HyperCalm Sketch: One-Pass Mining Periodic Batches in Data Streams

Zirui Liu[†], Chaozhe Kong[†], Kaicheng Yang[†], Tong Yang[†‡], Ruijie Miao[†],
Qizhi Chen[†], Yikai Zhao[†], Yaofeng Tu[§], and Bin Cui[†]
[†] School of Computer Science, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, Beijing, China [‡] Peng Cheng Laboratory, Shenzhen, China [§] ZTE Corporation
{zirui.liu, kcz, ykc, miaoruijie, hzyoi, zyk, bin.cui}@pku.edu.cn, {yangtongemail}@gmail.com, {tu.yaofeng}@zte.com.cn

***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*. We propose a one-pass sketching algorithm, namely the HyperCalm sketch, which takes two phases to detect periodic batches in real time. In phase 1, we propose a time-aware Bloom filter, namely HyperBloomFilter (*HyperBF*), to detect the start of batches. In phase 2, we propose an enhanced top-k algorithm, called Calm Space-Saving (*CalmSS*), to report top-k periodic batches. We theoretically derive the error bounds for HyperBF and CalmSS. Extensive experiments show HyperCalm outperforms the strawman solutions 4× in term of average relative error and 13.2× in term of speed. We also apply HyperCalm to a cache system and integrate HyperCalm into Apache Flink. All related codes are open-sourced [1].**

***Index Terms*—Data Streams, Sketch, Periodic Batch**

I. Introduction

A. Background and Motivation

Batch is an important pattern in data streams [1], 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 [1], networks [2], and machine learning [3], [4], 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 [2].

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 [5]. 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 [6] (*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 [7]. 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 the best of 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 [1], which records the last arrival time of recent items in a cyclic array, and uses another thread to clean the outdated information using CLOCK [8] 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.

B. Our Proposed Solution

To accurately detect periodic batches in real time, we propose a one-pass sketching algorithm, namely HyperCalm. HyperCalm takes two phases to find top-k periodic batches. In phase 1, for each item e arriving at time t, we check whether

[1] https://github.com/HyperCalmSketch/HyperCalmSketch

[2] We give the formal definition and an example of *periodicity* in § II-A.

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 [1]. 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 to existence detection, phase 1 should be aware of arrival time to divide a series of the same item into many batches. Bloom filter [9] 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) [10]. 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 [11], Unbiased Space-Saving [12], and Frequent [13]. However, their accuracy is significantly harmed by cold items [3]. 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 [16] and LogLogFilter [17] 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, and discard those cold items in the sliding window. Rather than using existing sliding window algorithms [18], [19], [20], 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 § III-C). Third, our LRU queue can be naturally integrated into the data structure of Space-Saving (see details in § III-D): 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 11c). 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 (§ V-C). All related codes are open-sourced [21].

[3]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 [14], [15]

C. Key Contributions

- We formulate the problem of finding periodic batches in data streams. We believe this is an important problem in data mining.
- 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 state-of-the-art solutions 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 showing that HyperCalm well achieves our design goal. The results show HyperCalm outperforms the strawman solutions $4\times$ in term of average relative error and $13.2\times$ in term of processing speed.
- We apply HyperCalm to a cache system showing that periodic batches can benefit real-world application. We integrate HyperCalm into Apache Flink [22] showing that HyperCalm can smoothly work in distributed systems.

II. Background and Related Work

A. 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 present an example to further clarify our problem definition. We focus on two kinds of 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.

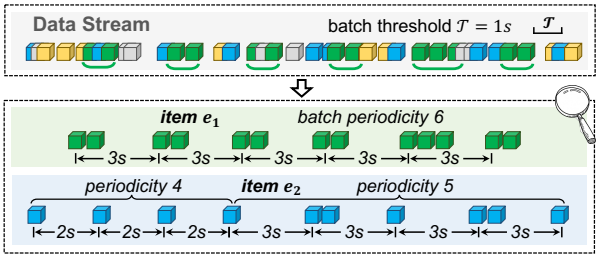


Fig. 1: Example of periodic batches.

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 § V-E 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. Further formulating more application-specific variants of periodic batches is our future work.

B. 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 [1], 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)* [23] and *SWAMP* [24].

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 [25], CU [26], Count [27], and more [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38]. 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 [11], Unbiased Space-Saving [12], Frequent [13], and more [39]. 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 [16] uses a two-layer CU sketch to filter cold items. However, as aforementioned, the structure of Cold Filter will be filled up very quickly. Cleaning the full Cold Filter will inevitably incur error and time overhead, which is still not addressed.

Mining periodic patterns: Although there have been some algorithms aiming at mining periodicity in time sequence data [40], [41], [42], [43], [44], [45], [46], 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 [43] defines a periodical problem in an incomplete sequence data, and develops an iterative algorithm with time complexity of $O(n^2)$. RobustPeriod [41] 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 [44]. These algorithms need to process one item multiple times, and thus cannot meet the above two requirements.

TABLE I: 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
$\mathcal{P}$	Predefined promotion threshold of the LRU queue

III. 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 $V = t - \hat{t}$. [4] Then we update the arrival time of last batch of e in the TimeRecorder queue to t. Next, we 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 use a hash table index to accelerate the lookup process of the TimeRecoder queue and CalmSS, achieving O(1) processing time complexity (see more details in § III-B and § III-C). The main symbols used in this paper are listed in Table I.

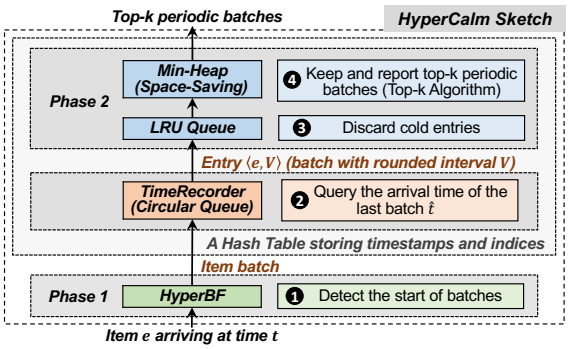


Fig. 2: HyperCalm sketch workflow.

A. 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 [1], HyperBF incidentally cleans the outdated cells during each insertion operation. Compared with 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 § IV.

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 § IV). 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.

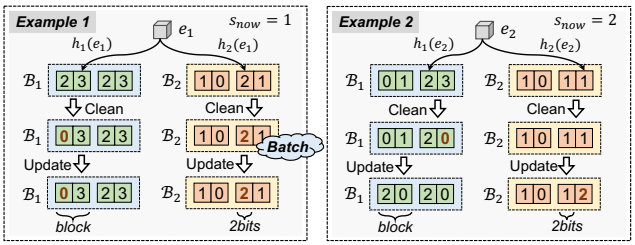


Fig. 3: Two examples of HyperBF ($d = 2$, $m = 4$, $l = 2$).

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}.

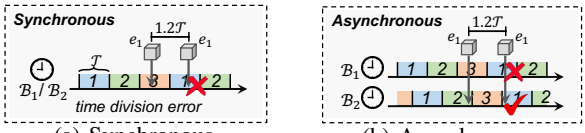
Example 2 (right of Figure 3): For item e_2 arriving at time slice $s_{now} = 2$, we first locate its two hashed cells $\mathcal{B}_1[3]$ and $\mathcal{B}_2[4]$. Next, we check the blocks which the two hashed cells reside, and clean the outdated cells with value 3, *i.e.*, clean $\mathcal{B}_1[4]$ to 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 true. The error of missing batches 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

[4] To tolerate noise in batch interval, in our experiments, V is rounded up according to the regulations described in the parameter setting part of § V-D.

timeline division. We provide the theoretical analysis of the three kinds of error in § IV, 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, we found the error incurred by fuzzy perception of time can be significantly reduced by the technique of *Asynchronous Timeline*.

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 IV.3 in § IV, proving that the time division error can be reduced by d times when using d evenly distributed timelines.



(a) Synchronous. (b) Asynchronous.

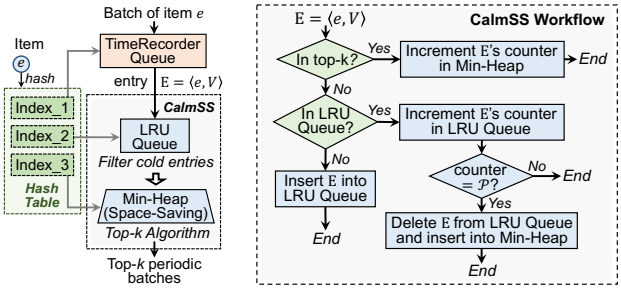
Fig. 4: Optimization using *Asynchronous Timeline*

B. 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. **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 § III-D.

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.



(a) Data structure. (b) Insertion workflow.

Fig. 5: Data structure and workflow of CalmSS.

C. 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 [11], which works by maintaining a Min-Heap of m bins. For each incoming entry E_1[5], 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, for each incoming entry, it 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 the Space-Saving. This *LRU Queue* can be considered as a guardian of Space-Saving to keep cold entries outside. We theoretically prove the error bound of CalmSS in § IV.

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

[5]Each entry $E = \langle e, V \rangle$ is the concatenation of an item ID e and a batch interval V.

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 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 [11] and a hash table to achieve O(1) time complexity to locate and update the smallest entry. The hash table is used to index both the Stream-Summary and the LRU queue. Note that CalmSS is a meta-framework that accommodates any top-k algorithm, which means the Space-Saving can be replaced by other top-k algorithm, such as Unbiased Space-Saving [12], Frequent [47], and more [39]. We use Space-Saving because it is most well known and has the best theoretical results. 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, respectively.

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.

Report: To report top-k periodic batches, CalmSS just reports the k entries with the k largest frequencies in the Min-Heap. Note that one item could have multiple groups of periodic batches, and thus could be reported more than once.

D. 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.

IV. Mathematical Analysis

In this section, we provide a thorough theoretical support for the HyperCalm sketch, and validate our theoretical analyses using experiments. Our theoretical analyses focus on the following three key issues.

- **How accurate can HyperBF detect batches?** We derive the error bound of HyperBF in Lemma IV.4 and Theorem IV.1, and conduct experiments to validate our bound in Figure 7b. The results show that both theoretical and experimental error rate 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 IV.2, and conduct experiments to validate our bound in Figure 8. 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 IV.3, and conduct experiments to validate it in Figure 9b. Both theoretical and experimental results show that *Asynchronous Timeline* technique significant improves the accuracy of HyperBF.

A. Error rate of HyperBF

We first prove the error rate of HyperBF in Theorem IV.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 [48] dataset as an instance, Figure 7a 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 $n\mathcal{T}$.

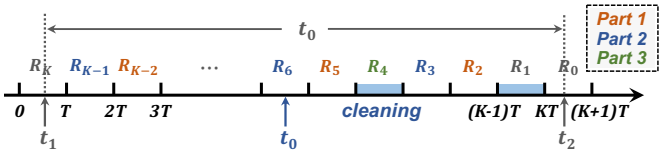


Fig. 6: Error rate analysis of HyperBF.

As shown in Figure 6, 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, which is defined as the probability that HyperBF does not report a batch at t_2.

Now consider a certain hashed cell $\mathcal{B}_i[h_i(e)]$ of item e. It is obvious that $\mathcal{B}_i[h_i(e)]$ is accessed by e at both t_1 and t_2.

Let t_0 be the last time that $\mathcal{B}_i[h_i(e)]$ was accessed before t_2. We have $t_1 \leqslant t_0 < t_2$. In particular, if $t_0 = t_1$, we can assert that there is no item hashed into $\mathcal{B}_i[h_i(e)]$ between t_1 and t_2. Assume that $0 \leqslant t_1 < \mathcal{T}$ and $K\mathcal{T} \leqslant t_2 < (K+1)\mathcal{T}$, *i.e.*, $K = \left\lfloor \frac{t_2}{\mathcal{T}} \right\rfloor - \left\lfloor \frac{t_1}{\mathcal{T}} \right\rfloor$. Since t_1 and t_2 are two arbitrary selected timestamps, and the timeline can be arbitrarily specified, the above assumption does not impair generality. Next, we use symbol R_i ($0 \leqslant i \leqslant K$) to denote the time interval $[(K-i)\mathcal{T}, (K+1-i)\mathcal{T})$. We restrict the range of possible t_0 into $[\mathcal{T}, (K+1)\mathcal{T})$. And we can assume that $\mathcal{B}_i[h_i(e)]$ can be cleaned by other items only in $[0, K\mathcal{T})$.

Lemma IV.1. *Let $\mathcal{A}'_j$ denote the event that $t_0 \leqslant (K+1-j)\mathcal{T}$. Let $j' = \frac{(j-K-1)\mathcal{T}+t_2}{\mathcal{T}}$. Then we have $\forall 1 \leqslant j < K$, $\Pr(\mathcal{A}'_j) \approx e^{-\frac{\gamma_{j'}}{m}}$.*

Proof. The probability that $\mathcal{B}_i[h_i(e)]$ is not selected by a certain hash function during the insertion of an item is $1 - \frac{1}{m}$. Note that $t_0 \leqslant (K+1-j)\mathcal{T}$ is equivalent to the statement that $\mathcal{B}_i[h_i(e)]$ is not selected by any item that arrives between $(K+1-j)\mathcal{T}$ and t_2. From our data stream assumption, the number of distinct items arrives between $(K+1-j)\mathcal{T}$ and t_2 is $\gamma_{j'} = \alpha + \beta(j-K-1)\mathcal{T} + \beta t_2$. Since the hash values of distinct items have no significant correlation between each other, then the probability that $\mathcal{B}_i[h_i(e)]$ is not selected by any of the $\gamma_{j'}$ distinct items is $\Pr(\mathcal{A}'_j) = \left(1 - \frac{1}{m}\right)^{\gamma_{j'}} \approx e^{-\frac{\gamma_{j'}}{m}}$ □

Lemma IV.2. *For $\forall 0 \leqslant j < K$, let $\mathcal{A}'_j$ denote the event that $t_0 \leqslant (K-j+1)\mathcal{T}$, and let $\mathcal{A}_j$ denote the event that $(K-j)\,\mathcal{T} \leqslant t_0 < (K-j+1)\,\mathcal{T}$, i.e., the event that $t_0 \in R_j$. Then we have that for $\forall 1 \leqslant j < K$, $\Pr(\mathcal{A}_j) > e^{-\frac{\gamma_j}{m}} - e^{-\frac{\gamma_{j+1}}{m}}$.*

Proof. It is obvious that for $\forall 0 \leqslant j < K$, we have

$$\Pr(\mathcal{A}'_j) = \Pr(\mathcal{A}'_{j+1}) + \Pr(\mathcal{A}_j)$$

According to Lemma IV.1, we have that for $\forall 1 \leqslant j < K$:

$$\begin{aligned}\Pr(\mathcal{A}_j) &= \Pr(\mathcal{A}'_j) - \Pr(\mathcal{A}'_{j+1}) \approx e^{-\frac{\gamma_{j'}}{m}} - e^{-\frac{\gamma_{j'+1}}{m}} \\ &= e^{-\frac{\alpha+\beta(j-K-1)\mathcal{T}+\beta t_2}{m}} \left(1 - e^{-\frac{\beta\mathcal{T}}{m}}\right) \\ &= e^{-\frac{\gamma_{j'}}{m}} \left(1 - e^{-\frac{\beta\mathcal{T}}{m}}\right)\end{aligned}$$

Since $t_2 < (K+1)\mathcal{T}$, we have that $\gamma_{j'} = \alpha + \beta(j-K-1)\mathcal{T} + \beta t_2 < \alpha + \beta j\mathcal{T} = \gamma_j$.

Thus, we have

$$\begin{aligned}\Pr(\mathcal{A}_j) &= e^{-\frac{\gamma_{j'}}{m}} \left(1 - e^{-\frac{\beta\mathcal{T}}{m}}\right) > e^{-\frac{\gamma_j}{m}} \left(1 - e^{-\frac{\beta\mathcal{T}}{m}}\right) \\ &= e^{-\frac{\alpha+\beta j\mathcal{T}}{m}} - e^{-\frac{\alpha+\beta(j+1)\mathcal{T}}{m}} = e^{-\frac{\gamma_j}{m}} - e^{-\frac{\gamma_{j+1}}{m}}\end{aligned}$$

□

Lemma IV.3. *Let $m' = \frac{m}{l-1}$. Given u time intervals of length $\mathcal{T}$, $T_1, \cdots, T_u$, let $\mathcal{C}_u$ denote the event that a certain cell, e.g., $\mathcal{B}_i[h_i(e)]$, is cleaned at least once in these time intervals. Then we have:*

$$1 - e^{-\frac{\gamma_u}{m'}} \leqslant \Pr(\mathcal{C}_u) \leqslant 1 - e^{-\frac{u\gamma_1}{m'}}$$

Proof. For each incoming item e_i, $\mathcal{B}_i[h_i(e)]$ is cleaned if and only if e_i is hashed into the same block but not the same cell with e. Therefore, the probability that $\mathcal{B}_i[h_i(e)]$ is not cleaned during the insertion of e_i is $1 - \frac{l-1}{m} = 1 - \frac{1}{m'}$.

Let x be the number of distinct items in the u intervals, $T_1, \cdots, T_u$. According to the data stream assumption, we have $\gamma_u \leqslant x \leqslant u\gamma_1$. Actually, when the u intervals are consecutive, we have $x = \gamma_u$, and when the u intervals are disjoint and separated far away enough from each other, we have $x = u\gamma_1$. Since the probability that $\mathcal{B}_i[h_i(e)]$ is not cleaned in $T_1, \cdots, T_u$ is $\left(1 - \frac{1}{m'}\right)^x \approx e^{-\frac{x}{m'}}$, we have $\Pr(\mathcal{C}_u) = 1 - e^{-\frac{x}{m'}}$. Thus, we have $1 - e^{-\frac{\gamma_u}{m'}} \leqslant \Pr(\mathcal{C}_u) \leqslant 1 - e^{-\frac{u\gamma_1}{m'}}$. □

Lemma IV.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 $K_1 = \left\lfloor \frac{K}{3} \right\rfloor - 1$, $K_2 = \left\lfloor \frac{K-1}{3} \right\rfloor - 1$, and $K_3 = \left\lfloor \frac{K-2}{3} \right\rfloor - 1$. 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 $P'_3 = \sum_{k=0}^{K_3} \left(e^{-\frac{\gamma_{3k+4}}{m}} - e^{-\frac{\gamma_{3k+5}}{m}}\right)\left(1 - e^{-\frac{\gamma_{u_{3k+4}}}{m'}}\right)$.*

Proof. Now we discuss the possible range of t_0. Since our goal is to derive the lower bound of P, we can just ignore the case where $t_0 \in R_K$. And we note that when $t_0 \in R_0$ or $t_0 \in R_1$, $\mathcal{B}_i[h_i(e)]$ cannot be *zero* at t_2. Therefore, we only discuss the cases where $\mathcal{T} \leqslant t_0 < (K-1)\mathcal{T}$, *i.e.*, the cases where $t_0 \in R_j$ ($2 \leqslant j \leqslant K-1$).

First, consider the cases where $t_0 \in R_{3k+2}$ ($0 \leqslant k \leqslant \left\lfloor \frac{K}{3} \right\rfloor - 1$). In these cases, $\mathcal{B}_i[h_i(e)]$ will be cleaned to *zero* when inserting item e at t_2. Let $K_1 = \left\lfloor \frac{K}{3} \right\rfloor - 1$. We can derive the first part of P as $P_1 = \sum_{k=0}^{K_1} \Pr(\mathcal{A}_{3k+2}) = \sum_{k=0}^{K_1} \left(e^{-\frac{\gamma_{3k+2}}{m}} - e^{-\frac{\gamma_{3k+3}}{m}}\right) = P'_1$.

Second, consider the cases where $t_0 \in R_{3k+3}$ ($0 \leqslant k \leqslant \left\lfloor \frac{K-1}{3} \right\rfloor - 1$). As shown in Figure 6, when $t_0 \in R_6$, in order to guarantee that $\mathcal{B}_i[h_i(e)] = 0$ at t_2, $\mathcal{B}_i[h_i(e)]$ must be cleaned at least once in time intervals R_4 and R_1. Generally, when $t_0 \in R_j$, let u_j denote the number of intervals in which $\mathcal{B}_i[h_i(e)]$ should be cleaned at least once. Then we have $u_j = \left\lceil \frac{j-2}{3} \right\rceil$. Let $K_2 = \left\lfloor \frac{K-1}{3} \right\rfloor - 1$. We can derive the second part of P as $P_2 = \sum_{k=0}^{K_2} \Pr(\mathcal{A}_{3k+3}) \Pr(\mathcal{C}_{u_{3k+3}}) \geqslant \sum_{k=0}^{K_2} \left(e^{-\frac{\gamma_{3k+3}}{m}} - e^{-\frac{\gamma_{3k+4}}{m}}\right)\left(1 - e^{-\frac{\gamma_{u_{3k+3}}}{m'}}\right) = P'_2$.

Third, consider the cases where $t_0 \in R_{3k+4}$ ($0 \leqslant k \leqslant \left\lfloor \frac{K-2}{3} \right\rfloor - 1$). These cases are similar to the cases in the second part, and the proof is also similar.

In summary, we have that $P = P_1 + P_2 + P_3 \geqslant P'_1 + P'_2 + P'_3$. □

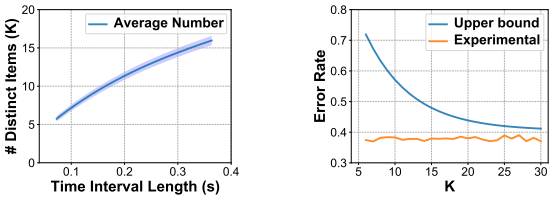
Theorem IV.1. *We define the error rate $\mathcal{E}$ of HyperBF (without Asynchronous Timeline) as the probability that HyperBF does not report a batch at t_2. Then we have:*

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

where P' is the lower bound in Lemma IV.4.

Proof. Note that HyperBF does not report a batch at t_2 if and only if all d hashed cells $\mathcal{B}_1[h_1(e)], \cdots, \mathcal{B}_d[h_d(e)]$ are *zero* at t_2. Since the d arrays of HyperBF are independent of each other, we have that $\mathcal{E} = (1-P)^d$, where P is the probability that one hashed cell is *zero* at t_2. Thus, we have $\mathcal{E} \leqslant (1-P')^d$, where P' is the lower bound of P derived in Lemma IV.4. This gives the proof of Theorem IV.1. □

Experimental analysis (Figure 7b): We conduct experiments on CAIDA [48] dataset to validate the theoretical bound in Lemma IV.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 our theoretical bound. Since 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. We can also see that 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. 7: Error rate of HyperBF.

B. 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.

Theorem IV.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*

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

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

Proof. Now suppose e fails to be discarded by the LRU queue, *i.e.*, it enters the top-k algorithm. Then e must arrives $\mathcal{P}$ times in a short time, and each arrival increments the counter of e in the LRU queue by one. Let random variables $T_1, T_2, \cdots, T_{\mathcal{P}-1}$ be the time gaps between every two adjacent occurrences of e. Since e arrives according to a Poisson process of intensity λ_c, we have T_i follows an exponential distribution with mean λ_c.

Let $\mathcal{D}_i$ denote the event that there arrive w hot items within T_i. It is clear that $\mathcal{D}_1, \cdots, \mathcal{D}_{\mathcal{P}-1}$ are independent of each other. Recall that the Poisson processes of different hot items are independent of each other. Let $\lambda_1 = n_h \lambda_h$ and $\lambda_2 = \lambda_c$. The probability of the event that there arrive x hot items within T is $P_{x,T} = \frac{(\lambda_1 T)^T}{x!} e^{-\lambda_1 T}$.

Then we have:

$$\Pr(\mathcal{D}_i) = \sum_{x=0}^{w-1} P_{x,T_i} = \sum_{x=0}^{w-1} \frac{(\lambda_1 T_i)^{T_i}}{x!} e^{-\lambda_1 T_i}$$

Recall that the Poisson processes of all distinct items are independent of each other. Then we have:

$$\begin{aligned} \zeta =& \mathbb{E}_{\{T_1, \cdots, T_{\mathcal{P}-1}\}} \left[\Pr \left(\prod_{i=1}^{\mathcal{P}-1} \mathcal{D}_i \right) \right] \\ =& \mathbb{E}_{\{T_1, \cdots, T_{\mathcal{P}-1}\}} \left[\prod_{i=1}^{\mathcal{P}-1} \Pr(\mathcal{D}_i) \right] = \prod_{i=1}^{\mathcal{P}-1} \mathbb{E}_{T_i} \left[\Pr(\mathcal{D}_i) \right] \end{aligned}$$

Further, we have:

$$\begin{aligned} \zeta &= \prod_{i=1}^{\mathcal{P}-1} \mathbb{E}_{T_i} \left[\Pr(\mathcal{D}_i) \right] = \prod_{i=1}^{\mathcal{P}-1} \sum_{x=0}^{w-1} \int_0^{+\infty} \frac{(\lambda_1 t_i)^x}{x!} e^{-\lambda_2 t_i} \mathrm{d}t_i \\ &= \left(\sum_{x=0}^{w-1} \frac{1}{x!} \frac{R^x}{(R+1)^{x+1}} \Gamma(x+1) \right)^{\mathcal{P}-1} \end{aligned}$$

where $R = \frac{\lambda_1}{\lambda_2}$ and $\Gamma(z)$ represents the Gamma function. □

Experimental analysis (Figure 8): We conduct experiments to validate our theoretical bound in Theorem IV.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.

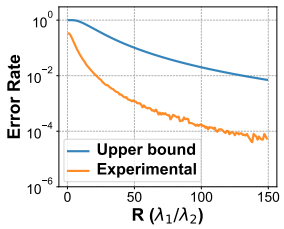


Fig. 8: Error of CalmSS.

C. Effectiveness of Asynchronous Timeline

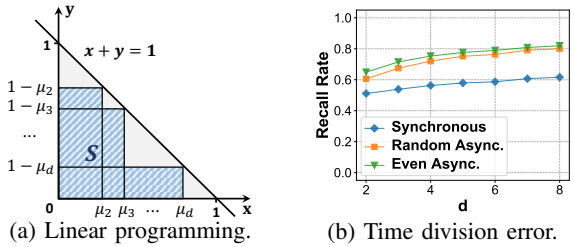
Theorem IV.3. *After using the Asynchronous Timeline technique, the time division error is minimized when the d timelines are evenly distributed, i.e., when the timeline offset for the i^{th} array is $o_i = \frac{(i-1)}{d}\mathcal{T}$, where the minimized error is reduced by d times compared to the synchronous version.*

Proof. The time division error occurs only when the batch interval spans two time slices in all of the d timelines. Without loss of generality, we take the first timeline as the reference timeline, *i.e.*, we set $o_1 = 0$, and we suppose $o_1 < o_2 < \cdots < o_d < \mathcal{T}$. Consider two batches arrives at timestamps $x\mathcal{T}$ and $(2-y)\mathcal{T}$ respectively, where $x > 0$, $y > 0$, and $x + y < 1$. The interval between the two batches is $\Delta t = (2 - y - x)\mathcal{T}$. It is clear that $\mathcal{T} < \Delta t < 2\mathcal{T}$. Consider the i^{th} timeline with offset $o_i \in [0, \mathcal{T})$, it can correctly perceive the second batch if and only if $x\mathcal{T} < o_i$ and $(2-y)\mathcal{T} > \mathcal{T} + o_i$. In other words, the i^{th} timeline can correctly perceive the second batch if the interval meets the above two constraints.

Our goal is to find the optimal $o_2, \cdots, o_d \in [0, \mathcal{T})$ to perceive as much intervals as possible. Suppose the valid values of x and y are uniformly distributed, the above problem can be transformed into a linear programming problem. As shown in Figure 9a, the triangular area under the line $x+y=1$ represents the feasible range of x and y. Let $o_i = \mu_i\mathcal{T}$ where $\mu_i \in [0, 1)$. Each timeline with offset o_i enables the intervals lie in the rectangular area $x > 0, y > 0, x < \mu_i$, and $y < 1-\mu_i$ to be correctly perceives. Therefore, the goal of the linear programming is to maximize the total area S, which is the union of the $d-1$ rectangles. And we have:

$$S = \mu_2(1-\mu_2) + (\mu_3-\mu_2)(1-\mu_3) + (\mu_4-\mu_3)(1-\mu_4) + \cdots + (\mu_d - \mu_{d-1})(1-\mu_d)$$

By applying the method of Lagrange multipliers, we can easily derive that S is maximized when $\mu_i = \frac{(i-1)}{d}$, *i.e.*, $o_i = \frac{(i-1)}{d}\mathcal{T}$, and the maximized S is $\frac{d-1}{d}$ times of the triangular area. Therefore, the time division error is reduced by d times compared to the synchronous version. □



(a) Linear programming.

(b) Time division error.

Fig. 9: Asynchronous Timeline analysis.

Experimental analysis (Figure 9b): We conduct experiments on CAIDA [48] to validate Theorem IV.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%.

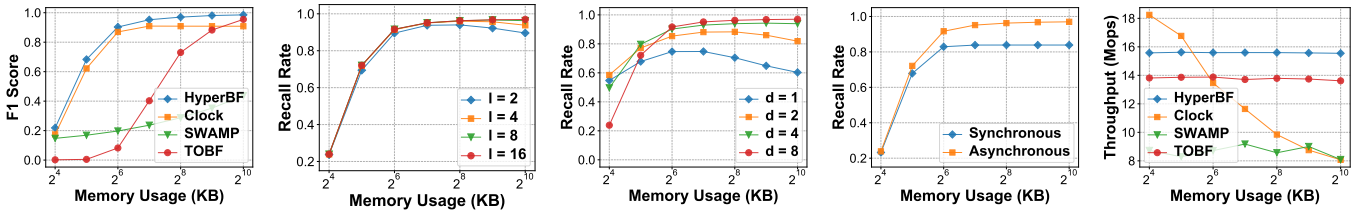
V. Experimental Results

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

- **Can HyperBF accurately and efficiently detects item batches?** We compare the performance of HyperBF with state-of-the-art Clock-Sketch [1], SWAMP [24], and Time-Out Bloom filter (TOBF) [23] in finding item batches. The results show that under the same memory usage, HyperBF always achieves higher accuracy and faster speed than state-of-the-art solutions. (§ V-B)
- **Can CalmSS accurately and efficiently detect top-k items?** We compare the performance of CalmSS with state-of-the-art Space-Saving (SS) [11], Unbiased Space-Saving (USS) [12], and Cold filter [16] + Space-Saving (CF+SS) in finding top-k items. The results show that under the same memory usage, HyperBF always achieves higher accuracy and faster speed than state-of-the-art solutions. (§ V-C)
- **Can HyperCalm accurately and efficiently detect top-k periodic batches?** We combine the state-of-the-art algorithms in detecting batches and finding top-k items to form one strawman solution for finding periodic batches, and compare HyperClam against it on two datasets. The results show that HyperCalm outperforms the strawman solutions $4\times$ in term of average relative error and $13.2\times$ in term of speed. (§ V-D)
- **Is it beneficial for real-world application to detect periodic batches?** We apply the HyperCalm sketch to a cache system, and use the measurement results of periodic batches to optimize the replacement and prefetch strategies. The results show that HyperCalm improves the hit rates of both LFU and LRU caches. (§ V-E)
- **Can HyperCalm work well in distributed systems?** We implement HyperCalm on top of Apache Flink [22], showing that our solution can be easily integrated into modern stream processing framework and work well in distributed environment. (§ V-F)

A. Experimental Setup

Platform and setting: We conduct experiments on an 18-core 4.2GHz CPU server (Intel i9-10980XE) with 128GB 3200MHz DDR4 memory and 24.75MB L3 cache. We implement all codes with C++ and build them with g++ 7.5.0 (Ubuntu 7.5.0-6ubuntu2) and -O3 option. The hash functions we use are 32-bit Murmur Hash [49]. We use SIMD (Single Instruction and Multiple Data) to accelerate the cleaning process of HyperBF. By default, the parameters of the comparing algorithms are set according to the recommendation of their authors. We first find the ground-truth batches / top-k items /



(a) F1 score. (b) Cell line size (l). (c) Number of Arrays (d). (d) Timeline distribution. (e) Processing speed.

Fig. 10: Performance of HyperBF (CAIDA).

periodic batches according to predefined parameters, and store them as golden labels in a large hash table.

Datasets:

1) CAIDA dataset: CAIDA Anonymized Internet Trace [48] is a data stream of anonymized IP trace collected in 2018. Each item is identified by its source IP (4 bytes) and destination IP (4 bytes).

2) Criteo dataset: Criteo dataset [50] is an online advertising click data stream consisting of about 45M ad impressions. Each item is identified by its categorical feature values and conversion feedback.

Evaluation Metrics:

1) Recall Rate (RR): The ratio of the number of correctly reported instances to the number of correct instances.

2) Precision Rate (PR): The ratio of the number of correctly reported instances to the number of reported instances.

3) F_1 **Score:** $\frac{2\times RR\times PR}{RR+PR}$.

4) Average Relative Error (ARE): $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}\left|f_i-\hat{f}_i\right|/f_i$, where f_i is the real frequency of item e_i, $\widehat{f_i}$ is its estimated frequency, and Ψ is the query set.

5) Throughput (Mops): Million operations per second.

B. Experiments on HyperBF

Parameter setting: We compare HyperBF with Clock-Sketch [1], SWAMP [24], and Time-Out Bloom filter (TOBF) [23]. For HyperBF, 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 10a): *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 10b): *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 10c): *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 10d): *We find that the **Asynchronous Timeline** technique can significantly improve the RR 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%).

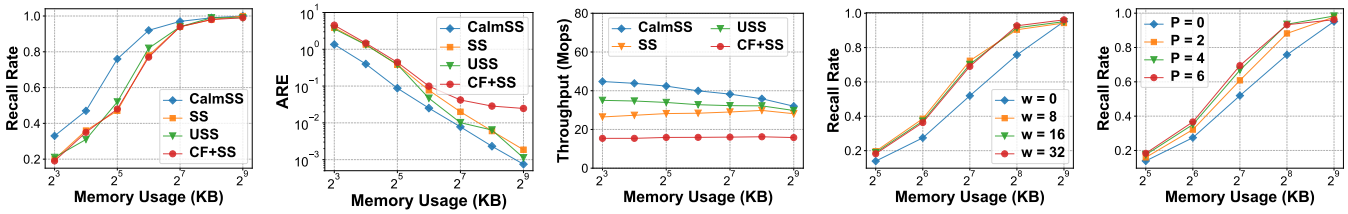
Processing speed (Figure 10e): *We find that HyperBF is faster than other algorithms.* The results show that under different memory constraints, the throughput of HyperBF is always 16 Mops, while that of TOBF and SWAMP are about 14 Mops and 9 Mops, respectively. The throughput of Clock drops rapidly 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 a half of that of HyperBF.

C. Experiments on CalmSS

Parameter setting: We compare CalmSS with Space-Saving (SS) [11], Unbiased Space-Saving (USS) [12], and Cold filter [16] + 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.

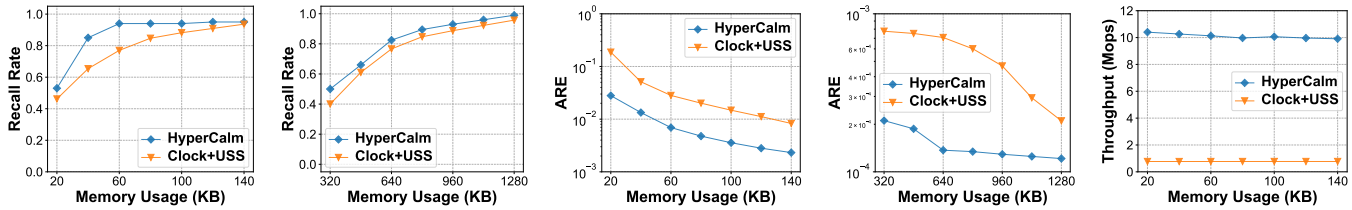
Accuracy of finding top-k items (Figure 11a): *We find that CalmSS always has better RR than SS, USS, and CF+SS.* The RR of CalmSS reaches 78% even if the memory size is only 32KB, while that of SS and USS are about 50%. As the memory size exceeds 128KB, the RR of CalmSS is very close to 100%. The RR 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 11b): *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.



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

Fig. 11: Performance of CalmSS (CAIDA).



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

Fig. 12: Performance of HyperCalm (CAIDA and Criteo).

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 11c): *We find that CalmSS is faster than SS, USS, 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 data structure. The LRU queue is small enough to be held in caches, and thus does not incur extra memory access.

Impact of LRU queue length (w) (Figure 11d): *We find that CalmSS performs well when the length of the LRU queue w is just 8.* When using 256KB of memory, the RR of CalmSS using an LRU queue of length $w = 8$ is 91%, while that of the baseline 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 11e): *We find that the optimal promotion threshold $\mathcal{P}$ is 4 or 6.* When using 256KB of memory, the RR of CalmSS with $\mathcal{P} = 2$ or $\mathcal{P} = 4$ is about 92%, while that of the baseline Space-Saving ($\mathcal{P} = 0$) is 76%. Note that the optimal $\mathcal{P}$ is highly correlated with the dataset.

D. 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 $l = 32$ by default. For CalmSS, we set $\mathcal{P} = 7$.

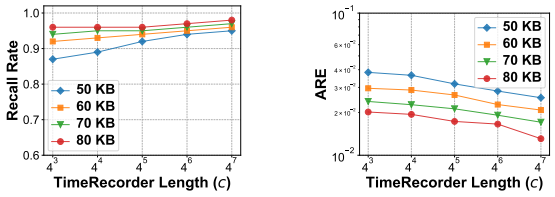
***1) Setting on CAIDA*:** We set the time-based batch threshold $\mathcal{T}$ to 0.072 millisecond. Each batch interval V is rounded to the nearest multiple of 0.72 millisecond. Under such settings, there are about 4.1M periodic batches in CAIDA dataset.

***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 12a-12b): *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%.

Frequency estimation of periodic batches (Figure 12c-12d): *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 12e): *We find that the processing speed of HyperCalm always outperforms the strawman solution on two datasets.* On CAIDA, when using 60KB of memory, the throughput of HyperCalm is 10.2 Mops, which is 13.2 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 harms the speed.



(a) Recall (CAIDA). (b) ARE (CAIDA).

Fig. 13: Impact of TimeRecorder queue length (c).

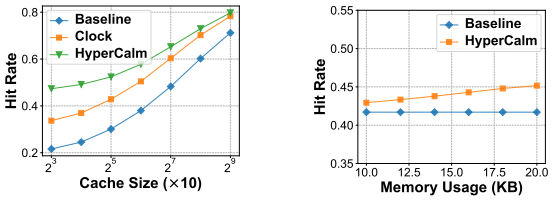
Impact of the length of TimeRecorder (c) (Figure 13): *We find that as the length of the TimeRecorder queue grows larger, the accuracy of HyperClam increases.* We also find

that HyperClam performs well when the TimeRecorder queue length $c = 1024$. As shown in Figure 13a, under 80KB of memory, when TimeRecorder queue length c is 1024, HyperClam achieves 97% RR. As shown in Figure 13b, under 80KB of memory, when TimeRecorder queue length c is 1024, the ARE of HyperClam is about 1.8×10^{-2}.

E. Applying HyperCalm to Cache Systems

We apply HyperCalm to a simulated cache system to showcase a promising application scenario, where the data stream is formed by many memory access requests. 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. Our experimental results show that with the help of HyperCalm, the hit rates of both LFU and LRU are significantly enhanced.

Implementation and datasets: We implement a fully associative cache simulator that mimics the behavior of a hardware cache. We use CAIDA [48] and treat source IP address (4 bytes) as memory access request. Note that the performance of caches is highly related to datasets. Only when the dataset includes many periodic batches (it usually does) can our solution improve hit rate. In practice, each cache entry consists of two fields: *key, value*. In many cases (MemC3, CDN, etc), the size of value field is very large. As HyperCalm does not store values, its size is only proportional to the number of periodic batches. Thus, its memory cost is very small.



(a) LFU. (b) LRU.

Fig. 14: Optimization to cache replacement policy.

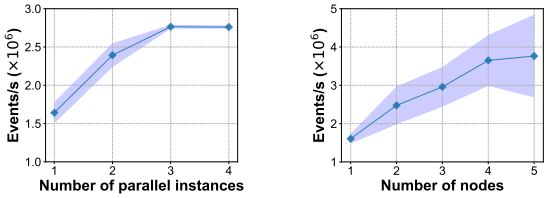
Experiments on LFU (Figure 14a): *We find that HyperCalm significantly improves the hit rate of LFU with small memory overhead, and HyperCalm outperforms Clock-Sketch [1] in both hit rate and processing speed.* Here, we compare HyperCalm with Clock-Sketch, which also uses the batch detection results to improve replacement policies. We set the memory usage of HyperCalm and Clock to 20KB, which is small compared to the cache storage. 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%, respectively.

Experiments on LRU (Figure 14b): *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. Note that although LRU is acknowledged as the best replacement algorithms, it has poor support for periodic requests. Thus, on our synthetic dataset where half of the requests arrive periodically, the hit rate of LRU is only 41%. When using 20KB of memory, HyperClam improves the hit rate from 41% to 45%.

F. Integration into Apache Flink

We implement HyperCalm on top of Apache Flink [22] showing that our solution can be easily integrated into modern stream processing framework and work in distributed systems.

Experimental setup: We run the experiments at a Flink cluster with 1 master and 5 workers using CAIDA [48] dataset. We deploy a Hadoop Distributed File System (HDFS) in our Flink cluster, where we set the master node as NameNode and the worker nodes as DataNodes. 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. The local experiments run in the local mode of the master node. All experiments are repeated 10 times and the average ($\pm$std) throughput is plotted.



(a) Local mode. (b) Cluster mode.

Fig. 15: Throughput on Apache Flink.

Experimental results (Figure 15): We find that HyperCalm can smoothly work on top of Flink framework. As shown in Figure 15a, 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 15b, in cluster mode, the throughput linearly scales up with more nodes used in the cluster.

VI. 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 periodic batches in real time. The two key components of HyperCalm, HyperBF and CalmSS, significantly outperform the 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. Further, we apply HyperCalm to a cache system and integrate HyperCalm into Apache Flink, showing the benefit of our approach to real world applications. In the future, we plan to deploy HyperCalm in more applications, and use our results to improve the performance of recommendation systems, financial markets, load balancing, *etc.*

REFERENCES

[1] Peiqing Chen, Dong Chen, Lingxiao Zheng, Jizhou Li, and Tong Yang. Out of many we are one: Measuring item batch with clock-sketch. SIGMOD, 2021.

[2] Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. Flowtune: Flowlet control for datacenter networks. In *NSDI*, 2017.

[3] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gábor Rétvári. Batchy: Batch-scheduling data flow graphs with service-level objectives. In *NSDI*, 2020.

[4] Runze Lei, Pinghui Wang, Rundong Li, Peng Jia, Junzhou Zhao, Xiaohong Guan, and Chao Deng. Fast rotation kernel density estimation over data streams. In *SIGKDD*, 2021.

[5] Craig Pirrong. Energy market manipulation: definition, diagnosis, and deterrence. *Energy LJ*, 31:1, 2010.

[6] Tong Chen, Hongzhi Yin, Hongxu Chen, Hao Wang, Xiaofang Zhou, and Xue Li. Online sales prediction via trend alignment-based multitask recurrent neural networks. *KAIS*, 2019.

[7] Z-L Zhang, Vinay J Ribeiro, Sue Moon, and Christophe Diot. Small-time scaling behaviors of internet backbone traffic: An empirical study. In *INFOCOM*, 2003.

[8] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[9] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[10] Yanqing Peng, Jinwei Guo, Feifei Li, Weining Qian, and Aoying Zhou. Persistent bloom filter: Membership testing for the entire history. In *SIGMOD*, 2018.

[11] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*. Springer, 2005.

[12] Daniel Ting. Data sketches for disaggregated subset sum and frequent item estimation. In *SIGMOD*, 2018.

[13] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Frequency estimation of internet packet streams with limited space. In *ESA*. Springer, 2002.

[14] Pratanu Roy, Arijit Khan, and Gustavo Alonso. Augmented sketch: Faster and more accurate stream processing. In *SIGMOD*, 2016.

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

[16] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *SIGMOD*, 2018.

[17] Peng Jia, Pinghui Wang, Junzhou Zhao, Ye Yuan, Jing Tao, and Xiaohong 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)*, pages 804–815. IEEE, 2021.

[18] Linfeng Zhang and Yong Guan. Frequency estimation over sliding windows. In *ICDE*, pages 1385–1387. IEEE, 2008.

[19] Ran Ben Basat, Roy Friedman, and Rana Shahout. Stream frequency over interval queries. *Proceedings of the VLDB Endowment*, 12(4):433–445, 2018.

[20] Shuhao Sun, Jingwei Zheng, and Dagang Li. Hee-sketch: an efficient sketch for sliding-window frequency estimation over skewed data streams. In *ISPA*, 2019.

[21] Hypercalm codes. https://github.com/HyperCalmSketch/HyperCalmSketch.

[22] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[23] Shijin Kong, Tao He, Xiaoxin Shao, Changqing An, and Xing Li. Time-out bloom filter: A new sampling method for recording more flows. In *ICOIN*, 2006.

[24] Eran Assaf, Ran Ben Basat, Gil Einziger, and Roy Friedman. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *INFOCOM*. IEEE, 2018.

[25] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1), 2005.

[26] Cristian Estan and George Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *TOCS*, 21(3), 2003.

[27] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*. 2002.

[28] Takuya Akiba and Yosuke Yano. Compact and scalable graph neighborhood sketching. In *SIGKDD*, 2016.

[29] Yang Yang, Ying Zhang, Wenjie Zhang, and Zengfeng Huang. Gb-kmv: An augmented kmv sketch for approximate containment similarity search. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 458–469. IEEE, 2019.

[30] Peng Jia, Pinghui Wang, Junzhou Zhao, Shuo Zhang, Yiyan Qi, Min Hu, Chao Deng, and Xiaohong Guan. Bidirectionally densifying lsh sketches with empty bins. In *SIGMOD*, 2021.

[31] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. A learned sketch for subgraph counting. In *SIGMOD*, 2021.

[32] Daniel Ting, Jonathan Malkin, and Lee Rhodes. Data sketching for real time analytics: Theory and practice. In *SIGKDD*, 2020.

[33] Pinghui Wang, Yiyan Qi, Yuanming Zhang, Qiaozhu Zhai, Chenxu Wang, John CS Lui, and Xiaohong Guan. A memory-efficient sketch method for estimating high similarities in streaming sets. In *SIGKDD*, 2019.

[34] Daniel Ting. Count-min: Optimal estimation and tight error bounds using empirical error distributions. In *SIGKDD*, 2018.

[35] Aécio Santos, Aline Bessa, Christopher Musco, and Juliana Freire. A sketch-based index for correlated dataset search. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022.

[36] Ran Ben Basat, Gil Einziger, Michael Mitzenmacher, and Shay Vargaftik. Salsa: self-adjusting lean streaming analytics. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 864–875. IEEE, 2021.

[37] Aécio Santos, Aline Bessa, Christopher Musco, and Juliana Freire. A sketch-based index for correlated dataset search. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022.

[38] Bohan Zhao, Xiang Li, Boyu Tian, and etal. Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *SIGKDD*, 2021.

[39] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In *SIDKDD*, 2020.

[40] K. Amphawan, P. Lenca, and A. Surarerks. Efficient mining top-k regular-frequent itemset using compressed tidsets. In *PAKDD*, 2011.

[41] Q. Wen, K. He, L. Sun, Y. Zhang, M. Ke, and H. Xu. Robustperiod: Time-frequency mining for robust multiple periodicities detection. 2020.

[42] M. G. Elfeky, W. G. Aref, and A. K. Elmagarmid. Stagger: Periodicity mining of data streams using expanding sliding windows. In *ICDM*, 2007.

[43] Quan Yuan, Jingbo Shang, Xin Cao, Chao Zhang, Xinhe Geng, and Jiawei Han. Detecting multiple periods and periodic patterns in event time sequences. In *CIKM*, 2017.

[44] M. Toller, T. Santos, and R Kern. Sazed: parameter-free domain-agnostic season length estimation in time series data. *DMKD*, (1), 2019.

[45] Zhuochen Fan, Yinda Zhang, Tong Yang, Mingyi Yan, Gang Wen, Yuhan Wu, Hongze Li, and Bin Cui. Periodicsketch: Finding periodic items in data streams. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022.

[46] Zhenhui Li, Bolin Ding, Jiawei Han, Roland Kays, and Peter Nye. Mining periodic behaviors for moving objects. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1099–1108, 2010.

[47] Erik Demaine, Alejandro López-Ortiz, and J Munro. Frequency estimation of internet packet streams with limited space. *Algorithms—ESA 2002*, 2002.

[48] CAIDA dataset. Available: http://www.caida.org/home.

[49] Murmur hashing source codes. https://github.com/aappleby/smhasher.

[50] Criteo dataset. Available: https://ailab.criteo.com/ressources/.