# Out of Many We are One: Measuring Item Batch with Clock-Sketch

Anonymous Author(s)

## ABSTRACT

Item batch denotes a consecutive sequence of identical items that are close in time in a data stream. It is a useful data stream pattern in cache, burst detection, APT detection ,*etc.* Basic item batch measurement tasks include membership, cardinality, time span and size. Currently, there is no algorithm tailored for item batch measurement. The greatest challenge lies in accurately estimating the time gap between two consecutive identical items. In this paper, we propose Clock-sketch, a framework that introduces the well-known CLOCK algorithm into item batch measurement. The methodology of Clock-sketch is to clean outdated information as much as possible, while guaranteeing that the information of all items visited within the time window $\mathcal{T}$ is preserved. We conduct experiments on three real-world datasets that feature in item batch pattern. We compare the accuracy and throughput performance of our Clock-sketch against the state-of-the-art and two naive approaches without using Clock-sketch technique. Results of item batch activeness show that Clock-sketch outperforms the state-of-the-art SWAMP in generating 50 times less false positive rate when memory is small. All source codes are open-sourced and released at Github.

## 1 INTRODUCTION

### 1.1 Background and Motivations

As the world grows interconnected, massive data transmission has become pervasive in financial markets [8, 18], social network [31] and web information retrieval [35]. Studying the pattern of data streams, *i.e.*, the internal correlation between items, can lead to better understanding of data. We focus on an important data stream pattern, namely item batch, and we are the first to study the pattern of item batch. [1] Given a threshold $\mathcal{T}$ and the time series of item $a$ (*i.e.*, the $i^{th}$ item $a_i$ arrives at $t_i$), if time gap between $(t_{i-1}, t_i)$ and $(t_j, t_{j+1})$ is larger than $\mathcal{T}$ and gap between $(t_k, t_{k+1})$ is smaller than $\mathcal{T}$ for any $k = i, i+1, ..., j-1$, then items $\{a_i, a_{i+1}, ..., a_j\}$ form a batch $B_a$. Given an item batch $B_a$, if all items have left for at least time $\mathcal{T}$, we call it inactive, and these information is outdated. Otherwise, we call it active. Basic item batch measurement includes activeness, cardinality (the number of active item batches at a time), time span, and size (the number of items contained). The following are four common cases where item batch measurements can make a real difference.

**Case 1 - Cache:** Cache flow is formed by a sequence of memory fetching requests. Each memory fetching request is an item. Two memory fetching requests with the same key are regarded as identical items. And an item batch is a continuous visit on the same key. Item batch measurement results can provide beneficial a knowledge for cache prefetching and help improve cache replacement policies. Firstly, item batch measurement can help prefetching and increase cache hit for periodical item batches. By observing the starting time and time span of each item batch, we are able to find item batches with periodical patterns. Therefore, prefetching an item from a periodical item batch into the cache can realize cache hit for all items in this item batch. Secondly, item batch measurement can help optimize Least Recently Used (LRU) principle. By knowing the sizes of active item batches, we can store larger item batches in caches while omitting smaller ones, though they may arrive more recently. On the other hand, LRU only preserves recent items in the cache, causing cache misses for larger item batches. Thirdly, item batch measurement can help optimize Least Frequently Used (LFU) principle. For example, LFU stores items in cache only depending on their historical frequency, and frequent items in the past will occupy cache memory and hold new items out. With the help of item batch measurement, we can find inactive item batches whose frequency records are still high and evict these items to make space for new incoming items. Another example is that LFU puts weight one on each incoming item. Thus items from larger item batches are not likely to be inserted into cache soon. With historical knowledge of the size of past item batches, we will be able to judge whether an incoming item belongs to a large item batch. If we change the weight of replacement from one to the size of its past item batches, larger incoming item batches will encounter fewer cache misses.

**Case 2 - Burst detection:** Burst detection in data streams is a hot topic and has wide applications in text stream mining [22], bursty topic mining [37], and trading volumes monitoring [40]. Current work include [22, 37, 40]. These work studies different burst detection problems. For example, [22] detects bursty topics in text stream based on a given topic, and [33] detects lasting and abrupt bursts in linear time. However, none of these work supports per-flow (all items identical to $a$ forms a per item flow of $a$) burst detection in real-time, which is an important topic in flow control and information retrieval. With the help of item batch measurements, we will be able to carry out per distinct item burst detection in real-time. For example, when detecting bursts in a financial transaction stream, each transaction is an item. Two transactions with the same sender are regarded as identical items. A simple approach is to define bursts as item batches with high density, *i.e.*, those with larger size but a smaller span. Further, by recording burst items, we will be able to find frequently appeared burst items, which can be reported and dealt with individually.

**Case 3 - APT detection:** Advanced Persistent Threat (APT) is a cyber threat that poses great harm to the information security of enterprises, government, and other organizations [16, 20, 34]. APT generally uses low frequency and small scale flows so that it is hard to be detected by traditional security measures. When detecting APT in a network data stream, each L-4 packet is an item. Two L-4 packets with the same header (*e.g.*, source IP, destination IP, source Port, destination Port, Protocol) are regarded as identical items. APT can be viewed as a kind of suspicious per item flow in

[1] The definition of batch in item batch is different from batch processing [11, 12] in machine learning, which means training the model with one group of data a time. Streaming algorithms for batch processing does not tell the difference between items with different ID within the same batch.

the data stream. APT features in small size per batch, long time gap between every two batches, and a large number of batches in total. With the help of item batch measurement, we will be able to detect suspicious flows and analyze this flow in the application layer.

**Case 4 - Online advertisement:** The online advertising is estimated to be a $230 billion industry [29]. The data stream is generated when customers click on different commodities through web pages. In the click stream of online advertisements, each click contains information of the customer's IP address, the commodity type and the clicking time, *etc.*. Each click is an item. Two clicks with the same customer's IP address and commodity type are regarded as identical items. An item batch is formed when continuous clicks on the same type of commodity happen. Such item batches imply the consumption habits of specific users: everlasting item batches indicates the user's enduring interest in specific commodity types, while new item batches indicate a new focus of a customer. Additionally, if we only study item batches belonging to a single customer, further information can be revealed: customers who only keep a few active item batches at a time are more focused on shopping, while those who keep a lot of active item batches simultaneously appear more aimless in shopping. Therefore, it would be more lucrative to deliver targeted advertisement of specific commodity types to the first kind of people and deliver a different of new products and popular series to the second kind of people.

From the above cases, it is clear that item batch is most helpful when it comes in real-time. For instance, in use case 1 of cache, real-time measurement results can help carry out replacement policy at once, thus minimizing cache miss; In use case 2 of burst detection, a collision of bursts may cause congestion in network. Real-time measurement results can help analyze the cycle of periodic bursts, and manage the traffic to avoid future burst collision.

## 1.2 Our Solution

The major challenges of item batch measurement are: 1. efficiently record comprehensive information of item batches with small time and space overhead 2. detect and clean out-dated information (*i.e.*, information of inactive batches) in time.. This calls for an estimation of the gap between every two consecutive identical items in the data stream. Currently, there is no algorithm tailored for item batch measurements. A straightforward solution is to use a circular queue to store all item IDs and time records (64 bits) of all items within a time window $\mathcal{T}$. Though this approach can provide an accurate result, it requires too much memory because of both the high volume of data streams and the large number of item batches. We aim to design a compact data structure that stores neither item IDs nor time records to fit into CPU caches [39].

In this paper, we introduce the well-known CLOCK [13] algorithm into item batch measurement for the first time, and propose a framework named Clock-sketch. The key idea of CLOCK is **conservative cleaning**: preserving information of all items that are visited within time window $\mathcal{T}$ and only outdated information can be cleaned. However, CLOCK still leaves outdated information, *i.e.*, information of inactive item batches. For example, when using one bit clock cell, CLOCK cleans all non-frequent items that are not visited within $2\mathcal{T}$ (detailed in Section 2.2), but may wrongly preserve items that are visited beyond $\mathcal{T}$ and within $2\mathcal{T}$. We define such time period as the error window. Items in the error window may generate false positive errors when querying the activeness of item batches.

The methodology of Clock-sketch is to clean outdated information as much as possible, while guaranteeing that the information of all items visited within time window $\mathcal{T}$ is preserved. To clean outdated information as much as possible, we need to reduce the impact of error window in item batch measurements. There are two strategies: enlarging the clock cell size and increasing the number of hash functions in the sketch. On the one hand, a larger clock cell shrinks the size of error window. Clock-sketch uses $s$-bit clock cell and accelerates the circular cleaning speed. This shrinks the error window from $\mathcal{T}$ to $\frac{\mathcal{T}}{2^s-2}$, far smaller than that of one-bit clock cells. On the other hand, more hash functions lower the impact of items in the error window. In a sketch, information of each item will be inserted into $k \geq 1$ randomly chosen cells which will be all used later for the query. Cleaning any of the $k$ cells will eliminate the existence/footprints of the item in the sketch. Given an item $a_i$, it will be preserved in the sketch for an extra time $\delta_t(a_i)$ beyond $\mathcal{T}$. In other words, the error incurred by item $a_i$ lasts $\delta_t(a_i)$. $\delta_t(a_i)$ is related to the distance between the hash positions and position of the clock hand. Using more hash functions will lead to a smaller expectation of $\delta_t(a_i)$ (see Figure 3). In summary, more clock bits (larger $s$) requiring more memory usage leads to smaller error window size; more hash functions (larger $k$) requiring more memory usage, leading to smaller $\delta_t$ of each item in the error window. Therefore, given a fixed size of memory, there is a balance of parameters $s$, $k$ to minimize the measurement error. In this paper, we study the best choice of parameters $s$ and $k$ through mathematical proofs in Section 5 and verify it by experiments in Section 6.

## 1.3 Main Experimental Results

We test our Clock-sketch on three real-world datasets which feature in item batch pattern and carried out experiments on both count-based and time-based item batch definition. In measuring item batch activeness, our Clock-sketch outperforms the state-of-the-art SWAMP in generating 50 times less false positive rate, and maintaining 50% higher insertion throughput and equivalent query throughput. In measuring item batch cardinality, Clock-sketch can outperform SWAMP by reaching less than $10^3$ relative error when memory is small. In measuring time span and size, due to the lack of comparable algorithms, we design two naive approaches without using Clock-sketch framework. Results show that Clock-sketch algorithms can outperform the naive approach in generating 10 times less error when memory is small. Moreover, Clock-sketch can achieve comparable, if not higher, throughput in all above four tasks. All related codes are open-sourced and available at Github anonymously [5].

# 2 RELATED WORK

## 2.1 Item Batch Measurement

Currently, no work is specially designed for item batch measurement. However, some algorithms for sliding window measurement can be introduced into activeness and cardinality task of item batch measurement, among which TBF [38], TOBF [23] fit activeness measurement, CVS [28], TSV [21] fit cardinality measurement, while SWAMP [7] fits both.

#### 2.1.1 Algorithms for item batch activeness.

**The Time-Out Bloom Filter (TOBF) [23]** uses an array of timestamps. For insertion, it sets the $k$ hashed locations as current time $t_{cur}$. For query, if any of the $k$ hashed timestamp is inactive (*i.e.*, earlier than $t_{cur} - \mathcal{T}$), it reports an inactive item batch. Otherwise, it returns true.

**The Timing Bloom filter (TBF) [38]** uses a wraparound counter array to record arrival time instead of recording timestamps directly. Every time an item is inserted, TBF scans a piece of the array to remove inactive time records.

**Sliding Window Approximate Measurement Protocol (SWAMP) [7]** uses a cyclic queue and a Tiny Table[17]. The cyclic queue records the fingerprints of the latest $w$ items. The Tiny Table records the frequency of distinct items in the latest $w$ items. For insertion, the oldest fingerprint in the cyclic queue is replaced tuples concerned are updated in the Tiny Table.

#### 2.1.2 Algorithms for item batch cardinality.

**The Counter Vector Sketch (CVS) [28]** uses an array of totally $n$ counters. For insertion, it sets the values of $k$ hashed counters to maximum $c$. Afterward, it randomly chooses several counters and decrements them by 1. For query, if $u$ among all $n$ counters are non-zero, the reported cardinality shall be $uln\frac{n}{u}$. CVS falls short in the error induced by the randomness in picking counters to decrement.

**The Timestamp-Vector algorithm (TSV) [21]** uses an array of totally $n$ timestamps. For insertion, it sets the hashed counters to current time $t_{cur}$. For query, it counts the number of active timestamps (*i.e.*, timestamps no earlier than $t_{cur} - \mathcal{T}$) in the array, denoted as $u$. The reported cardinality is also $uln\frac{n}{u}$

### 2.2 CLOCK

CLOCK [13] is a classic cache policy proposed by Frank Corbat'o *et al.*. CLOCK has wide applications in operating systems [14, 19, 24], databases, and file systems [9]. Cache memory is commonly organized as an array of uniformly-sized units known as page counters, just suited for storing a page extracted from the external memory. CLOCK views this array of pages as a cyclic queue and attaches a reference bit [13] to each page counter. What's more, CLOCK uses an additional thread called clock hand to cyclically update each reference bit and corresponding page counter information that it traverses. Suppose the cycle of clock hand traversing is $\mathcal{T}$. Pages that are visited again within $\mathcal{T}$ time will always get a cache hit, while pages not visited again beyond $2\mathcal{T}$ time will be automatically cleaned from the cache, clearing space for other incoming pages.

The clock hand update and memory fetch events are independent. The clock hand update is performed at a predefined constant speed (*i.e.*, $\mathcal{T}$ for a cycle). In the meantime of a memory fetch, the cleaning process is still performing. In order to save space, we suppose both the memory fetch and the clock hand update happen at time $t_1$ and $t_2$ in Figure 1. At time $t_1$, a cache hit happens at counter 5 and the clock hand traverses counter 0. Upon the cache hit, the reference bit of counter 5 updates to 1, while the information in the page counter remains. Upon the clock hand traversal, the clock hand meets a counter whose reference bit is 1. It changes the reference bit into 0, and the page counter remains intact. At time $t_2$, a memory fetch is directed to counter 7 and the clock hand traverses counter 3. Upon the memory fetch, a new page $page_{32}$ is fetched from the external memory. The page counter stores $page_{32}$ and the attached reference bit is set to 1. Upon the clock hand traversal, the clock hand meets a counter whose reference bit is 0. It clears the page information stored in page counter 3.

## 3 ALGORITHM

We first show the rationale the algorithm design of Clock-sketch. To address the challenge of cleaning outdated information, Clock-sketch uses small clock cells to serve as counting downs rather than using large 64-bit timestamps. Using small clock cells, we can safely and efficiently evict out-dated information, while preserving all items within a time window. Clock cells inevitably bring about error in cleaning outdated information, noted as the time window error (window error for short). To shrink the impact of window error, we use multiple bits in each clock cell. Important notations used in the rest of the paper are demonstrated in Table 2.

### 3.1 Preliminaries and Problem Statement

**Preliminaries on item batch model:** Item batch is a data stream model. A data stream is an infinite sequence of items (allowing duplication). An item batch is defined as a group of same items in the data stream, where the time gap between each two adjacent items is below a predefined threshold $\mathcal{T}$. Threshold $\mathcal{T}$ can be either time-based (containing items in a time window with $\mathcal{T}$ time units, *e.g.*, 10 ms) or count-based (containing $\mathcal{T}$ items, *e.g.*, $1,000,000$ items). These two kinds of definitions are equal when the data stream passes at a constant speed.

**Common sketch model:** In recent years, sketches have become popular data structures to deal with data stream [25–27, 30, 32]. Our Clock-sketch can work with a series of sketch algorithms, which we summarize as a common sketch model. The common sketch model is an array of cells (called sketch cells). Each cell can be either a bit, a counter, a timestamp, or a combination of them. Several hash functions are used to hash each item into several cells, called hashed cells. To insert an item into the sketch, the hashed cells of this item are updated. The query of an item depends on all information stored in all hashed cells.

**Table 1: Notations used in the rest of the paper.**

| Notation | Meaning |
|---|---|
| $a$ | an item in the data stream |
| $a$ | the item batch that $a$ belongs to |
| $t_{cur}$ | current time |
| $\mathcal{T}$ | size of a sliding window |
| $n$ | number of sketch\clock cells in a sketch |
| $sc[i]$ | the $i^{th}$ sketch cell in the sketch |
| $cc[i]$ | the $i^{th}$ clock cell in the clock array |
| $k$ | number of hash functions |
| $s$ | number of bits in a clock |

### 3.2 Clock-Sketch Framework

**Data Structure:** Clock-sketch is a common sketch model with a clock-like structure. Each sketch cell in the model is attached with an extra $s$ bit cell, named as the clock cell. The cell array is viewed

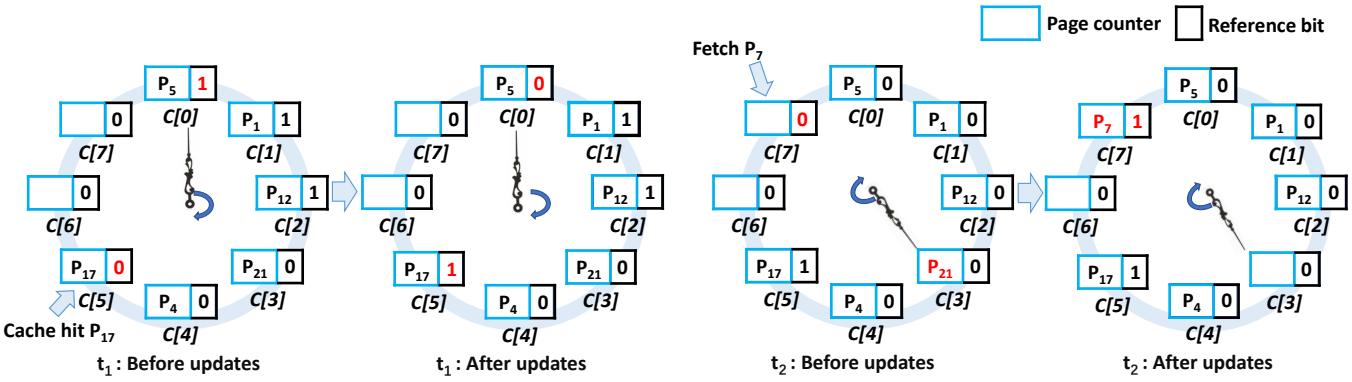



**Figure 1: Insertions and updates of basic CLOCK (one reference bit version)**

as a cyclic queue, and a cleaning pointer sweeps through the clock cells as time passes. The value in a clock cell is initialized as the maximum value, $2^s - 1$, and will decrement by 1 when the pointer sweeps through it. Different from CLOCK [13], we preserve the zero value of clock cells as the valid flag. When the value is decremented to zero, we think the sketch cell is invalid and clean the stored information. Figure 4 gives an example of how Clock-sketch works:

**Insertion:** Given an incoming item, we first calculate the hash functions and get the hashed cells, then update the cells. The sketch cells are updated according to the sketch model, and the clock cells are set to $2^s - 1$.

**Query:** The query process is the same as the sketch model. We only refer to the information stored in sketch cells.

**Cleaning process:** A cleaning pointer cyclically sweeps through the clock cells. Specifically, the cleaning pointer first points at the first cell in the array and sweeps through each clock cells one by one. After the last cell of the array is swept through, the pointer will point at the first cell again. For each clock cell, the pointer decrements the value by 1. When the value of a clock cell reaches zero, we clean the information stored in the corresponding sketch cell. The cleaning process is parallel to the insertion or query. For a time-based Clock-sketch, a clock cell is processed after a certain number of time units pass, and for a count-based Clock-sketch, it is processed after a certain number of items are inserted. For a sliding window with size of $\mathcal{T}$, to make sure that items in the time window will not be cleaned, the time for the pointer to sweep through the whole array should be $\frac{\mathcal{T}}{2^s-2}$.

## 3.3 Error Analysis

The error in Clock-sketch is generated mainly from two aspects. The first aspect is the hash collision between active items, which is the intrinsic flaw brought by the sketch model. If all hashed cells of an item are the hashed cells of other items, then the stored information of it may be overwritten or biased. [2]

The second aspect is the effect of outdated items that is not cleaned timely. To guarantee that any item in the time window will not be cleaned, the hashed cells of each item can be swept through by the cleaning pointer no more than $s^2 - 2$ times. As a result, the size of the time window should be at most $s^2 - 2$ times the size of the time for the cleaning pointer to circle the array. However, the items appear in a small time span before the time window is outdated, but their hashed cells may not be swept through $s^2 - 1$ times, so that they may not be considered outdated. The time span before the sliding time window is called as the error window, and its size is the time for the pointer to circle the array (*i.e.*, $\frac{\mathcal{T}}{2^s-2}$). Figure 2 is an example of outdated items. Item *a* only appears in the error window, and may incur error in the measurement. Item *b* only appears in the standard window and has no outdated information. Item *c* appears in both the error window and the standard window, which may incur error in some applications (*e.g.*, item batch activeness measurement) while not in other applications (*e.g.*, item batch size measurement).

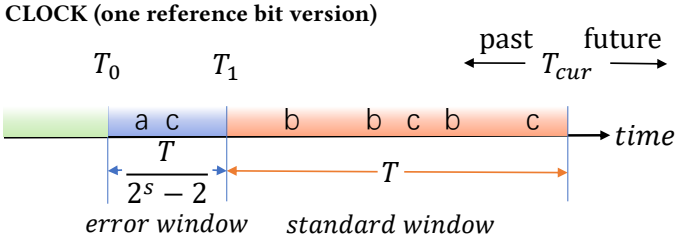



**Figure 2: Impact analysis of items in the error window**

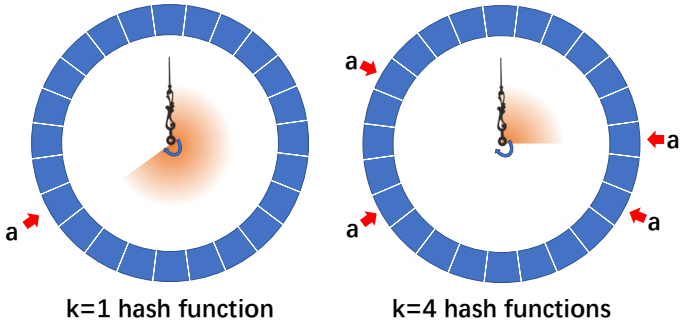



**Figure 3: Impact of the number of hash functions $k$**

**Effect of parameter $k$:** Figure 3 gives an example of an item in the error window for a Clock-sketch using $k = 1$ hash functions or using $k = 4$ hash functions. The item in the error window will not be cleaned as long as any one of $k$ hashed cells is not swept through by the cleaning pointer in the last circle. Suppose the cleaning pointer sweeping through one of the item's hashed cell for the first time

[2] **The Bloom filter** [10] uses an array of bits which are initially 0. There are 5 hash locations for each item in the array. When an item *a* arrives, it sets all 5 bits to 1. For an item which hasn't appeared, if only 3 of its hashed locations are set to 1 by other items, the algorithm still returns false because the remaining 2 bits are still 0. However, if all 5 bits are set to 1 due to hash collision, the algorithm will return yes, which is a biased answer.

happens $delta_t$ time after the item is inserted, then the item will be kept in Clock-sketchfor $\mathcal{T} + delta_t$. The larger $k$ is, the smaller $delta_t$ is expected to be, and there is less possibility that a outdated item is still kept in Clock-sketch.

**Effect of parameter $s$:** $s$ is the number of bits for each clock cell. A larger $s$ can reduce the size of the error window, thus reduce the error from outdated items. However, for a fixed memory of Clock-sketch, a larger $s$ means that each clock cell consumes more memory, and the number of cells should be reduced to meet the memory requirement. For the sketch model, the smaller the number of cells is, the larger the possibility of a hash collision is.

Under the restriction of limited memory and a fixed sliding window size, we want to select the optimal $k$ and $s$ to minimize the error in measurements. In Section 5, we show how to pick the optimal $k$ and $s$ for specific measurement tasks.

# 4 APPLICATION

In this section, we showcase how clock framework can work together with the sketch data structure for basic item batch measurements. In order to save space, we only demonstrate pseudocode of item batch cardinality. The pseudocode of all algorithms are released in the technical report in [6]

## 4.1 Item Batch Activeness

Item batch activeness denotes whether or not an item batch is currently active. Specifically, an item batch $B_a$ is active if and only if an item $a$ shows up within the past time window $\mathcal{T}$. We use the Bloom filter[10] with Clock-sketch framework to detect item batch activeness. We attach an $s$-bit clock cell to each cell (*i.e.*, a bit) in the Bloom filter. Noticing the fact that each cell is 1 if and only if its corresponding clock cell is non-zero and vice versa, we can omit the cell array and save one bit for each bucket. Therefore, a query can be conducted based on the zeroness and nonzeroness of the clock array.

**Data structure:** It contains an $n$-clock cell array, $cc[0]$, $cc[1]$, ..., $cc[n-1]$ and $k$ hash functions $H_1, H_2, ..., H_k$. Each clock cell is composed of $s$ bits. All clock cells are initially set to 0.

**Insert:** When an item $a$ arrives at the current time $t_{cur}$, we calculate $k$ hashed locations $l_1, l_2, ..., l_k$ by $l_i(a) = H_i(a)\%n (i = 1, 2, ..., k)$. Then we update the value to $2^s - 1$ at the $l_1^{th}$, $l_2^{th}$, ..., $l_k^{th}$ clock cells. Besides, we use an additional thread to circularly scans the whole array at the speed of $\frac{\mathcal{T}}{2^s-2}$ per cycle. The thread decreases any clock cell with a positive value by one each time.

**Query:** To query the activeness of item batch $B_a$, we find the $k$ clock cells corresponding to item $a$. If all clock cells have non-zero values, the return value is positive, *i.e.*, $B_a$ is an active item batch. Otherwise, the return value is negative, *i.e.*, $B_a$ is an inactive item batch.

## 4.2 Item Batch Cardinality

Item batch cardinality denotes the number of currently active item batches. Specifically, if there are $x$ different items within the past time window $\mathcal{T}$, the current item batch cardinality is $x$. We use the Bitmap[36] with Clock-sketch framework to measure item batch cardinality. We attach an $s$-bit clock cell to each cell (*i.e.*, a bit) in the Bitmap. Same to the Bloom filter, we can omit the original cell of the Bitmap and only depend on the zeroness and nonzeroness of the clock cell array during query.

**Algorithm 1:** Item Batch Activeness

```
Insert:
  Input: An item a,current time t_cur
for i = 1 to k do
    l_i[a] = H_i[a]%n;
    cc[l_i[a]] = 2^s − 1;
Return;
Query:
  Input: An item a,current time t_cur
  Output: Whether the item a showed up within the past
          time window
for i = 1 to k do
    l_i[a] = H_i[a]%n;
    if cc[l_i[a]] = 0 then
        Return False;
Return True;
Refresh: ptr = 0;
while 1 do
    if cc[ptr] > 0 then
        cc[ptr] = cc[ptr] − 1;
    ptr = (ptr + 1)%n;
    wait(T/(2^s−2));
```

**Data structure:** It contains an $n$-clock cell array, $cc[0]$, $cc[1]$, ..., $cc[n-1]$ and one hash function $H_1$. Each clock cell is composed of $s$ bits. All clock cells are initially set to 0.

**Insert:** When an item $a$ arrives at the current time $t_{cur}$, we calculate one hash location $l_1$ by $l_1(a) = H_1(a)\%n$. Then we update the value to $2^s - 1$ at the $l_1^{th}$ clock cells. Besides, we use an additional thread to circularly scans the whole array at the speed of $\frac{\mathcal{T}}{2^s-2}$ per cycle. The thread decreases any clock cell with a positive value by one each time.

**Query:** To query the cardinality of item batch $B_a$, we count the numbers of clock cells whose value are 0, denoted as $u$.The Bitmap gives the maximum likelihood estimation to data stream cardinality as: $-n \ln \frac{u}{n}$.

## 4.3 Item Batch Time Span

Item batch time span denotes the time gap between the first item of $B_a$ and the current time $t_{cur}$. We remodel the Bloom filter[10] to fit item batch time span measurement. By changing each sketch cell of the Bloom filter from a bit to a 64-bit timestamp, we will be able to record the arriving time of the first item $a$ in item batch $B_a$.

**Data structure:** It contains an $n$-clock cell array, $cc[0]$, $cc[1]$, ..., $cc[n-1]$, $n$-sketch cell array, $sc[0]$, $sc[1]$, ..., $sc[n-1]$ and $k$ hash functions $H_1, H_2, ..., H_k$. Each clock cell is composed of $s$ bits and each sketch cell is a 64-bit timestamp. All clock cells are initially set to 0 and all sketch cells are initially set to 0.

**Insert:** When an item $a$ arrives at the current time $t_{cur}$, we calculate $k$ hashed locations $l_1, l_2, ..., l_k$ by $l_i(a) = H_i(a)\%n (i = 1, 2, ..., k)$. Then we update the value to $2^s - 1$ at the $l_1^{th}$, $l_2^{th}$, ..., $l_k^{th}$ clock cells.

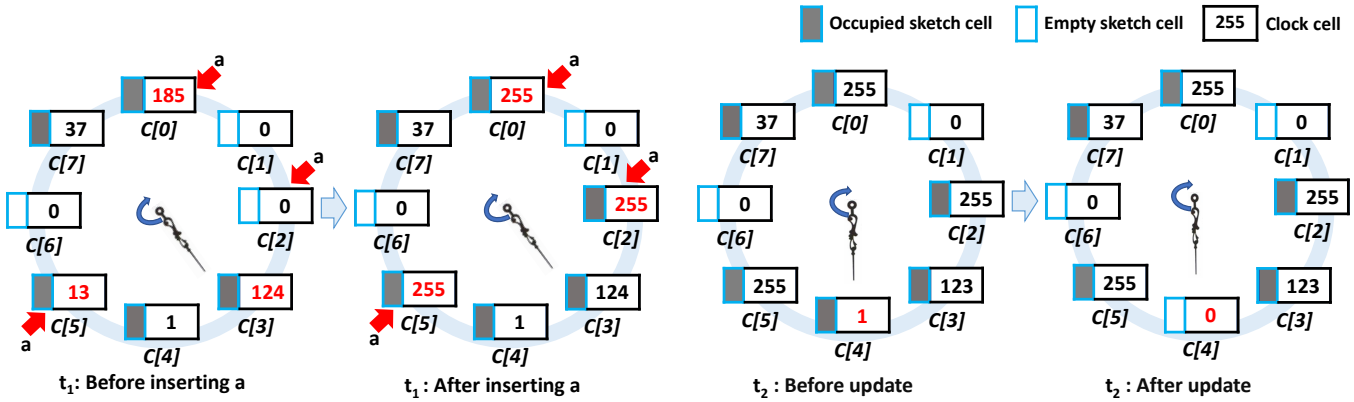



**Figure 4: Insertion and update of Clock-sketch, using 8-bit clock cell**

**Algorithm 2:** Item Batch Cardinality

**Insert:**
**Input:** An item $a$,current time $t_{cur}$
**for** $i = 1$ *to* $k$ **do**
  $l_i[a] = H_i[a]\%n$;
  $cc[l_i[a]] = 2^s - 1$;
Return;
**Query:**
**Output:** the cardinality of item batch
$u = 0$;
**for** $i = 1$ *to* $n$ **do**
  **if** $cc[i] = 0$ **then**
    $u = u + 1$;
Return $-nln\frac{u}{n}$;
**Refresh:**
$ptr = 0$;
**while** 1 **do**
  **if** $cc[ptr] > 0$ **then**
    $cc[ptr] = cc[ptr] - 1$;
  $ptr = (ptr + 1)\%n$;
  wait($\frac{T}{2^s-2}$);

**Algorithm 3:** Item Batch Time Span

**Insert:**
**Input:** An item $a$,current time $t_{cur}$
**for** $i = 1$ *to* $k$ **do**
  $l_i[a] = H_i[a]\%n$;
  $cc[l_i[a]] = 2^s - 1$;
  **if** $sc[l_i[a]] = 0$ **then**
    $sc[l_i[a]] = t_{cur}$
Return;
**Query:**
**Input:** An item $a$,current time $t_{cur}$
**Output:** the time span of item batch $B_a$
$t_{begin} = 0$;
**for** $i = 1$ *to* $k$ **do**
  $l_i[a] = H_i[a]\%n$;
  $t_{begin} = max\{t_{begin}, sc[l_i[a]]\}$
Return $t_{cur} - t_{begin}$;
**Refresh:**
$ptr = 0$;
**while** 1 **do**
  **if** $cc[ptr] > 0$ **then**
    $cc[ptr] = cc[ptr] - 1$;
    **if** $cc[ptr] = 0$ **then**
      $sc[ptr] = 0$;
  $ptr = (ptr + 1)\%n$;
  wait($\frac{T}{2^s-2}$);

If the sketch cell $sc[l_i^{th}]$ is 0, then we set it to the current time $t_{cur}$. Besides, we use an additional thread to circularly scans the whole array at the speed of $\frac{\mathcal{T}}{2^s-2}$ per cycle. The thread decreases any clock cell with a positive value by one each time. Once the value of $cc[i](i = 1, 2, ..., n)$ is decreased to 0, then we set 0 to the $sc[i](i = 1, 2, ..., n)$.

**Query:** To query the time span of item batch $B_a$, we find the $k$ hashed clock cells and sketch cells corresponding to item $a$ at the current time $t_{cur}$. If all clock cells have non-zero values, the life span exists. Then we get the newest time (*i.e.*, time closest to $t_{cur}$) $t_{begin}$ in the $k$ sketch cells as the beginning time of the item batch $B_a$. The time span can be represented by the value of $t_{cur} - t_{begin}$.

## 4.4 Item Batch Size

Item batch size denotes the number of items in a currently active item batch $B_a$. Specifically, it measures the number of duplicate items $a$ between the arrival of the first item in $B_a$ and the current time. We use the Count-Min sketch [15] with clock framework to measure item batch size. We attach an $s$-bit clock cell to each sketch cell (*i.e.*, a counter) in the Count-Min sketch. Information of count values in cells remains valid as long as their corresponding clock cell values are non-zero. Therefore, a cell is able to record the number of items within an item batch. On the other hand, when the clock cell counts down to zero, the information in the sketch cell is immediately erased, indicating the end of an item batch.

**Data structure:** It contains an $n$-clock cell array, $cc[0]$, $cc[1]$, ..., $cc[n-1]$ and an $n$-sketch cell array, $sc[0]$, $sc[1]$, ..., $sc[n-1]$ and

$k$ hash functions $H_1$,$H_2$,...,$H_k$. Hash function $H_i$ is attached to $i^{th}$ $n$-counter array ($i = 1, 2, ..., d$). Each clock cell is composed of $s$ bits, and each sketch cell is a counter. The size of the counter depends on the actual size of the window in applications. All clock cells and sketch cells are initially set to 0.

**Insert:** When an item $a$ arrives at the current time $t_{cur}$, we calculate $k$ hashed locations $l_1, l_2, ..., l_k$ by $l_i(a) = H_i(a)\%n(i = 1, 2, ..., k)$. Then we increase the value of the $cc[l_i](i = 1, 2, ..., k)$ by 1. And we update the value to $2^s - 1$ at clock cell $cc[l_i](i = 1, 2, ..., d)$. Besides, we use an additional thread to circularly scans the whole array at the speed of $\frac{\mathcal{T}}{2^s-2}$ per cycle. The thread decreases any clock cell with a positive value by one each time. Once the value of the $cc[j](i = 1, 2, ..., n)$ is decreased to 0, we set 0 to the sketch cell $sc[j](i = 1, 2, ..., n)$.

**Query:** To query the size of item batch $B_a$, we find the $k$ sketch cells corresponding to item $a$. Firstly,we calculate $k$ hashed locations $l_1, l_2, ..., l_d$ by $l_i(a) = H_i(a)\%n(i = 1, 2, ..., d)$. Then we get the $k$ sketch cells $sc[l_i](i = 1, 2, ..., k)$. The minimal value among the $k$ counters is the estimation of the item batch size.

**Algorithm 4:** Item Batch Size

**Insert:**
**Input:** An item $a$,current time $t_{cur}$
**for** $i = 1$ *to* $k$ **do**
  $l_i[a] = H_i[a]\%n$;
  $cc[l_i[a]] = 2^s - 1$;
  $sc[l_i[a]] = sc[l_i[a]] + 1$;
Return;
**Query:**
**Input:** An item $a$
**Output:** the size of item batch $B_a$,current time $t_{cur}$
$size = Infinity$;
**for** $i = 1$ *to* $k$ **do**
  $l_i[a] = H_i[a]\%n$;
  $t_{begin} = min\{size, sc[l_i[a]]\}$
Return $size$;
**Refresh:**
$ptr = 0$;
**while** 1 **do**
  **if** $cc[ptr] > 0$ **then**
    $cc[ptr] = cc[ptr] - 1$;
    **if** $cc[ptr] = 0$ **then**
      $sc[ptr] = 0$;
  $ptr = (ptr + 1)\%n$;
  wait($\frac{T}{2^s-2}$);

# 5 MATHEMATICAL ANALYSIS

We demonstrate how to choose the best clock size $s$ in the above four applications. We give an analysis on count-based window $\mathcal{T}$, which is same to time-based window when all items in the data stream arrive at a constant speed. The time span of a stream is subject to an exponential distribution with parameter $\lambda_1$. The number of new streams that are generated in one unit of time is $n_0$, and the size of a stream subject to an exponential distribution with parameter$\lambda_2$.

## 5.1 Item Batch Membership

Item batch membership denotes whether an item belongs to a current active item batch. We use BF+clock to measure it. Suppose that the cell number of BF+clock is $n$, the number of hash functions is $k$, the size of the time window is $\mathcal{T}$, and the number of bits per cell is $s$. Our goal is to minimize the false positive rate $f(s)$ while the memory $M = ns$ is fixed.

Considering the interruption of outdated elements, the total number of active elements is $\mathcal{T}(1 + \frac{1}{2^s-2})$. Since only half of the hash mappings of outdated elements are valid (not cleaned up), there are $k\mathcal{T}(1 + \frac{1}{2(2^s-2)})$) valid hash mappings in total, So the FPR is:

$$f(s) = (1 - (1 - \frac{1}{n})^{k\mathcal{T}(1+\frac{1}{2(2^s-2)})})^k \approx (1 - e^{-\frac{k\mathcal{T}(1+\frac{1}{2(2^s-2)})}{n}})^k \quad (1)$$

Similar to the usual bloom filter, the optimal k is $\frac{nln2}{\mathcal{T}(1+\frac{1}{2(2^s-2)})}$. Then

$$f(s) \approx 2^{-k} = 2^{-\frac{nln2}{\mathcal{T}(1+\frac{1}{2(2^s-2)})}} \quad (2)$$

Plug $n = \frac{M}{s}$, and we get

$$f(s) \approx 2^{-k} = 2^{-\frac{Mln2}{\mathcal{T}s(1+\frac{1}{2(2^s-2)})}} \quad (3)$$

Since $s = 2, 3, ...$, it is obvious that f(s) gets its minimum when $s = 2$, which is:

$$f^* \approx 2^{-k} = 2^{-\frac{3Mln2}{8\mathcal{T}}} \approx 0.8351^{\frac{M}{\mathcal{T}}} \quad (4)$$

For comparison, the FPR of TBF is:

$$g = O(0.6185^{\frac{M}{\mathcal{T}log\mathcal{T}}}) \quad (5)$$

We can see that our algorithm is superior to TBF by a scale of $log\mathcal{T}$.

From equation (4), in order to achieve an FPR of $\epsilon$, the memory our algorithm needs is:

$$M_1(\epsilon) \approx \frac{8}{3ln2}\mathcal{T}log_2\frac{1}{\epsilon} \approx 3.8472\mathcal{T}log_2\frac{1}{\epsilon} \quad (6)$$

For comparison, in order to achieve a FPR of $\epsilon$, the memory that SWAMP needs is:

$$M_2(\epsilon) > \mathcal{T}log_2\frac{\mathcal{T}}{\epsilon} \quad (7)$$

Our algorithm is also superior to SWAMP by a scale of $log\mathcal{T}$. Let $\mathcal{T} = 2^{16}$

$$M_2(\epsilon) > 16\mathcal{T}log_2\frac{1}{\epsilon} \quad (8)$$

So we can see that our algorithm is significantly superior to TBF.

## 5.2 Item Batch Cardinality

Item batch cardinality denotes how many different members have arrived during the last time window. We use Bitmap+clock to measure it. Assume that the cell number of Bitmap+clock is $n$, the number of hash functions is $k$, the size of time window is $\mathcal{T}$, and the number of bits per cell is $s$. Our goal is to minimize the relative error $RE(s)$ while the memory $M = ns$ is fixed. Considering

the interruption of outdated elements, an upper bound of the total number of active elements is

$$m_0 = m(1 + \frac{1}{2^s - 2}) \tag{9}$$

Assume that the number of 0 in Bitmap is $u$, then the output of our algorithm $m_1$ is:

$$m_1 = -n\ ln\frac{u}{n} \tag{10}$$

It is easy to see that the probability of "a certain cell of Bitmap is 0" is $(1 - 1/n)^{m_0} \leq e^{-\frac{m_0}{n}}$, so

$$ne^{-\frac{m}{n}} \leq E(u) \leq ne^{-\frac{m_0}{n}} \tag{11}$$

Therefore, the probability of "the relative error of a single measure is greater than $(\frac{1}{2^s-2} + \epsilon)$" $P(s, \epsilon)$ satisfies :

$$\begin{aligned} P(s,\epsilon) &\leq Pr(|m_1 - m_0| > \epsilon m) \\ &\leq Pr(|u - E(u)| > ne^{-\frac{m}{n}(1+\frac{1}{2^s-2})}(1 - e^{-\frac{m\epsilon}{n}}) \\ &\leq Pr(|u - E(u)| > n\frac{\epsilon}{4}) \end{aligned} \tag{12}$$

Use Hoeffding Bound, and we see that:

$$P(s,\epsilon) \leq Pr(|u - E(u)| > n\frac{\epsilon}{4}) \leq 2e^{-\frac{n\epsilon^2}{8}} \tag{13}$$

Let $\epsilon = \sqrt{\frac{8}{n} ln(\frac{2}{\delta})}$, then the equation below satisfies with a probability of not less than $1 - \delta$:

$$RE(s) \leq \frac{1}{2^s - 2} + \sqrt{\frac{8}{n} ln(\frac{2}{\delta})} \tag{14}$$

Plug $n = \frac{M}{s}$:

$$RE(s) \leq \frac{1}{2^s - 2} + \sqrt{\frac{8s}{M} ln(\frac{2}{\delta})} \tag{15}$$

## 5.3 Item Batch Time Span

Item batch time span denotes how long an active item batch has last. We use BF+clock with timestamp to measure it. Assume that the cell number of BF+clock with timestamp is $n$, the number of hash functions is $k$, the size of the time window is $\mathcal{T}$, the number of bits per cell is $s$, and the number of bits per timestamp is $t$ (in our experiment, $t = 64$). Our goal is to minimize the error rate $f(s)$ given that the memory $M = n(s + t)$ is fixed. In BF+clock with timestamp the error rate consists of two parts: the first part $f_1(s)$ is caused by hash collision, and the second part $f_2(s)$ is caused by the interruption of outdated elements. Then:

$$F(s) \leq f_1(s) + f_2(s) \tag{16}$$

Assume that when the generation and extinction of streams come into a balance, the number of active streams is $x$, then the average number of streams that disappears in one unit of time is $\lambda_1 x$ due to the property of exponential distribution. So $\lambda_1 x = n_0$, that is

$$x = \frac{n_0}{\lambda_1} \tag{17}$$

First we consider $f_2(s)$. $f_2(s)$consists of two parts :

a) The stream generated before $(1 + \frac{1}{2^s-2})\mathcal{T}$ time ago, and disappears in the time range of $(\mathcal{T}, (1+\frac{1}{2^s-2})\mathcal{T}]$. Since there are $x$ active streams at $(1+\frac{1}{2^s-2})\mathcal{T}$ time. Assume the number of streams that disappear during that period is $x_1$, then

$$E[x_1] = x(1 - e^{-\frac{\lambda_1 \mathcal{T}}{2^s-2}}) \tag{18}$$

b) The stream both generates and disappears in the time range of $(\mathcal{T}, (1 + \frac{1}{2^s-2})\mathcal{T}]$. Assume the number of streams that disappear during that period is $x_2$, then

$$E[x_2] = \sum_{i=0}^{\frac{\mathcal{T}}{2^s-2}-1} n_0(1 - e^{-\lambda_1(\frac{\mathcal{T}}{2^s-2}-i)}) \approx \frac{\mathcal{T}}{2^s - 2} - \frac{1 - e^{-\lambda_1 \frac{\mathcal{T}}{2^s-2}}}{\lambda_1} \tag{19}$$

The probability of "the stream satisfying either of the two conditions above will make the query result wrong" is $\frac{1}{k+1}$, so

$$f_2(s) = \frac{x_1 + x_2}{(x_1 + x_2 + x)(k + 1)} \tag{20}$$

We can see that $x_1 + x_2$ is much smaller than $x$ when $s$ is not too small, so $f_2(s)$ is approximately linear to $x_1$ and $x_2$, so

$$E[f_2(s)] \approx \frac{E[x_1] + E[x_2]}{(E[x_1] + E[x_2] + x)(k + 1)} \tag{21}$$

Then we consider $f_1(s)$. the valid hash mapping of streams is at most $k(x + x_1 + x_2)$. Similar to normal BF, we can get

$$f_1(s) \approx (1 - e^{-\frac{k(x+x_1+x_2)}{n}})^k \tag{22}$$

Finally, plug $x$ and $n = \frac{M}{s+t}$, and we get:

$$\begin{aligned} F(s) &= f_1(s) + E[f_2(s)] \\ &\approx (1 - e^{-\frac{(s+t)k(n_0(2^s-2)+\lambda_1\mathcal{T})}{M\lambda_1(2^s-2)}})^k + \frac{\lambda_1\mathcal{T}}{(\lambda_1\mathcal{T} + n_0(2^s - 2))(k + 1)} \end{aligned} \tag{23}$$

Plug the concrete value of $\lambda_1, M, k, t, \mathcal{T}, n_0$ in experimental conditions, and we get that the optimal s generally lies in [8, 64], and it increases as $M$ increases and $\mathcal{T}$ decreases.

## 5.4 Item Batch Size

Item batch size denotes how many members an active item batch has. We use CM+clock to measure it. Assume that the cell number of a CM+clock array is n, the number of arrays is $k$, the size of time window is $\mathcal{T}$, the number of bits per cell is $s$, and the number of bits per counter is $b$ (in our experiment, $b = 16$). Our goal is to minimize the relative error given that the memory $M = kn(s + t)$ is fixed. Since the real value of size is fixed, minimizing the relative error is equivalent to minimizing the absolute error $f(s)$. Ignoring the interruption of outdated elements, there are two types of error: First, before the stream of an element $e$ starts, its counters may not be 0. Second, after the stream of $e$ starts, its counters may be in collision with other streams. Assume that when the stream of $e$ starts, the value of $i - th$ counter of $e$ is $X_i$, and after the stream of $e$ starts, the second-type error of $i - th$counter of $e$ is $Y_i$. Then

$$f(s) = min(X_1 + Y_1, X_2 + Y_2, ..., X_k + Y_k) \tag{24}$$

Assume that when the generation and extinction of streams come into a balance, the number of active streams is $x$, then the average number of streams that disappears in one unit of time is

$\lambda_1 x$ due to the property of exponential distribution. So $\lambda_1 x = n_0$, that is

$$x = \frac{n_0}{\lambda_1} \tag{25}$$

In one counter array, these streams have $x$ hash mappings, so the probability of "a certain cell of counter array is not 0" is $1 - (1 - 1/n)^x \approx 1 - e^{-\frac{x}{n}}$. We can assume that $\frac{x}{n} = \frac{n_0}{n\lambda_1}$ is far less than 1 so that it is almost impossible that one cell is simultaneously used by two streams before the stream of a certain element e starts. Therefore, in the condition that this cell is non-zero its value subject to an exponential distribution with parameter $\lambda_2$. So, for any $i = 1, 2, ...k$

$$Pr(X_i > m) = (1 - e^{-\frac{x}{n}})e^{-\lambda_2 m}, \forall m > 0 \tag{26}$$

and the expectation of $X_i$ is

$$E[X_i] = (1 - e^{-\frac{x}{n}})\frac{1}{\lambda_2} \tag{27}$$

Since $\frac{x}{n} = \frac{n_0}{n\lambda_1}$ is far less than 1, we have:

$$E[X_i] \approx \frac{n_0}{n\lambda_1\lambda_2} \tag{28}$$

Then we consider $Y_i$. We can assume that when measuring, the time span of the stream of $e$, $t(e)$, subject to an exponential distribution with parameter$\lambda_1$. Since $t(e)$ elements arrive during $t(e)$ time, we can get:

$$E[Y_i] = \frac{E[t(e)]}{n} = \frac{1}{n\lambda_1} \tag{29}$$

So

$$E[X_i + Y_i] \approx \frac{n_0 + \lambda_2}{n\lambda_1\lambda_2} \tag{30}$$

Apparently, $X_i + Y_i$is non-negative, so according to Markov inequality :

$$P(f(s) > c\frac{n_0 + \lambda_2}{n\lambda_1\lambda_2}) = P(X_i + Y_i > cE[X_i + Y_i])^k \leq c^{-k}, \forall c > 1 \tag{31}$$

Plug $n = \frac{M}{k(s+b)}$:

$$P(f(s) > c\frac{k(s+b)(n_0 + \lambda_2)}{M\lambda_1\lambda_2}) \leq c^{-k}, \forall c > 1 \tag{32}$$

Note that we have not take the interruption of outdated elements into consideration. Similar to bf+clock with timestamp, we know that the probability of "there is an interruption of outdated element, and it makes the query result wrong" is $\frac{\lambda_1\mathcal{T}}{(\lambda_1\mathcal{T} + n_0(2^s-2))(k+1)}$. So, after taking the interruption of outdated elements into consideration, we have:

$$\begin{aligned} &P(f(s) > c\frac{k(s+b)(n_0 + \lambda_2)}{M\lambda_1\lambda_2}) \\ &\leq c^{-k} + \frac{\lambda_1\mathcal{T}}{(\lambda_1\mathcal{T} + n_0(2^s - 2))(k+1)}, \forall c > 1 \end{aligned} \tag{33}$$

Plug the concrete parameters, and we get that when $s = 2$, $f(s)$ is lower than the case that equals to other values (4, 8, 16, 32) with high probability. However, when $s = 2$, the probability of "there is an interruption of outdated element and it makes the query result wrong" is also larger than other cases, which may bring very big errors. So in practice, the relative error of $s = 2$ and $s = 4$ is almost the same ($s = 4$ is slightly better when memory is large, and $s = 2$ is slightly better when memory is small).

# 6 EXPERIMENTAL RESULTS

In this section, we provide experimental results of our Clock-sketch so as to give practical proof for mathematical analysis in Section 5. All abbreviations of Clock-sketch applications and the state-of-the-art are used in the evaluation, and their full name are shown in Table 2.

**Table 2: Abbreviations in experiments**

| Abbreviation | Full name |
|---|---|
| BF+clock | Clock-sketch for item batch activeness |
| BM+clock | Clock-sketch for item batch cardinality |
| BF-ts+clock | Clock-sketch for item batch time span |
| CM+clock | Clock-sketch for item batch size |
| TBF | Timing Bloom Filter |
| TOBF | Time-Out Bloom Filter |
| TSV | Timestamp Vector algorithm |
| SWAMP | Sliding Window Approximate Measurement Protocol |
| CVS | Counter Vector Sketch |

## 6.1 Experimental Setup

**Implementation:** We implement Clock-sketch and all other algorithms in C++. The hash functions are implemented using the 32-bit Bob Hash (obtained from the open-source website [1]) with different initial seeds.

**Datasets:** We use three datasets and carry out experiments on the above mentioned algorithms. These datasets are all data streams that feature in item batch pattern.

**1) CAIDA** is a public traffic dataset released by CAIDA [2]. Each trace collected from the dataset contains approximately 30M items and 600K distinct items (srcIP). CAIDA features in item batch pattern of flow transmission.

**2) Criteo Dataset** contains feature values and conversion feedback for clicked display ads sampled over a two-month period.[3]. Every ad is associated with a timestamp and 9 categorical terms hashed for anonymity, for a total of 150K unique hashed categorical terms. Criteo features in item batch patterns of user behavior.

**3) Network Dataset** contains users' posting history on the stack exchange website [4]. Each item has three values u,v,t, which means user u answered user v's question at time $t$. We use $u$ as ID. Network also features in item batch pattern of user behavior.

We carry out count-based experiment on CAIDA, Criteo and Network datasets to prove the mathematical analysis in Section 5. For generality, we carry out time-based experiment on CAIDA which demonstrates similar experimental results. Due to space limitations, we display graphs of four experiments on item batch membership measurement, and give CAIDA (count-based) as a representative for the other three tasks.

**Computation Platform:** We conduct the experiments on a machine with two 6-core processors (12 threads, Intel Core i7 8700K CPU @4.8 GHz) and 64 GB DRAM memory @3600MHz. Each processor has three levels of cache memory: one 32KB L1 data caches and one 32KB L1 instruction cache for each core, one 256KB L2 cache for each core, and one 15MB L3 cache shared by all cores.

**Metrics:**

**1)False Positive Rate (FPR):** $\frac{n}{m}$, where $m$ denotes the number of queried inactive item batches, and $n$ denotes the number of queries that returns positive. We use FPR to evaluate the accuracy of BF+clock. Because we use inactive item batches to perform query, there will only be false positives but no true positives.

**2)Relative Error (RE):** $\frac{|f-\widehat{f}|}{f}$, where $f$ denotes the true value of the measurement results and $\widehat{f}$ denotes the estimated measurement result of $f$. We use RE to evaluate the accuracy of BM+clock and BF-ts+clock. For BM+clock, $f$ is the number of distinct and active item batches. For BF-ts+clock, $f$ is the number of item batches whose time span is accurately measured. RE is a suitable metric for BF-ts+clock because the algorithm either gives an accurate answer or a high valuation.

**3) Average Relative Error (ARE):** $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}|f_i-\hat{f}_i|/f_i$, where $f_i$ is the real frequency of item $e_i$, $\widehat{f_i}$ is its estimated frequency, and $\Psi$ is the query set. We use ARE to evaluate the accuracy of CM+clock by querying each active item batch once.

**4) Throughput:** Million operations (insertions or queries) per second (Mops). In comparing Clock-sketch with the state-of-the-art, we only test time consumed to insert into each sketch cell because the clock cell traversal can be performed by another thread in practice. Further, we study the relation between clock cell's size and throughput. In this case, we include the clock traversal time to display the effect. Experiments are repeated 10 times and the average throughput is reported.

**Multithread throughput:** Using two threads to perform the insertion and cleaning process will cause overhead in synchronization, which lowers throughput. To raise throughput, we use two methods: 1) Cancelling the synchronization between the insertion thread and the cleaning thread: Cancelling data synchronization between the insertion thread and the cleaning thread only affects a few cells, which are very likely to be cleaned in the following time. Therefore, cancelling synchronization will barely affect accuracy. 2) Using SIMD acceleration: Because the cleaning thread performs identical operation on an array of cells, this process can be accelerated using SIMD.

## 6.2 Item Batch Activeness

**Optimal Clock Cell Size (Figure 5).** We show the performance of BF+clock with different clock size $s$ under the same memory and window size constraint. We pick the optimal number of hash functions $k$ according to the given clock size $s$. Experiment results show that picking $s = 2$ always leads to the lowest FPR, which confirmed the results in Section 5.1.

**Accuracy evaluation (Figure 6).** We compare BF+clock with the state-of-the-art, namely TBF, TOBF, SWAMP. For BF+clock, we set $s = 2$ and optimal hash function number according to 5.1. We use the recommended parameter for algorithms for comparison. Respectively, we set 18 bits for each counter and 8 hash functions of TBF [38]. For TOBF [23], we use the 64-bit timestamp. For SWAMP, we use its ISMEMBER estimator [7]. All data points that are not shown in these graphs denote zero FPR according to our estimation. Results show when the window size is $2^{16}$, and the memory is limited from 8 KB to 512 KB, our algorithm is better than the state-of-the-art for the different datasets, especially when memory is small. The FPR is two orders of magnitude lower than the other algorithms when the memory is less than 64 KB. Moreover, the **ideal** curve denotes estimating item batch activeness by artificially eliminating the error window, *i.e.*, only items from $t_{cur} - \mathcal{T}$ to $t_{cur}$ into a Bloom filter and query activeness. Experimental results from all datasets demonstrates that our clock-sketch algorithm, BF+clock, is the best approach to the ideal curve.

**Stability evaluation (Figure 7).** We show that the BF+clock gives a comparable FPR when being queried at different times. This indicates that BF+clock is suitable for enduring operation.

**Evaluation on window size (Figure 8).** We give a FPR trend under different window sizes and memory restrictions. On all four datasets used, BF+clock displays a diminishing FPR when the window size shrinks or memory expands.

**Throughput evaluation (Figure 12).** We use a memory of $8KB$ and a window size of 4096 for Bf+clock. It reaches a throughput at about 20 Mops in both insertion and query, which is already high enough and doesn't need SIMD acceleration. This throughput rate rivals all state-of-the-art.

**Evaluation on cache policies (Figure 8).** We compare BF+clock with a typical cache policy, namely LFU (Least Frequently Used), under different cache sizes. For each cache miss, BF+clock chooses the next vacant cell or inactive cell to store the new item. We choose the window size of BF+clock as twice the size of cache, because we want to let the cache store all the active items in the time window and there are duplicate items in the time window. In real applications, the size of BF+clock is small compared to cache storage and can be neglected. Experimental results show that our BF+clock algorithm generally has a higher hit rate than LFU. When the cache size is limited, BF+clock performs notably better.

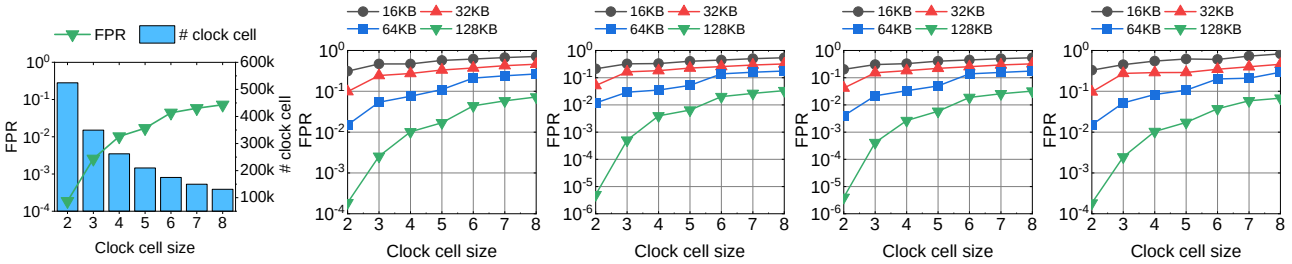
## 6.3 Item Batch Cardinality

**Optimal Clock Cell Size (Figure 9a).** We show the optimal clock size $s$ under different constraints when using CAIDA and count-based time. We give an error bound of $RE(s)$, $\frac{1}{2^s-2}+\sqrt{\frac{8s}{M}ln(\frac{2}{\delta})}$. Given memory $M = 128KB$, window size $W = 16384$ and $\delta = 0.8$, theoretical optimal clock size is $s = 8$ (5.2), which corresponds to the experimental results.

**Accuracy evaluation (Figure 9b).** We compare BM+clock with the state-of-the-art, namely TSV, CVS, SWAMP. We choose optimal $k$ and $s$ for BM+clock and recommended parameter for algorithms for comparison. Specifically, we use 64-bit timestamp for TSV, the maximum value of counter as 10 for CVS and use DISTINCTMLE estimator for SWAMP. Results show that our bitmap algorithm generally performs better than the state-of-the-art when the whindow size is $2^{12}$, and the memory varies from 2 KB to 32 KB. When the memory is less than 32 KB, the RE of our algorithm is more than two orders of magnitude lower than TSV and SWAMP.Our algorithm is also a little better than the CVS in almost all cases.

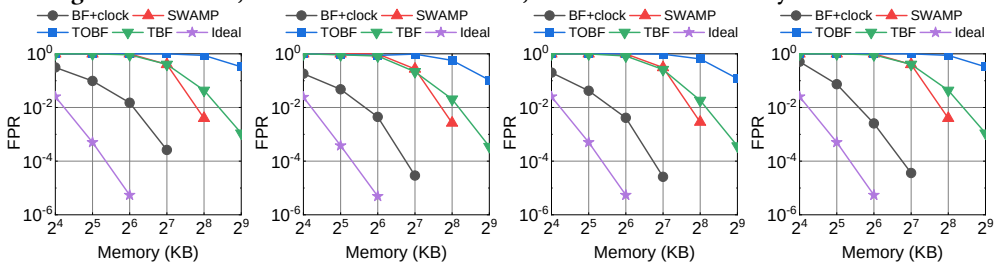
**Stability evaluation (Figure 9c).** We show the BM+clock's RE fluctuation with time. When cardinality changes as time passes, measurement error follows. Generally, RE is beneath 0.08 when memory $M = 4$ KB and window size $W = 2^{12}$.

**Evaluation on window size:** Figure 9d gives a RE trend with different window size. On the Caida dataset, when the memory is



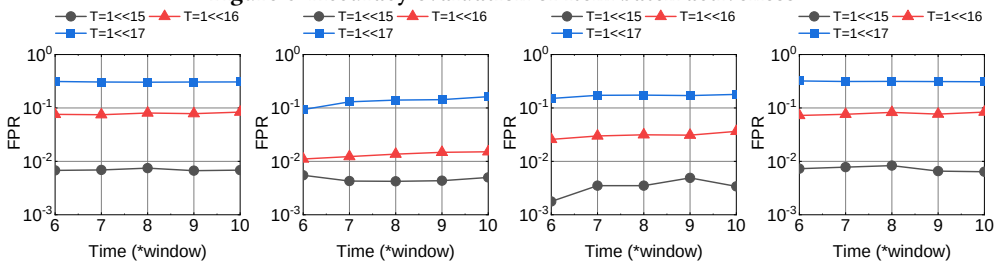

(a) Clock cell number (b) CAIDA(count-based) (c) Criteo(count-based) (d) Network(count-based) (e) CAIDA(time-based)

**Figure 5: Optimal clock cell size for BF+clock under fixed memory. When using BF+clock to measure item batch activeness, the optimal clock size is always 2 under different memory constraints. Under fixed memory, as the clock cell size grows, the number of clock cells decreases, which increases the possibility of hash collisions, further increasing FPR. Figure 5a shows the relationship among clock cell size, the number of clock cells, and FPR when memory is 128KB.**



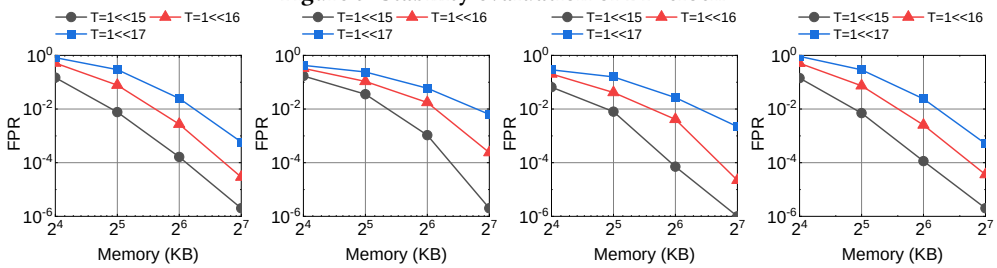

(a) CAIDA(count-based) (b) Criteo(count-based) (c) Network(count-based) (d) CAIDA(time-based)

**Figure 6: Accuracy evaluation of item batch activeness**




(a) CAIDA(count-based) (b) Criteo(count-based) (c) Network(count-based) (d) CAIDA(time-based)

**Figure 7: Stability evaluation of BF+clock**




(a) CAIDA(count-based) (b) Criteo(count-based) (c) Network(count-based) (d) CAIDA(time-based)

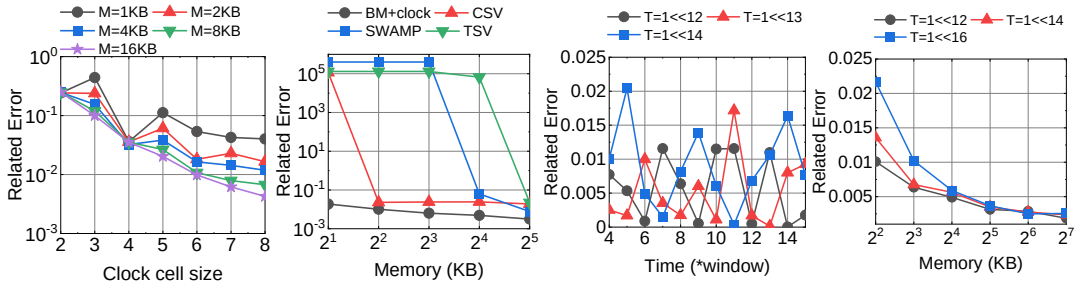
**Figure 8: Window size evaluation of BF+clock**

larger than 4 KB, the case of window size $\mathcal{T} = 2^{14}$ is better than the case of window size $\mathcal{T} = 2^{12}$ or $\mathcal{T} = 2^{10}$.

**Throughput evaluation.** In throughput comparison with the state-of-the-art, we use a memory of $8KB$ and a window size of 8192. The insertion throughput for BM+clock, CVS, TSV and SWAMP are 8.20, 4.97, 13.1, 11.3 Mops respectively.

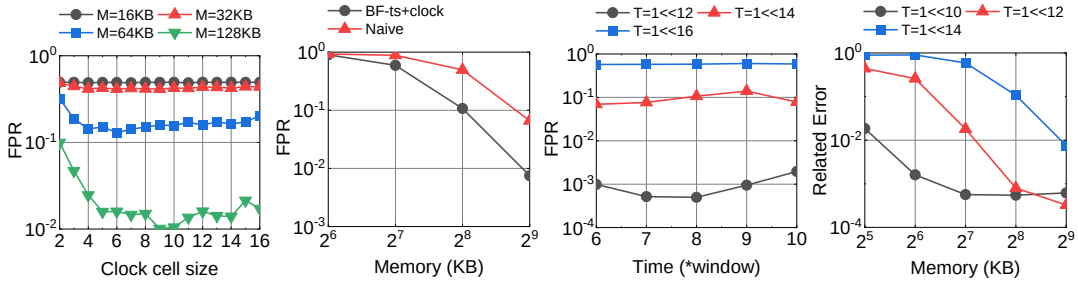
## 6.4 Item Batch Time Span

**Optimal Clock Cell Size (Figure 10a).** We show the optimal clock size is 8 when the window size is 4096 and the memory is $128KB$. The default counter size is 16. This result corresponds to the optimal mathematical value of $F(s)$ in Section 5.3.



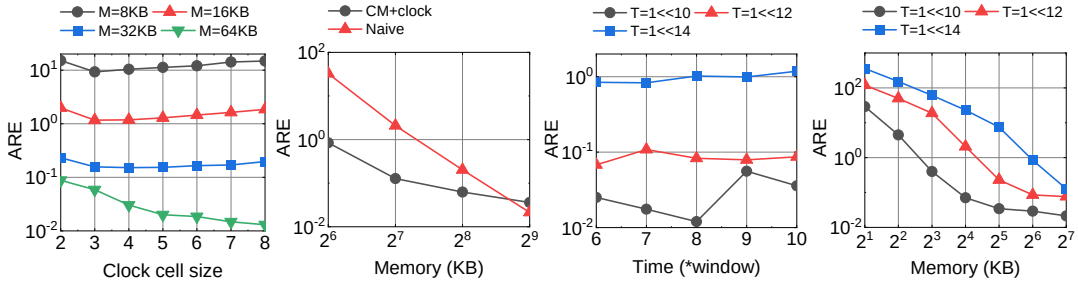

(a) Optimal clock size (b) Accuracy evaluation (c) Stability evaluation (d) Window size evaluation

**Figure 9: Evaluation of item batch cardinality**




(a) Optimal clock size (b) Accuracy evaluation (c) Stability evaluation (d) Window size evaluation

**Figure 10: Evaluation of item batch time span**




(a) Optimal clock size (b) Accuracy evaluation (c) Stability evaluation (d) Window size evaluation

**Figure 11: Evaluation of item batch size**

**Table 3: Evaluation on Throughput**

| Algorithm | Clock cell size | Throughput (single thread) | Throughput (multi-thread) | Throughput (multi-thread & SIMD) | Accuracy (single thread) | Accuracy (multi-thread) |
|---|---|---|---|---|---|---|
| BF+clock | 2 | 17.7227 Mops | - | - | - | - |
| BM+clock | 8 | 1.14506 Mops | 1.12482 Mops | 8.2028 Mops | RE=0.001853 | RE=0.00188 |
| CM+clock | 8 | 0.245895 Mops | 0.250286 Mops | 2.3119 Mops | ARE=0.000327 | ARE=0.004392 |
| BF-ts+clock | 8 | 1.10144 Mops | 1.08607 Mops | 6.52566 Mops | FPR=0.000257 | FPR=0.000253 |

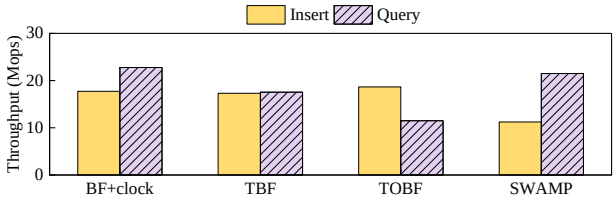



**Figure 12: Throughput comparison of item batch activeness**

**Accuracy evaluation (Figure 10b).** As no algorithm is specially designed for item batch time-span measurement, here we give a

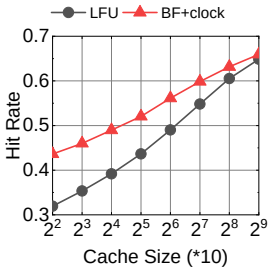



**Figure 13: Hit rate comparison on cache size**

**naive solution** without using clock framework. We substitute each clock in BF-ts+clock with a 64-bit timestamp $t_l$ which records the last time that this counter is visited. The sketch cells store $t_{sr}$, the start time of item batches. For insertion of item $a$, it picks $k$ hash positions and compares $t_{cur}$ and $t_l$. If the gap between $t_{cur}$ and $t_l$ exceeds $\mathcal{T}$, we change both $t_{sr}$ and $t_l$ to $t_{cur}$. Otherwise, we only change $t_l$ to $t_{cur}$. For query, we pick the $k$ hash positions and choose the earliest $t_l$, denoted as $t_f$. For any active item batch, the gap between $t_f$ and $t_{cur}$ must be smaller than $\mathcal{T}$. The algorithm returns the latest $t_{sr}$ timestamp whose corresponding $t_l = t_f$. Apparently, this naive solution either gives a correct answer or gives an overestimation of the time span of $B_a$, which is the same as BF-ts+clock. Figure 10b shows the experimental result of BF-ts+clock versus the naive solution. Results show that the optimal clock size is 8 when using 128KB memory. Also, when memory increases, the optimal clock size increases, which proves the effect of clock framework.

**Stability evaluation (Figure 10c).** We demonstrate the stability of BF+clock over time.

**Evaluation on window size (Figure 10d).** We give a similar trend of FPR as the former tasks.

**Throughput evaluation.** We use a memory of $128KB$ and a window size of 4096, and set the clock cell size to 8. The insertion throughput and query throughput of CM+clock are 1.10144 Mops and 21.501, respectively.

## 6.5 Item Batch Size

**Optimal Clock Cell Size (Figure 11a).** We show that when the window size is $2^{14}$, the optimal clock size is 3 or 4 when the memory is $16KB$ to $32KB$ and the optimal clock size is 8 when the memory increase to $64KB$. Both results correspond to the optimal value given in 5.4.

**Accuracy evaluation (Figure 11b).** As no algorithm is specially designed for item batch size measurement, here we give a **naive solution** without using clock framework. Similar to the naive solution proposed in Section 6.4, we substitute each clock in CM+clock with a 64-bit timestamp $t_l$ which records the last time that this counter is visited. For insertion of item $a$, it picks $k$ hashed locations and compare $t_{cur}$ and $t_l$. If the gap between $t_{cur}$ and $t_l$ exceeds $\mathcal{T}$, we reset the counter to 0 and set $t_l$ to $t_{cur}$. Otherwise, we only increment the counter by 1. For query, we pick the smallest counter among all $k$ hashed locations. Figure 11b shows the experimental result of CM+clock versus the naive solution. Results show when the memory is less than $256KB$, the CM (ours) is always better than the CM (naive), which proves the effect of clock framework.

**Stability evaluation (Figure 11c).** We show the CM sketch gives a comparable ARE when being queried at different times. This indicates that CM sketch is suitable for enduring operation.

**Evaluation on window size (Figure 11d).** We give a trend of ARE in different window sizes. It shows that when clock $s = 2$, the ARE increases as the window size increases from $W = 2^{10}$ to $W = 2^{14}$. And as the memory used increases from 32 KB to 512 KB, the difference decreases.

**Throughput evaluation.** We use a memory of $512KB$ and a window size of 16384, and set the clock cell size to 8. The insertion throughput and query throughput of CM+clock are 0.245895 Mops and 14.3706 respectively.

# 7 CONCLUSION AND FUTURE WORK

We are the first to measure item batch, a useful pattern in common data streams. We define an item batch as a consecutive sequence of identical items that are close in time. We introduce CLOCK into item batch measurement and propose Clock-sketch in this paper. The key idea is preserving information of all active item batches in the data structure and clean information of inactive item batches as much as possible. We perform extensive experiments on three real world datasets to test measurement results on data stream with batch pattern. Results show that our Clock-sketch can achieve high accuracy and high speed with small memory restriction. All related source codes are anomalously released at Github [5].

In this paper, we discuss item batch composed of identical items, *i.e.*, items with the same ID in a data stream. Other real world applications may require different definitions of item batch: 1) Item batch may be composed of similar items rather than identical items. For example, when processing a stream of purchase records, beef and steak are similar items while soap and milk are not. 2) The threshold $\mathcal{T}$ for two different item batches $B_a$ and $B_b$ may differ and an algorithm should learn the proper thresholds for different item batches. Besides exploring various definitions of item batch in other applications, item batch measurement is also useful in distributed systems. We only perform single node item batch measurement using Clock-sketch. Combining Flink framework can help save synchronization cost in distributed measurement. These are promising directions that will further lead the study of item batch.

# REFERENCES


[1] 2008. Bob Jenkins' hash function web page, paper published in Dr Dobb's journal. http://burtleburtle.net/bob/hash/doobs.html.

[2] 2020. The CAIDA Anonymized Internet Traces. http://www.caida.org/data/overview/.

[3] 2020. The Criteo dataset Internet Traces. https://cmt3.research.microsoft.com/SIGMOD2021/Submission/Details/25.

[4] 2020. The Network dataset Internet Traces. http://snap.stanford.edu/data/.

[5] 2020. The source codes of our and other related algorithms. https://github.com/Clock-sketch/clock-sketch2020.

[6] 2020. The technical report of our paper, including all pseudocode. https://github.com/Clock-sketch/clock-sketch2020/blob/master/Technical_report.pdf.

[7] Eran Assaf, Ran Ben Basat, Gil Einziger, and Roy Friedman. 2018. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2204–2212.

[8] Bryan Ball, Mark Flood, Hosagrahar Visvesvaraya Jagadish, Joe Langsam, Louiqa Raschid, and Peratham Wiriyathammabhum. 2014. A flexible and extensible contract aggregation framework (caf) for financial data stream analytics. In *Proceedings of the International Workshop on Data Science for Macro-Modeling*. 1–6.

[9] Sorav Bansal and Dharmendra S Modha. 2004. CAR: Clock with Adaptive Replacement.. In *FAST*, Vol. 4. 187–200.

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

[11] Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin. 2014. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1441–1451. https://doi.org/10.14778/2733004.2733016

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

[13] Fernando J Corbato. 1968. *A paging experiment with the multics system.* Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC.

[14] Fernando J Corbato. 1968. *A paging experiment with the multics system.* Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC.

[15] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[16] Michael K Daly. 2009. Advanced persistent threat. *Usenix, Nov* 4, 4 (2009), 2013–2016.

[17] Gil Einziger and Roy Friedman. 2015. Counting with TinyTable: Every bit counts!. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 77–78.

[18] Lajos Gergely Gyurkó, Terry Lyons, Mark Kontkowski, and Jonathan Field. 2013. Extracting information from the signature of a financial data stream. *arXiv preprint arXiv:1307.7244* (2013).

[19] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce Lindsay, Hamid Pirahesh, Michael J. Carey, and Eugene Shekita. 1990. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge & Data Engineering* 1 (1990), 143–160.

[20] Thoufique Haq, Jinjian Zhai, and Vinay K Pidathala. 2017. Advanced persistent threat (APT) detection center. US Patent 9,628,507.

[21] Hyang-Ah Kim and David R O'Hallaron. 2003. Counting network flows in real time. In *GLOBECOM'03. IEEE Global Telecommunications Conference (IEEE Cat. No. 03CH37489)*, Vol. 7. IEEE, 3888–3893.

[22] Jon Kleinberg. 2003. Bursty and hierarchical structure in streams. *Data Mining and Knowledge Discovery* 7, 4 (2003), 373–397.

[23] Shijin Kong, Tao He, Xiaoxin Shao, Changqing An, and Xing Li. 2006. Time-out bloom filter: A new sampling method for recording more flows. In *International Conference on Information Networking*. Springer, 590–599.

[24] Marshall Kirk McKusick, Keith Bostic, Michael J Karels, and John S Quarterman. 1996. *The design and implementation of the 4.4 BSD operating system.* Vol. 2. Addison-Wesley Reading, MA.

[25] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 775–787. https://doi.org/10.1145/3035918.3035963

[26] Yanqing Peng, Jinwei Guo, Feifei Li, Weining Qian, and Aoying Zhou. 2018. Persistent bloom filter: Membership testing for the entire history. In *Proceedings of the 2018 International Conference on Management of Data*. 1037–1052.

[27] Pratanu Roy, Arijit Khan, and Gustavo Alonso. 2016. Augmented sketch: Faster and more accurate stream processing. In *Proceedings of the 2016 International Conference on Management of Data*. 1449–1463.

[28] Jingsong Shan, Jianxin Luo, Guiqiang Ni, Zhaofeng Wu, and Weiwei Duan. 2016. CVS: fast cardinality estimation for large-scale data streams over sliding windows. *Neurocomputing* 194 (2016), 107–116.

[29] Anshumali Shrivastava, Arnd Christian Konig, and Mikhail Bilenko. 2016. Time adaptive sketches (ada-sketches) for summarizing data streams. In *Proceedings of the 2016 International Conference on Management of Data*. 1417–1432.

[30] Kai Sheng Tai, Vatsal Sharan, Peter Bailis, and Gregory Valiant. 2018. Sketching linear classifiers over data streams. In *Proceedings of the 2018 International Conference on Management of Data*. 757–772.

[31] Nan Tang, Qing Chen, and Prasenjit Mitra. 2016. Graph stream summarization: From big bang to big crunch. In *Proceedings of the 2016 International Conference on Management of Data*. 1481–1496.

[32] Daniel Ting. 2018. Count-Min: Optimal Estimation and Tight Error Bounds using Empirical Error Distributions. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2319–2328.

[33] Tingting Chen, Yi Wang, Binxing Fang, and Jun Zheng. 2006. Detecting Lasting and Abrupt Bursts in Data Streams Using Two-Layered Wavelet Tree. In *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)*. 30–30.

[34] Nikos Virvilis and Dimitris Gritzalis. 2013. The big four-what we did wrong in advanced persistent threat detection?. In *2013 international conference on availability, reliability and security*. IEEE, 248–254.

[35] Zhewei Wei, Ge Luo, Ke Yi, Xiaoyong Du, and Ji-Rong Wen. 2015. Persistent data sketching. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*. 795–810.

[36] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. 1990. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)* 15, 2 (1990), 208–229.

[37] Wei Xie, Feida Zhu, Jing Jiang, Ee-Peng Lim, and Ke Wang. 2016. Topicsketch: Real-time bursty topic detection from twitter. *IEEE Transactions on Knowledge and Data Engineering* 28, 8 (2016), 2216–2229.

[38] Linfeng Zhang and Yong Guan. 2008. Detecting click fraud in pay-per-click streams of online advertising networks. In *2008 The 28th International Conference on Distributed Computing Systems*. IEEE, 77–84.

[39] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. 2018. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data*. 741–756.

[40] Yunyue Zhu and Dennis Shasha. 2003. Efficient elastic burst detection in data streams. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. 336–345.