\alpha + \beta T_1$ and $\alpha + \beta T_2$, respectively. And the number of distinct items observed in the two intervals is $\alpha + \beta(T_1 + T_2)$. Most data streams can be formulated by these two variables. Take CAIDA [56] dataset as an instance, Figure 10a shows the average number ($\pm$5std) of distinct items observed in time intervals of different length. We can see that the linear relationship almost holds where $\alpha = 3195.2$ and $\beta = 35238.9$. Next, consider two adjacent occurrences of item e at t_1 and t_2, where $t_2 - t_1 > 2\mathcal{T}$. Let $K = \lfloor \frac{t_2}{\mathcal{T}} \rfloor - \lfloor \frac{t_1}{\mathcal{T}} \rfloor$. Let $\gamma_n = \alpha + \beta n \mathcal{T}$ denote the number of distinct items observed in a time interval of length $n\mathcal{T}$.

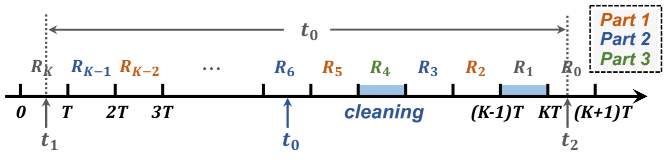


Fig. 9: Error rate analysis of HyperBF.

As shown in Figure 9, consider two adjacent occurrences of an item e in the data stream. Assume the timestamps of the two occurrences are t_1 and t_2, respectively. Assume $t_2 - t_1 > 2\mathcal{T}$, meaning that the second occurrence of e is the start of a batch and there is no *time division error*. Next, we derive the *error rate* of HyperBF in Theorem 4.1 (and Lemma 4.4), which is defined as the probability that HyperBF does not report a batch at t_2. The detailed proofs are provided in our supplementary materials [53].

Lemma 4.4. *Let P be the probability that a certain hashed cell of item e (e.g., $\mathcal{B}_i[h_i(e)]$) is zero at t_2. Let $m' = \frac{m}{l-1}$ and $u_j = \left\lceil \frac{j-2}{3} \right\rceil$. Let $K_1 = \lfloor \frac{K}{3} \rfloor - 1$, $K_2 = \lfloor \frac{K-1}{3} \rfloor - 1$, and $K_3 = \lfloor \frac{K-2}{3} \rfloor - 1$. Then the lower bound of P is $P' = P'_1 + P'_2 + P'_3$, where $P'_1 = \sum_{k=0}^{K_1} \left(e^{-\frac{\gamma_{3k+2}}{m}} - e^{-\frac{\gamma_{3k+3}}{m}} \right)$,*

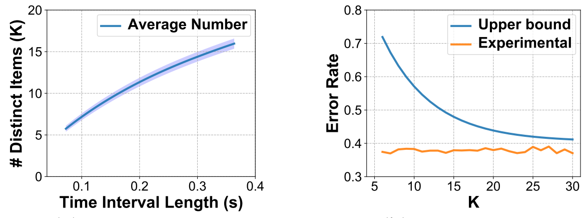
$P'_2 = \sum_{k=0}^{K_2}\left(e^{-\frac{\gamma_{3k+3}}{m}} - e^{-\frac{\gamma_{3k+4}}{m}}\right)\left(1-e^{-\frac{\gamma_{u_{3k+3}}}{m'}}\right)$, *and*
$P'_3 = \sum_{k=0}^{K_3}\left(e^{-\frac{\gamma_{3k+4}}{m}} - e^{-\frac{\gamma_{3k+5}}{m}}\right)\left(1-e^{-\frac{\gamma_{u_{3k+4}}}{m'}}\right)$.

Theorem 4.1. *We define the error rate $\mathcal{E}$ of HyperBF (without Asynchronous Timeline) as the probability that HyperBF does not report a batch at t_2. Then we have:*

$$\mathcal{E} \leqslant (1-P')^d$$

where P' is the lower bound in Lemma 4.4.

Experimental analysis (Figure 10b): We conduct experiments on CAIDA [56] to validate the bound in Lemma 4.4. We use the HyperBF that just has one array ($d = 1$), and allocate 4KB of memory to it ($m = 16000$). The results show that the experimental error rate is always well bounded by theoretical bound. As the volume of CAIDA data stream is very large, almost all outdated cells in HyperBF can be cleaned promptly. Therefore, the experimental error rate does not vary with K. As K grows larger, our theoretical bound becomes more accurate. Note that we only focus on a single array of HyperBF here. If we use the HyperBF consisting of $d = 8$ arrays, the error rate will be < 0.01.



(a) Data stream. (b) Error rate.

Fig. 10: Error rate of HyperBF.

4.2 Error Rate of CalmSS

We define the error rate ζ of CalmSS as the probability that a cold item fails to be discarded by LRU queue, *i.e.*, the probability that a cold item enters the top-k algorithm in CalmSS. Next, we derive the upper bound of ζ.

We assume the data stream consists of two types of items: cold items and hot items, and all items of the same type have the same arrival speed. The data stream is essentially the sum of many independent Poisson processes of two kinds (hot items and cold items). Let λ_h and λ_c be the parameters of the two Poisson processes, respectively. Let n_h and n_c be the number of distinct hot items and cold items, respectively. Notice that $n_h \gg w$ and $n_c \gg w$. Therefore, we can assume that in a short time interval, all arriving items are distinct. Consider a cold item e, we assume all items that arrives between the time when e enters the LRU queue and the time when e is removed from the LRU queue are distinct *hot* items. Here, we assume all of these items are hot because we want to derive an upper bound of ζ. Cold items only promote the LRU queue to discard e, resulting in a smaller ζ. We derive the error upper bound of CalmSS in Theorem 4.2. The detailed proofs are provided in our supplementary materials [53].

Theorem 4.2. *For a cold item e, the probability ζ that it fails to be discarded by CalmSS, i.e., the error rate of CalmSS, is*

$$\zeta = \left(\sum_{x=0}^{w-1}\frac{1}{x!}\cdot\frac{R^x}{(R+1)^{x+1}}\cdot\Gamma(x+1)\right)^{\mathcal{P}-1}$$

where $R = \frac{n_h\lambda_h}{\lambda_c}$, and $\Gamma(z)$ represents the Gamma function.

Experimental analysis (Figure 11): We conduct experiments to validate our theoretical bound in Theorem 4.2. We set $w = 16$, $\mathcal{P} = 4$, and generate the data stream using two kinds of Poisson processes where $n_h = 50$ and $n_c = 1$. The results show that the experimental error rate is always bounded by the theoretical upper bound. Note that when $R < 50$, the intensity of cold items λ_c is smaller than the intensity of hot items λ_h, meaning that cold items are actually not cold. Therefore, when R is small, CalmSS has large theoretical and experimental error. As R increases, our data stream assumption will be closer to truth. When $R > 50$, $\lambda_c < \lambda_h$, meaning that the cold items are really cold. When $R = 125$, the theoretical error rate is 10^{-2} and the experimental error rate is 10^{-4}, showing that CalmSS is highly effective in filtering cold items in real cases.

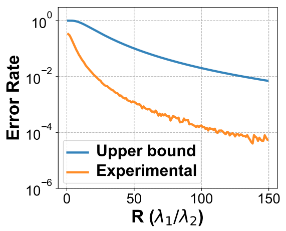


Fig. 11: Error of CalmSS.

4.3 Effectiveness of Asynchronous Timeline

Theorem 4.3. *After using the Asynchronous Timeline technique, the time division error is minimized when the d timelines are evenly distributed, i.e., when the timeline offset for the i^{th} array is $o_i = \frac{(i-1)}{d}\mathcal{T}$, where the minimized error is reduced by d times compared to the synchronous version.*

Detailed proofs are in our supplementary materials [53].

Experimental analysis (Figure 12): We conduct experiments on CAIDA [56] to validate Theorem 4.3. We set the batch threshold $\mathcal{T}$ to 1.454 μs, and fix the memory usage of HyperBF to 50KB. We find that *Asynchronous Timeline* technique significantly improves the accuracy of HyperBF. We also find that when using *Asynchronous Timeline*, HyperBF using d evenly distributed timelines is more accurate than HyperBF using d randomly distributed timelines. For example, when using $d = 8$ arrays, the RR of the basic HyperBF is 61%, while that of the HyperBF using *Asynchronous Timeline* is about 80%. Specifically, the RR of the HyperBF using randomly distributed timelines is 80.07%, while that of the HyeprBF using evenly distributed timelines is 81.95%.

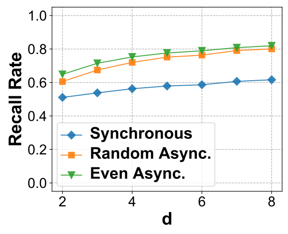


Fig. 12: Time division error of HyperBF.

4.4 Error Rate of Bucketized Space-Saving

We theoretically analyze the error of bucketized Space-Saving in estimating the periodicities of periodic batches. We first prove bucketized Space-Saving inherits the property of basic Space-Saving in Theorem 4.4. Then we derive an error bound that is related to the parameters of bucketized Space-Saving in Theorem 4.6 The detailed proofs are provided in our supplementary materials [53].

Theorem 4.4. *For an arbitrary entry E that is recorded in bucketized Space-Saving, let f and $\hat{f}$ be the real and estimated* *frequency of E respectively (i.e., f and $\hat{f}$ are the real and estimated periodicities of periodic batch E). We have that*

$$0 \leqslant \hat{f} - f \leqslant f_{min}$$

where f_{min} is the smallest counter in the hashed bucket of E.

Theorem 4.5. *For an arbitrary entry E with real frequency $f > \gamma||f||_1$, let $\mathcal{P}$ be the probability that E is recorded in bucketized Space-Saving. We have that*

$$\mathcal{P} \geqslant 1 - \frac{1}{b\gamma z}$$

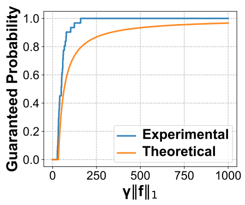
where $||f||_1$ is the frequency sum of all entries, b is the number of slots in each bucket, and z is the number of buckets.

Theorem 4.6. *For an arbitrary entry E that is recorded in bucketized Space-Saving, let f and $\hat{f}$ be the real and estimated frequency of E respectively. We have that*

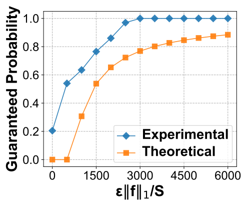
$$\Pr\left(\hat{f} - f \leqslant \epsilon \cdot \frac{||f||_1}{S}\right) \geqslant 1 - \frac{1}{\epsilon}$$

where $||f||_1$ is the frequency sum of all entries, and $S = zb$ is the number of slots in bucketized Space-Saving.

Experimental analysis (Figure 13): We conduct experiments using CAIDA [56] dataset to validate the theoretical bounds in Theorem 4.5 and Theorem 4.6, where we set $b = 16$. Figure 13a shows the experimental and theoretical probability in Theorem 4.5. We can see that the experimental probability is well bounded by the theoretical bound, and when $\gamma||f||_1 > 500$, both the experimental and theoretical probability are larger than 90%. Figure 13a shows the experimental and theoretical guaranteed probability in Theorem 4.6. We can see that the experimental guaranteed probability is well bounded by the theoretical probability.



(a) Theorem 4.5.



(b) Theorem 4.6.

Fig. 13: Accuracy of Bucketized Space-Saving.

5 Experimental Results

We conduct extensive experiments to validate the effectiveness of HyperCalm and its benefits to real-world applications. Our experiments focus on the following five issues.

- **Can HyperBF accurately and efficiently detect batches?** We compare HyperBF with SOTA Clock-Sketch [2], SWAMP [26], and Time-Out Bloom filter [25]. The results show that the accuracy and speed of HyperBF always outperform SOTA under the same memory usage. (§ 5.1)
- **Can CalmSS accurately and efficiently detect top-k items?** We compare CalmSS with SOTA Space-Saving [12], Unbiased Space-Saving [13], and Cold filter [17]. The results show that the accuracy and speed of CalmSS always outperform SOTA under the same memory. (§ 5.2)
- **Can HyperCalm accurately and efficiently detect periodic batches?** We combine the SOTA algorithms in detecting batches and finding top-k items to form one strawman solution for finding periodic batches, and compare HyperClam against it. The results show that HyperCalm outperforms the strawman solutions 4× in term of average relative error and 98.1× in term of speed. (§ 5.3)
- **Can periodic batches benefit real-world application?** We apply the HyperCalm sketch to two applications: cache (§ 5.8) and network measurement (§ 5.9). The results show that HyperCalm can well improve the hit rates of LFU/LRU caches, and it achieves high accuracy in detecting network anomalies.
- **Can HyperCalm work well in mainstream streaming framework and database?** We implement HyperCalm on top of Apache Flink [23] (§ 5.10) and Redis [24] (§ 5.11), showing that HyperCalm can be easily deployed into popular streaming processing framework and KV database.

We implement HyperCalm sketch and the other algorithms with C++. We use three datasets: small-scale CAIDA dataset with 30M items (default), large-scale CAIDA dataset with 1.5G items, Criteo dataset with 45M items. For more details about the *platform, setting, datasets, and metrics*, please refer to our supplementary materials [53].

5.1 Experiments on HyperBF

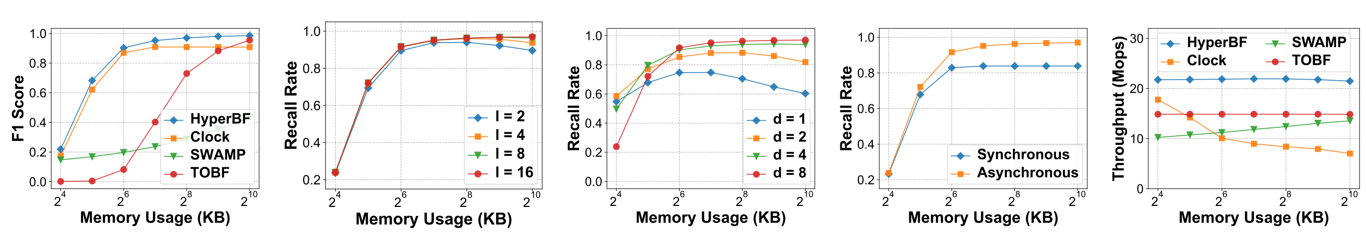
Parameter setting: We compare HyperBF with Clock-Sketch [2], SWAMP [26], and Time-Out Bloom filter (TOBF) [25]. We set $d = 8$ and $l = 32$ by default. For CAIDA, we set the time-based batch threshold $\mathcal{T}$ to 0.72 seconds. For Criteo, we set the count-based batch threshold $\mathcal{T}$ to 40,000. Under such settings, there are about 0.96M batches in CAIDA dataset, and about 4.9M batches in Criteo dataset.

Accuracy of detecting batches (Figure 14a): *We find that HyperBF always achieves the best accuracy.* In fact, HyperBF, Clock, and TOBF always have 100% PR, but HyperBF achieves better RR than Clock and SWAMP. SWAMP always has 100% RR because it reports all unrecorded items as batches, but its PR is less than 40% as it suffers high false positive errors. When using 256KB of memory, HyperBF achieves 97% F_1 score, significantly outperforms Clock (90%), SWAMP (28%), and TOBF (73%).

Impact of cell line size (l) (Figure 14b): *We find that a larger value of cell line size l goes with higher RR of HyperBF, and when the cell line size exceeds 8, HyperBF achieves the optimal accuracy.* When setting $l = 2$ and using more than 256KB of memory, the RR of HyperBF decreases as the memory usage increases because the outdated cells are not cleaned in time. The two curves of $l = 8$ and $l = 16$ are highly in coincidence, meaning that $l = 8$ is already enough to achieve the optimal accuracy.

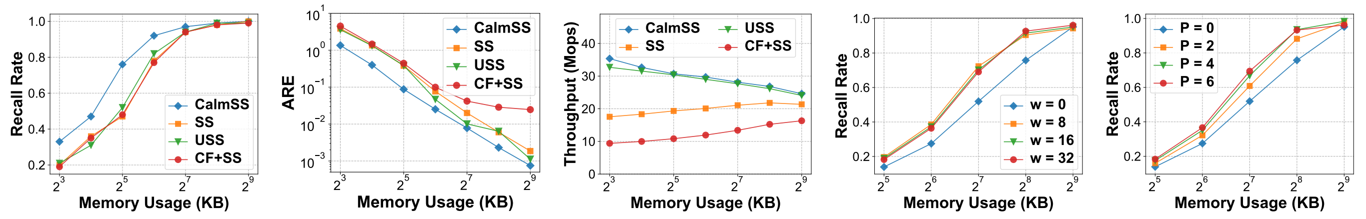
Impact of number of arrays (d) (Figure 14c): *We find that HyperBF performs well when using $d = 4$ or $d = 8$ arrays.* When the memory usage is small, smaller d goes with higher RR. This is because when the total memory usage is fixed, smaller d leads to larger size of each array, and thus leads to less hash collisions in each array. When the memory usage is large, larger d goes with higher Recall Rate. This is because if the array size is too small, the outdated cells cannot be cleaned in time, which compromises the accuracy of HyperBF. When setting $d = 8$ and using 256KB of memory, the RR of HyperBF exceeds 95%.

Impact of *Asynchronous Timeline* (Figure 14d): *We find that **Asynchronous Timeline** can significantly improve the RR*



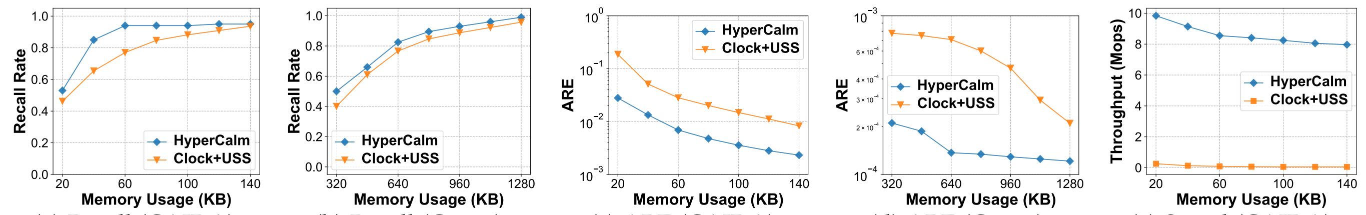
(a) F1 score. (b) Cell line size (l). (c) # arrays (d). (d) Timeline distr. (e) Processing speed.

Fig. 14: Performance of HyperBF (CAIDA).



(a) Recall. (b) ARE. (c) Processing speed. (d) Queue length (w). (e) Promotion thld. ($\mathcal{P}$).

Fig. 15: Performance of CalmSS (CAIDA).



(a) Recall (CAIDA). (b) Recall (Criteo). (c) ARE (CAIDA). (d) ARE (Criteo). (e) Speed (CAIDA).

Fig. 16: Performance of HyperCalm (CAIDA and Criteo).

of HyperBF. Here, the *Asynchronous Timeline* technique uses d evenly distributed timelines. When using 256KB of memory, HyperBF using *Asynchronous Timeline* achieves 97% RR, significantly outperforms that of the basic version (82%).

Processing speed (Figure 14e): *We find that HyperBF is faster than other algorithms.* The results show that under different memory constraints, the throughput of HyperBF is always 21 Mops, while that of TOBF and SWAMP are about 15 Mops and 12 Mops, respectively. The throughput of Clock drops rapidly with the increase of memory usage because when using more memory, Clock needs to clean more cells per insertion. When using 1024KB of memory, the throughput of Clock is only one-third of that of HyperBF.

5.2 Experiments on CalmSS

Parameter setting: We compare CalmSS with Space-Saving (SS) [12], Unbiased Space-Saving (USS) [13], and Cold filter [17] + Space-Saving (CF+SS). For CalmSS, we set $w = 16$ and $\mathcal{P} = 4$ by default. We set $k = 100$ and conduct the experiments using CAIDA.

Accuracy of finding top-k items (Figure 15a): *We find that CalmSS always has better RR than SS, USS, and CF+SS.* The RR of CalmSS reaches 78% even if the memory size is only 32KB, while that of SS and USS are about 50%. As the memory size exceeds 128KB, the RR of CalmSS is very close to 100%. The RR of CF+SS is smaller than ours because the large volume of data stream fill it up very quickly.

Frequency estimation for top-k items (Figure 15b): *We find that CalmSS always achieves smaller ARE than SS, USS, and CF+SS.* When using 32KB of memory, the ARE of CalmSS is 0.1, about 4 times lower than that of the other algorithms. When using 512KB of memory, the ARE of CalmSS is 7.5×10^{-4}, while that of SS, USS, CF+SS are 1.8×10^{-3}, 1.1×10^{-3}, and 2.1×10^{-2}, respectively.

Processing speed (Figure 15c): *We find that CalmSS is slightly faster than USS and significantly faster than SS and CF+SS.* CF+SS is slow because Cold filter needs extra memory accesses and hash computation. Surprisingly, CalmSS is faster than SS and USS because it sends only hot items to Space-Saving, resulting in fewer memory accesses to Space-Saving.

Impact of LRU queue length (w) (Figure 15d): *We find that CalmSS performs well when the length of the LRU queue w is just 8.* When using 256KB memory, the RR of CalmSS using an LRU queue of length $w = 8$ is 91%, while that of Space-Saving ($w = 0$) is 77%. Since the three curves of $w = 8$, $w = 16$, and $w = 32$ are highly in coincidence, we conclude that $w = 8$ is enough to achieve satisfactory accuracy.

Impact of *promotion threshold* ($\mathcal{P}$) (Figure 15e): *We find that the optimal promotion threshold $\mathcal{P}$ is 4 or 6.* When using 256KB memory, the RR of CalmSS with $\mathcal{P} = 2$ or $\mathcal{P} = 4$ is about 92%, while that of Space-Saving ($\mathcal{P} = 0$) is 76%. Note that the optimal $\mathcal{P}$ is highly correlated with the dataset.

5.3 Experiments on HyperCalm

Parameter setting: We combine the state-of-the-art Clock-Sketch and Unbiased Space-Saving to form a strawman solution for finding top-k periodic batches (Clock+USS), and compare our HyperCalm with it. The parameters (including memory proportion) of HyperCalm and the strawman solution are empirically set so that they achieve relatively good performance. For HyperBF, we set $d = 8$ and $l = 32$. For CalmSS, we set $\mathcal{P} = 7$. We set $k = 200$ by default.

***1) Setting on CAIDA*:** We set the time-based batch threshold $\mathcal{T}$ to 0.072 millisecond. Each batch interval V is rounded to the nearest multiple of 0.72 millisecond. Under such settings, there are about 4.1M periodic batches in CAIDA.

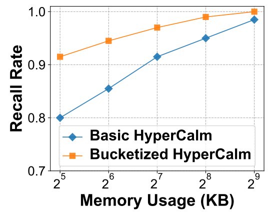
***2) Setting on Criteo*:** We set the count-based batch threshold $\mathcal{T}$ to 20,000. Each batch interval V is rounded to the nearest multiple of 100,000. Under such settings, there are about 14.7M periodic batches in Criteo dataset.

Accuracy of finding periodic batches (Figure 16a-16b): *We find that the RR of HyperCalm always outperforms the strawman solution on two datasets.* On CAIDA, when using 60KB of memory, the RR of HyperCalm is 94%, while that of the strawman solution is 78%. On Criteo, when using 800KB of memory, the RR of HyperCalm is 90%, while that of the strawman solution is 85%.

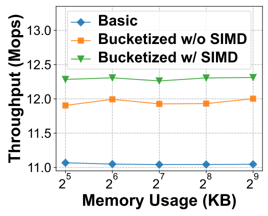
Periodicity estimation of periodic batches (Figure 16c-16d): *We find that HyperCalm always has smaller ARE than the strawman solution on two datasets.* On CAIDA, when using 60KB of memory, the ARE of HyperCalm is about 6.9×10^{-3}, which is 4 times lower than that of the strawman solution. On Criteo, when using 800KB of memory, the ARE of HyperCalm is about 1.3×10^{-4}, which is 4.6 times lower than that of the strawman solution.

Processing speed (Figure 16e): *We find that the processing speed of HyperCalm always outperforms the strawman solution on two datasets.* When using 60KB of memory, the throughput of HyperCalm is 8.54 Mops, which is 98.1 times higher than that of the strawman solution. The gap between HyperCalm and Clock+USS is huge because Clock needs to clean many cells per insertion, which is very inefficient.

Experiments on large-scale dataset (Figure 17): *We find that on large-scale dataset, HyperCalm still has high accuracy in finding periodic batches and fast insertion speed.* In this experiment, we use the 1-hour CAIDA dataset with 1.5G items, and build larger HyperCalm sketch to detect periodic batches. We can see that when using 512KB memory, HyperCalm achieves 99% RR and 12.4 Mops throughput.



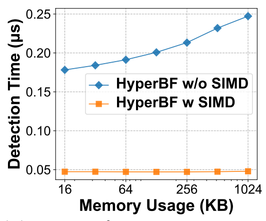
(a) Recall.



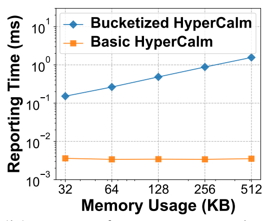
(b) Throughput.

Fig. 17: Performance of HyperCalm on large-scale dataset.

Time for detecting batches and periodic batches (Figure 18): We evaluate the time for HyperBF to detect item batches (Figure 18a) and the time for HyperCalm to report periodic batches (Figure 18b). As shown in Figure 18a, the reaction time for HyperBF to detect batches is always < 0.25 microsecond. For example, when using 64KB memory, the detection time for HyperBF with/without SIMD acceleration is only 0.047/0.191 microsecond. The detection time of HyperBF is very short because HyperBF processes items in a one-pass manner with $O(1)$ time complexity, which is very efficient. As shown in Figure 18b, the time for HyperCalm to report top-k ($k = 200$) periodic batches is always < 1.6 millisecond. For example, when using 256KB memory, the reporting time for HyperCalm with/without the Bucketized Partition Optimization is 0.879/0.0034 millisecond. The reporting time for Bucketized HyperCalm is longer because it needs to traverse the entire bucket array with $O(z \times b)$ time complexity to select the top-k entries (periodic batches). By contrast, basic HyperCalm can directly report the most frequent k entries from the Stream-Summary structure (a doubly-linked list sorted by frequency) with $O(k)$ time complexity. In summary, our framework achieves microsecond-level batch detection and millisecond-level periodic batch reporting, which it very efficient.



(a) Time for HyperBF to detect item batch.



(b) Time for HyperCalm to report periodic batch.

Fig. 18: Time for detecting batches and periodic batches.

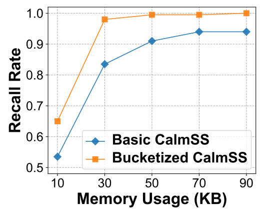
Performance improvement in statistical significance tests (Table 2): We add statistical tests to show the significance of our HyperBF and HyperCalm on detecting batches and periodic batches over existing methods. 1) For batch detection task, we fix the memory usage of HyperBF and existing methods (Clock [2], SWAMP [26], TOBF [25]) to 32KB, and repeat the experiments 100 times to collect 100 sets of paired samples on F1 Score and Throughput. Next, we use these samples to conduct Wilcoxon signed-rank test [57]. We define our one-sided alternative hypothesis such that the difference between the paired samples (*e.g.*, the F1-Score/Throughput difference between HyperBF and Clock) is stochastically greater than a distribution symmetric about Δ. We carry out multiple tests and use binary search to find the maximum performance improvement Δ satisfying the condition of `p-value` < 0.005, which are displayed in Table 2. The results show that HyperBF demonstrates a significant improvement (`p-value` < 0.005) in both F1 Score (up to +81.85%) and Throughput (up to +11.99 M/s) compared to existing methods. 2) For periodic batch detection task, we fix the memory usage of HyperCalm (with Bucketized Partition Optimization) and existing method (Clock+USS) to 32KB. Similarly, we collect 100 sets of samples on Recall and Throughput, and carry out multiple Wilcoxon signed-rank tests to find the maximum performance improvement Δ satisfying `p-value` < 0.005. The results show that HyperCalm demonstrates a significant improvement (`p-value` < 0.005) in both Recall (+43.75%) and Throughput (+9.18 M/s) compared to existing method (Clock+USS).

TABLE 2: Maximum performance improvement (Δ) satisfying the condition of `p-value`< 0.005.

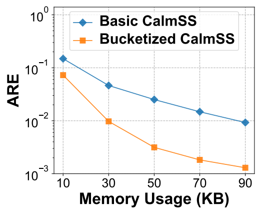
Methods	F1 Score/Recall	Throughput
HyperBF vs. Clock	+2.50%	+8.10 M/s
HyperBF vs. SWAMP	+60.65%	+11.99 M/s
HyperBF vs. TOBF	+81.85%	+5.13 M/s
HyperCalm vs. Clock+USS	+43.75%	+9.18 M/s

5.4 Experiments on Bucketized Partition Optimization

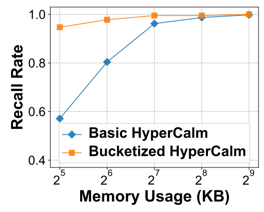
We evaluate the performance of Bucketized CalmSS on finding top-k items under the setting of § 5.2. We evaluate the performance of HyperCalm sketch using Bucketized TimeRecorder, LRU-Queue, and Space-Saving on finding periodic batches under the setting of § 5.3. By default, we allocate 3KB memory to HyperBF, and allocate the rest memory to Bucketized TimeRecorder, LRU-Queue, and



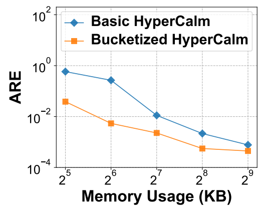
(a) Recall of CalmSS.



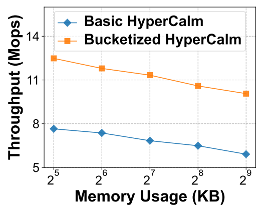
(b) ARE of CalmSS.



(c) Recall of HyperCalm.



(d) ARE of HyperCalm.



(e) Speed of HyperCalm.

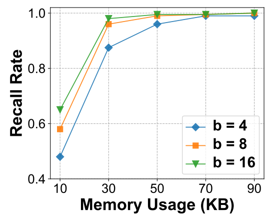
Fig. 19: Performance of Bucketized Partition Optimization.

Space-Saving in a ratio of 7:1:7. We set the number of slots per bucket in TimeRecorder, LRU-Queue, and Space-Saving to 32, 32, and 16, respectively, and we use 16-bit fingerprints in TimeRecorder and LRU-Queue.

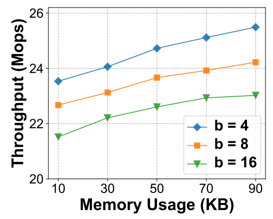
Accuracy on finding top-k items (Figure 19a-19b): *We find that our Bucketized Partition technique effectively improves the Recall Rate and ARE of CalmSS on finding top-k items.* When using 30KB memory, Bucketized Partition improves the Recall Rate of CalmSS from 83% to 98%, and improves the ARE from 0.046 to 0.010.

Accuracy on finding periodic batches (Figure 19c-19d): *We find that our Bucketized Partition technique effectively improves the Recall Rate and ARE of HyperCalm on finding periodic batches.* When using 32KB memory, Bucketized Partition improves the Recall Rate of HyperCalm from 57% to 95%, and improves the ARE from 0.583 to 0.039. As discussed in § 3.5, Bucketized HyperCalm achieves memory efficient by removing the hash table index, and thus achieves higher accuracy under the same memory usage.

Processing speed (Figure 19e): *We find that Bucketized HyperCalm sketch has faster speed than the basic version.* When using 32KB memory, Bucketized Partition improves the insertion speed from 7.65 Mops to 12.48 Mops. Bucketized HyperCalm is faster because it reduces the number of memory access, and avoid complicated pointer operations.



(a) Recall.



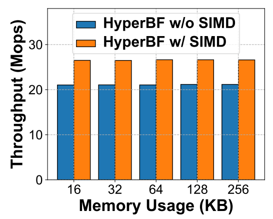
(b) Speed.

Fig. 20: Impact of number of slots per CalmSS bucket (b).

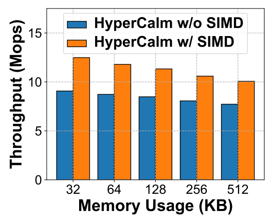
Impact of number of slots per bucket in CalmSS (b) (Figure 20): *We find that larger b goes with higher accuracy and slower speed.* This is because using large buckets allows us to better approximate the theoretical results of basic Space-Saving, but this requires us to probe more slots simultaneously. In practice, when using 32-bit ID, we recommend to set $b = 16$, so that the access of each bucket can be accelerated with AVX-512 SIMD instructions (§ 5.5).

5.5 Experiments on SIMD Optimization

We evaluate the speed improvement of the SIMD optimization described in § 3.6. For HyperBF, we set $d = 8$ and $l = 256$. Under such setting, each block of HyperBF is of 512-bit, which can be accelerated using AVX-512 instructions. We use Bucketized TimeRecorder, LRU-Queue, and CalmSS, where we use 16-bit fingerprints and set their numbers of slots per bucket to 32, 32, and 16, respectively. Under such setting, the IDs/fingerprints in each bucket occupy 512 bits, which again can be accelerated with AVX-512 instructions.



(a) HyperBF.



(b) HyperCalm.

Fig. 21: Impact of SIMD acceleration.

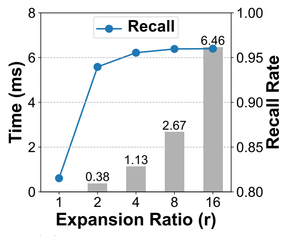
Speed of HyperBF (Figure 21a): *We find that SIMD instructions improve the speed of HyperBF by 25.7%∼26.1%.* When using 16KB memory, the insertion throughput of HyperBF without and with SIMD acceleration are 21.04 Mops and 26.50 Mops, respectively. We can see that SIMD instructions effectively optimize the speed of HyperBf by using 512-bit vectorization operations on each block.

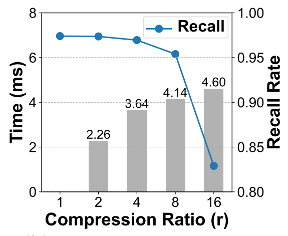
Speed of HyperCalm (Figure 21b): *We find that SIMD instructions improve the speed of HyperCalm by 30.3%∼37.6%.* When using 32KB memory, the insertion throughput of HyperCalm sketch without and with SIMD acceleration are 9.07 Mops and 12.48 Mops. We can see that SIMD instructions effectively optimize the speed of HyperCalm by accelerating HyperBF and accelerating the access to the buckets of TimeRecorder/LRU-Queue/Space-Saving.

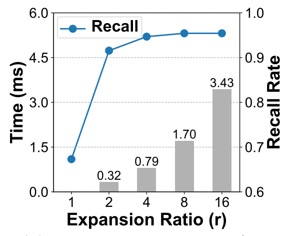
5.6 Experiments on Dynamic Memory Adjustment

We conduct experiments to evaluate the performance of HyperBF/HyperCalm under dynamic memory adjustment operations. We conducted experiments using CAIDA dataset. Initially, we build HyperBF/HyperCalm and insert the first 10% of the dataset into them. Subsequently, we execute the expansion/compression operations on HyperBF/HyperCalm to expand/compress them by different ratios (r). Afterwards, we proceed to insert the remaining 90% of the dataset. We report the time taken by the expansion and compression operations, along with the final Recall Rate for detecting batches and periodic batches.

Performance of HyperBF on dynamic memory adjustment (Figure 22a-22b): We find that HyperBF can flexibly manage the trade-off between its accuracy and memory usage via the memory adjustment operations. Moreover, the memory adjustment operations of HyperBF can be efficiently completed within milliseconds. In Figure 22a, we build an initial HyperBF of 16KB, and expand it by four different ratios ($r = 2, 4, 8, 16$). The results show that by expanding its size by $r = 2$ times, HyperBF improves its Recall Rate from 81.5% to 94.0%, and it only takes 0.38ms to complete the expansion operation. In Figure 22b, we build an initial HyperBF of 256KB, and compress it by four different ratios


(a) Exp. on HyperBF.


(b) Comp. on HyperBF.


(c) Exp. on HyperCalm.

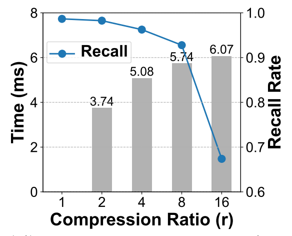
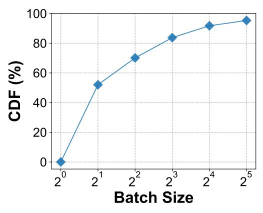
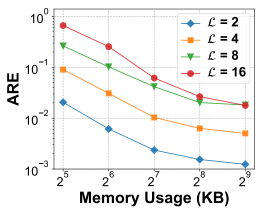
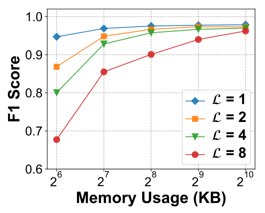
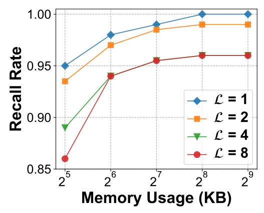

(d) Comp. on HyperCalm.

Fig. 22: Performance of dynamic memory adjustment.


(a) Batch size distr.


(b) ARE of batch size.


(c) F1 of large batch.


(d) Recall of PLB.

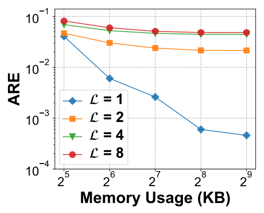

(e) ARE of PLB.

Fig. 23: Performance of estimating batch sizes and mining *Periodic Large Batches (PLB)*.

($r = 2, 4, 8, 16$). The results show that after compressing HyperBF by $r = 2$ times, its Recall Rate only drops from 97.4% to 97.3%, and it only takes 2.26ms to complete the compression operation.

Performance of HyperCalm on dynamic memory adjustment (Figure 22c-22d): We find that HyperCalm sketch can flexibly manage the trade-off between its accuracy and memory usage via the memory adjustment operations, and it also achieves millisecond-level memory adjustment. In Figure 22c, we build an initial HyperCalm of 16KB, and expand it by four different ratios ($r = 2, 4, 8, 16$). Specifically, we perform the expansion operation on all of its components (HyperBF, Bucketized TimeRecorder, Bucketized LRU-Queue, and Bucketized Space-Saving). The results show that by expanding its size by $r = 2$ time, HyperCalm improves its Recall Rate from 67.3% to 91.6%, and it only takes 0.32ms to complete the expansion operation. In Figure 22d, we build an initial HyperCalm of 256KB, and compress it by four different ratios ($r = 2, 4, 8, 16$). The results show that after compressing HyperCalm by $r = 2$ times, its Recall Rate only drops from 98.7% to 98.2%, and it only takes 3.74ms to complete the compression operation.

5.7 Experiments on Mining Periodic Large Batches

We evaluate the performance of our extended HyperCalm sketch in estimating batch sizes and mining periodic large batches. The experiments are conducted using CAIDA dataset under the setting in § 5.3. By default, we use $\lceil log(\mathcal{L})\rceil$-bit counters in our CM sketch.

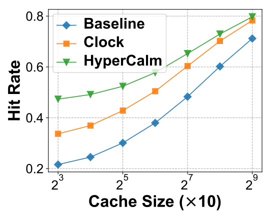
Batch size distribution (Figure 23a): We first study the batch size distribution in CAIDA dataset. We can see that the batch size distribution is highly skewed, where most batches are of small sizes and only a few batches have large sizes. For example, there are 70.1%/91.8% batches whose sizes are smaller than 4/16. Therefore, after efficiently filtering small batches, we can significantly reduce the memory overhead of TimeRecorder and CalmSS.

Accuracy on estimating batch sizes (Figure 23b): *We find that the extended HyperBF achieves high accuracy on estimating batch sizes for small batches.* We evaluate the ARE of the estimated size for the batches whose real sizes are not larger than $\mathcal{L}$. We can see that when using 512KB total memory, the ARE for batches whose sizes are smaller than 16 is < 0.018.

Accuracy on finding large batches (Figure 23c): *We find that the extended HyperBF achieves high accuracy on finding large batches.* When using 1024KB memory, extended HyperBF achieves > 0.963 F1 score on reporting batches whose sizes are larger than 8. Thus, extended HyperBF can accurately detect periodic large batches.

Accuracy on finding periodic large batches (Figure 23d): *We find that the extended HyperCalm achieves high accuracy on finding periodic large batches.* When using 128KB total memory, our HyperCalm achieves $> 95\%$ Recall Rate in finding periodic batches with sizes larger than 8.

Accuracy on periodicity estimation for periodic large batches (Figure 23e): *We find that the extended HyperCalm achieves high accuracy on estimating the periodicity of periodic large batches.* When using 512KB total memory, the ARE for periodic batches with sizes larger than 8 is 0.048.


(a) LFU.

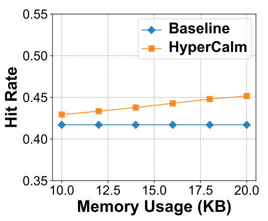
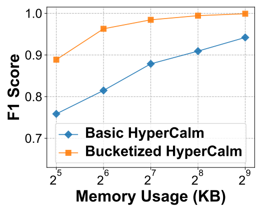

(b) LRU.

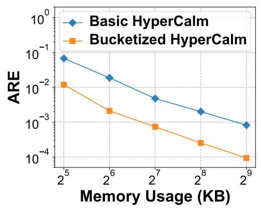
Fig. 24: Optimization to cache replacement policy.

5.8 Applying HyperCalm to Cache Systems

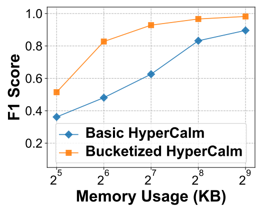
We apply HyperCalm to a simulated cache system. HyperCalm yields two insights to optimize cache performance. First, with the help of real-time batch detection, we can find out the batches that are still active now. When cache is full, we do not discard those items that still have active batches because they are highly likely to arrive again in the near future. Second, with the historical knowledge of periodic batches, we can forecast the arrival time of new batches, so as to prefetch data into cache before their arrival. We implement a fully associative cache simulator that mimics



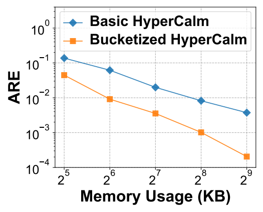
(a) F1 on packet drops.



(b) Batch size ARE.



(c) F1 on inflated delay.



(d) Batch timespan ARE.

Fig. 25: Performance of HyperCalm on network measurement.

the behavior of a hardware cache. We use CAIDA [56] and treat source IP address (4 bytes) as memory access request. We use the measurement results of HyperCalm to improve the cache hit rate as described above.

Experiments on LFU (Figure 24a): *We find that HyperCalm significantly improves the hit rate of LFU with small memory overhead, and HyperCalm outperforms Clock-Sketch [2] in both hit rate and processing speed.* We set the memory of HyperCalm and Clock to 20KB. The results show that HyperCalm always has higher hit rate than Clock and the LFU baseline. With the cache size of 1280, the hit rate of HyperCalm is 67%, while that of Clock and LFU are 60% and 48%.

Experiments on LRU (Figure 24b): *We find that HyperCalm improves the hit rate of LRU, and the hit rate grows higher as the memory of HyperCalm grows larger.* We set the cache size to 640 lines (about 80KB) and change the memory of HyperCalm. The results show that HyperCalm always has higher hit rate than the LRU baseline. When using 20KB of memory, HyperClam improves the hit rate from 41% to 45%.

5.9 Applying HyperCalm to Network Measurement

We apply our HyperCalm sketch to the scenario of network traffic measurement. In this scenario, each item in the data stream is a packet in the network traffic, whose ID is defined as the 5-tuple of a flow. We deploy one HyperCalm sketch on each hop (*e.g.*, switch) inside the network to report periodic large batches and some batch-level information (*e.g.*, batch size, sequence number range, *etc.*). A central analyzer periodically collects these information from all switches, and further analyze these information to identify abnormal events. We focus on two abnormal events: packet drops and inflated queuing delays. The experiments are conducted using CAIDA [56] dataset, where we simulate two switches connected by a link. To create abnormal events, we configure the link to proactively drop packets or increase the packet intervals for some batches. Then we analyze the batch-level information of periodic large batches reported by upstream/downstream switches to find abnormal events.

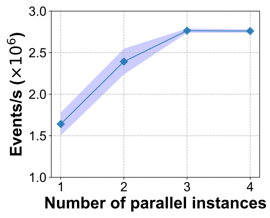
Finding packet drops (Figure 25a-25b): *We find that HyperCalm achieves $>95\%$ F1 score in finding packet drops.* In this experiment, we configure HyperCalm to report the size and the TCP Sequence Number range for each batch, which is implemented by adding three fields to each slot in LRU-Queue/Space-Saving. The analyzer first use the Sequence Number range to trace each batch, and it reports a batch experiencing packet drops if its size suddenly decreases. We can see that even with 32KB memory, HyperCalm still achieves nearly 90% F1 score on finding packet drops. We also evaluate the ARE of HyperCalm in estimating batch sizes in Figure 25b. We can see that when using 512KB memory, HyperCalm achieve $< 10^{-4}$ ARE, showing that HyperCalm has high accuracy in estimating batch sizes.

Finding inflated queuing delays (Figure 25c-25d): *We find that HyperCalm achieves $>95\%$ F1 score in finding inflated queuing delays.* In this experiment, we configure HyperCalm to report the timespan and the TCP Sequence Number range for each batch, which is implemented by adding four fields to each slot in LRU-Queue/Space-Saving. The analyzer first use the Sequence Number range to trace each batch, and it reports a batch experiencing inflated queuing delay if its timspan suddenly increases. We can see that when using 128KB memory, HyperCalm achieves $> 90\%$ F1 score in finding inflated delays. We also evaluate the ARE of HyperCalm in estimating batch timespans in Figure 25c. We can see that when using 512KB memory, HyperCalm achieves $< 2 \times 10^{-4}$ ARE, showing that HyperCalm has high accuracy in estimating batch timespans.

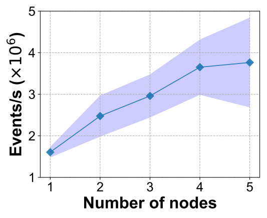
5.10 Integration into Apache Flink

Experimental setup: We run the experiments at a Flink cluster with 1 master and 5 workers using CAIDA [56]. We deploy a Hadoop Distributed File System (HDFS) with one NameNode (master) and 5 DataNodes (workers) in our cluster. Each node has 4 virtual CPU cores of Intel XEON Platinum 8369B, and 8 GB main memory. The job manager and each task manager of Flink are configured with 1 GB of memory. Each node uses Flink 1.13.1, Java 11 and Hadoop 2.8.3 running on Ubuntu 20.04 LTS. All experiments are repeated 10 times and average ($\pm$std) throughput is plotted.

Experimental results (Figure 26): We find that HyperCalm can smoothly work on top of Flink framework. As shown in Figure 26a, in local mode experiments, the throughput linearly increases up to 3 parallel instances (parallelism). Afterwards, the throughput growth becomes less linear. As shown in Figure 26b, in cluster mode, the throughput linearly scales up with more nodes used in the cluster.



(a) Local mode.



(b) Cluster mode.

Fig. 26: Throughput on Apache Flink.

5.11 Integration into Redis Database

We implement HyperCalm in Redis database, a popular in-memory data structure store widely used by database, cache, and streaming engine, showing that HyperCalm can be easily integrated into mainstream KV databases.

Experimental setup: We implement our HyperBF/CalmSS/HyperCalm using Redis module, where we provide API for users to create sketches, insert items, and query top-k periodic batches. The experiments are conducted on a machine with dual 18-core CPUs (36 threads, Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz) and 125GB DRAM memory.

Experimental results (Figure 27): We find HyperBF/CalmSS/HyperCalm achieve 0.234/0.227/0.115 Mops insertion throughput in Redis, which are not affected by memory usage. This is because the speed bottleneck of Redis lies in the communication with Redis server rather than sketch insertion, and thus the memory has little effect on throughput.

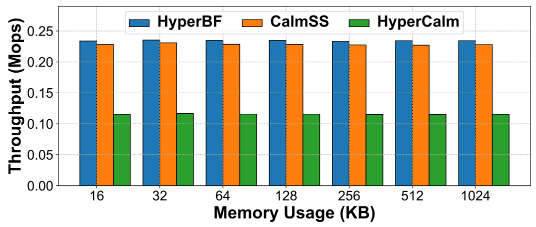


Fig. 27: Throughput in Redis.

6 Conclusion

This paper proposes a new pattern in data streams, namely periodic batches, which is useful in many applications. We propose the HyperCalm sketch, to accurately detect batches and periodic batches in real time. The two key components of HyperCalm, HyperBF and CalmSS, significantly outperform state-of-the-art solutions in detecting batches and finding top-k items, respectively. We provide theoretical guarantees for HyperBF and CalmSS. Extensive experimental results demonstrate the effectiveness of our approach. All related codes are available at GitHub [22].

Acknowledgment

We thank the anonymous reviewers for their valuable suggestions. This work is supported by Key-Area Research and Development Program of Guangdong Province 2020B0101390001, and National Natural Science Foundation of China (NSFC) (No. U20A20179, 61832001).

References

[1] Z. Liu, C. Kong, K. Yang, T. Yang, R. Miao, Q. Chen, Y. Zhao, Y. Tu, and B. Cui, "Hypercalm sketch: One-pass mining periodic batches in data streams," in *2023 IEEE 39th International Conference on Data Engineering (ICDE). IEEE*, 2023.

[2] P. Chen, D. Chen, L. Zheng, J. Li, and T. Yang, "Out of many we are one: Measuring item batch with clock-sketch." SIGMOD, 2021.

[3] J. Perry, H. Balakrishnan, and D. Shah, "Flowtune: Flowlet control for datacenter networks," in *NSDI*, 2017.

[4] T. Lévai, F. Németh, B. Raghavan, and G. Rétvári, "Batchy: Batch-scheduling data flow graphs with service-level objectives," in *NSDI*, 2020.

[5] R. Lei, P. Wang, R. Li, P. Jia, J. Zhao, X. Guan, and C. Deng, "Fast rotation kernel density estimation over data streams," in *SIGKDD*, 2021.

[6] C. Pirrong, "Energy market manipulation: definition, diagnosis, and deterrence," *Energy LJ*, vol. 31, p. 1, 2010.

[7] T. Chen, H. Yin, H. Chen, H. Wang, X. Zhou, and X. Li, "Online sales prediction via trend alignment-based multitask recurrent neural networks," *KAIS*, 2019.

[8] Z.-L. Zhang, V. J. Ribeiro, S. Moon, and C. Diot, "Small-time scaling behaviors of internet backbone traffic: An empirical study," in *INFOCOM*, 2003.

[9] F. J. Corbato, "A paging experiment with the multics system," MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, Tech. Rep., 1968.

[10] 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.

[11] Y. Peng, J. Guo, F. Li, W. Qian, and A. Zhou, "Persistent bloom filter: Membership testing for the entire history," in *SIGMOD*, 2018.

[12] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *ICDT*. Springer, 2005.

[13] D. Ting, "Data sketches for disaggregated subset sum and frequent item estimation," in *SIGMOD*, 2018.

[14] E. D. Demaine, A. López-Ortiz, and J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *ESA*. Springer, 2002.

[15] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *SIGMOD*, 2016.

[16] G. Cormode, "Sketch techniques for approximate query processing," *Foundations and Trends in Databases. NOW publishers*, 2011.

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

[18] P. Jia, P. Wang, J. Zhao, Y. Yuan, J. Tao, and X. Guan, "Loglog filter: Filtering cold items within a large range over high speed data streams," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 804–815.

[19] L. Zhang and Y. Guan, "Frequency estimation over sliding windows," in *ICDE*. IEEE, 2008, pp. 1385–1387.

[20] R. B. Basat, R. Friedman, and R. Shahout, "Stream frequency over interval queries," *Proceedings of the VLDB Endowment*, vol. 12, no. 4, pp. 433–445, 2018.

[21] S. Sun, J. Zheng, and D. Li, "Hee-sketch: an efficient sketch for sliding-window frequency estimation over skewed data streams," in *ISPA*, 2019.

[22] "Hypercalm sketch source codes," https://github.com/HyperCalmSketch/HyperCalmSketch.

[23] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[24] "The redis in-memory data store," https://redis.io.

[25] S. Kong, T. He, X. Shao, C. An, and X. Li, "Time-out bloom filter: A new sampling method for recording more flows," in *ICOIN*, 2006.

[26] E. Assaf, R. B. Basat, G. Einziger, and R. Friedman, "Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free," in *INFOCOM*. IEEE, 2018.

[27] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, 2005.

[28] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *TOCS*, vol. 21, no. 3, 2003.

[29] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *Automata, Languages and Programming*, 2002.

[30] T. Akiba and Y. Yano, "Compact and scalable graph neighborhood sketching," in *SIGKDD*, 2016.

[31] Y. Yang, Y. Zhang, W. Zhang, and Z. Huang, "Gb-kmv: An augmented kmv sketch for approximate containment similarity search," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 458–469.

[32] P. Jia, P. Wang, J. Zhao, S. Zhang, Y. Qi, M. Hu, C. Deng, and X. Guan, "Bidirectionally densifying lsh sketches with empty bins," in *SIGMOD*, 2021.

[33] D. Ting, J. Malkin, and L. Rhodes, "Data sketching for real time analytics: Theory and practice," in *SIGKDD*, 2020.

[34] P. Wang, Y. Qi, Y. Zhang, Q. Zhai, C. Wang, J. C. Lui, and X. Guan, "A memory-efficient sketch method for estimating high similarities in streaming sets," in *SIGKDD*, 2019.

[35] D. Ting, "Count-min: Optimal estimation and tight error bounds using empirical error distributions," in *SIGKDD*, 2018.

[36] A. Santos, A. Bessa, C. Musco, and J. Freire, "A sketch-based index for correlated dataset search," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022.

[37] R. B. Basat, G. Einziger, M. Mitzenmacher, and S. Vargaftik, "Salsa: self-adjusting lean streaming analytics," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 864–875.
[38] A. Santos, A. Bessa, C. Musco, and J. Freire, "A sketch-based index for correlated dataset search," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022.
[39] B. Zhao, X. Li, B. Tian, and etal, "Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing," in *SIGKDD*, 2021.
[40] Z. Fan, R. Wang, Y. Cai, R. Zhang, T. Yang, Y. Wu, B. Cui, and S. Uhlig, "Onesketch: A generic and accurate sketch for data streams," *IEEE Transactions on Knowledge and Data Engineering*, 2023.
[41] D. Yang, B. Li, L. Rettig, and P. Cudré-Mauroux, "D˘22 histosketch: Discriminative and dynamic similarity-preserving sketching of streaming histograms," *IEEE Transactions on Knowledge and Data Engineering*, vol. 31, no. 10, pp. 1898–1911, 2018.
[42] D. Yang, B. Qu, J. Yang, L. Wang, and P. Cudre-Mauroux, "Streaming graph embeddings via incremental neighborhood sketching," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 5, pp. 5296–5310, 2022.
[43] W. Xie, F. Zhu, J. Jiang, E.-P. Lim, and K. Wang, "Topicsketch: Real-time bursty topic detection from twitter," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 8, pp. 2216–2229, 2016.
[44] Y. Wu, Z. Liu, X. Yu, J. Gui, H. Gan, Y. Han, T. Li, O. Rottenstreich, and T. Yang, "Mapembed: Perfect hashing with high load factor and fast update," in *SIGKDD*, 2021.
[45] K. Amphawan, P. Lenca, and A. Surarerks, "Efficient mining top-k regular-frequent itemset using compressed tidsets," in *New Frontiers in Applied Data Mining: PAKDD 2011 International Workshops, Shenzhen, China, May 24-27, 2011, Revised Selected Papers 15*. Springer, 2012, pp. 124–135.
[46] Q. Wen, K. He, L. Sun, Y. Zhang, M. Ke, and H. Xu, "Robustperiod: Robust time-frequency mining for multiple periodicity detection," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2328–2337.
[47] M. G. Elfeky, W. G. Aref, and A. K. Elmagarmid, "Stagger: Periodicity mining of data streams using expanding sliding windows," in *Sixth International Conference on Data Mining (ICDM'06)*. IEEE, 2006, pp. 188–199.
[48] Q. Yuan, J. Shang, X. Cao, C. Zhang, X. Geng, and J. Han, "Detecting multiple periods and periodic patterns in event time sequences," in *CIKM*, 2017.
[49] M. Toller, T. Santos, and R. Kern, "Sazed: parameter-free domain-agnostic season length estimation in time series data," *Data Mining and Knowledge Discovery*, vol. 33, no. 6, pp. 1775–1798, 2019.
[50] Z. Fan, Y. Zhang, T. Yang, M. Yan, G. Wen, Y. Wu, H. Li, and B. Cui, "Periodicsketch: Finding periodic items in data streams," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022.
[51] Z. Li, B. Ding, J. Han, R. Kays, and P. Nye, "Mining periodic behaviors for moving objects," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 1099–1108.
[52] M. Flynn, "Some computer organizations and their effectiveness. ieee trans comput c-21:948," *Computers, IEEE Transactions on Computers*, vol. C-21, pp. 948 – 960, 10 1972.
[53] "Supplementary materials of hypercalm sketch," https://github.com/HyperCalmSketch/HyperCalmSketch/blob/main/HyperCalm_Supplementary.pdf.
[54] Z. Liu, Y. Zhao, Z. Fan, T. Yang, X. Li, R. Zhang, K. Yang, Z. Zhong, Y. Huang, C. Liu, J. Hu, G. Xie, and B. Cui, "Burstbalancer: Do less, better balance for large-scale data center traffic," in *Proceedings of the IEEE 30th International Conference on Network Protocols (ICNP)*. IEEE, Nov. 2022, pp. 1–13.
[55] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *SIGCOMM*, 2018, pp. 561–575.
[56] "CAIDA dataset," Available: http://www.caida.org/home.
[57] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 196–202.

Authors

Zirui Liu received the B.S. degree in computer science from Peking University in 2021. He is currently a third-year Ph.D. student in the School of Computer Science of Peking University, advised by Prof. Bin Cui and Prof. Tong Yang. His research interest is probablistic data structures, and their application in network measurement and streaming data mining. He published papers in SIGMOD, SIGCOMM, NSDI, SIGKDD, ICDE, ICNP, *etc.*

Xiangyuan Wang is currently an undergraduate student of Peking University majoring in Information and Computing Sciences. His research interests include data structures and algorithms in network measurement.

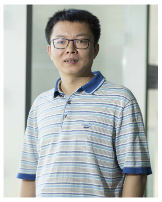
Yuhan Wu received his bachelor degree in the Department of Electrical Engineering and Computer Science at Peking University in 2021. Currently he is a CS PhD student in School of Computer Science at Peking University, advised by Tong Yang. His research interests lie in the fields of computer network and database, including key-value stores, network measurement, and sketches.

Tong Yang (Member, IEEE) received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently an Associate Professor with the School of Computer Science, Peking University. His research interests include network measurements, probabilistic algorithms, and KV stores. He has published more than 20 papers in SIGCOMM, SIGMOD, and SIGKDD.

Kaicheng Yang received his B.S. degree in Computer Science from Peking University in 2021. He is currently pursuing the Ph.D. degree in the School of Computer Science, Peking University, advised by Tong Yang. His research interests include network measurement and programmable data plane. He published papers in SIGCOMM, SIGMOD, NSDI, ICDE, WWW, ICNP, *etc.*

Hailin Zhang received the B.S. degree in computer science from Peking University in 2020. He is currently pursuing the Ph.D. degree with the School of Computer Science, Peking University, advised by Prof. Bin Cui. His Research interests include machine learning systems and distributed systems. He published papers in SIGMOD, VLDB, SCIS, etc.

Yaofeng Tu received the Ph.D. degree from Nanjing University of Aeronautics and Astronautics. China, in 2019. He is currently a research professor in ZTE company. His main research interests include distributed computing, big data system, and machine learning.



Bin Cui (Senior Member, IEEE) is a professor and Vice Dean in School of CS at Peking University. He obtained his Ph.D. from National University of Singapore in 2004. His research interests include database system architectures, query and index techniques, big data management and mining. He is serving as Vice Chair of Techical Commettee on Database China Computer Federation (CCF) and Trustee Board Member of VLDB Endowment, is also in the Editorial Board of Distributed and Parallel Databases, Journal of Computer Science and Technology, and SCIENCE CHINA Information Sciences, and was an assocaite editor of IEEE Transactions on Knowledge and Data Engineering (TKDE) and VLDB Journal.