Measuring Item Freshness in Data Streams

Zirui Liu*
Peking University
zirui.liu@pku.edu.cn

Zihan Jiang*
Peking University
jumbo0715@stu.pku.edu.cn

An Zhang*
Peking University
magicine_maker_21@stu.pku.edu.cn

Zhouran Shi
Hong Kong University of Science and Technology
1900010625@pku.edu.cn

Yuxuan Tian*
Peking University
tianyuxuan@stu.pku.edu.cn

Tong Yang*†
Peking University
yangtong@pku.edu.cn

Abstract

This paper studies an unexplored attribute in data streams – *item freshness*. The freshness of an item refers to the time interval between its last arrival and the present moment. The information of item freshness is useful in various scenarios like cache, online advertising, computer network, *etc.* Currently, there is no algorithm tailored for estimating item freshness. We propose a theoretically guaranteed sketch algorithm called RingSketch, which integrates time-agnostic sketch algorithm with time-aware CLOCK algorithm for real-time freshness measurement. With the key idea of tracing the trajectory of the clock pointer, the estimation process of RingSketch is akin to observing the length of the growth rings in a tree trunk. We theoretically derive the average error of RingSketch and validate it with extensive experiments. The results show that RingSketch simultaneously achieves high accuracy ($< 10^{-3}$ average relative error) and fast update speed ($> 11.4\ M/s$), outperforming the baseline solutions by at least 13.3× and 1.5× respectively. All codes are open-sourced at GitHub [1].

CCS Concepts

• **Theory of computation → Sketching and sampling**.

Keywords

data streams, sketches, item freshness, clock algorithm

ACM Reference Format:
Zirui Liu, Zihan Jiang, An Zhang, Zhouran Shi, Yuxuan Tian, and Tong Yang. 2025. Measuring Item Freshness in Data Streams. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '25), August 3–7, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3711896.3737044

*National Key Laboratory for Multimedia Information Processing, School of Computer Science, Peking University, Beijing, China
†Corresponding author: Tong Yang (yangtong@pku.edu.cn)

KDD '25, Toronto, ON, Canada

ACM ISBN 979-8-4007-1454-2/2025/08
https://doi.org/10.1145/3711896.3737044

1 Introduction

Data stream is an important model in many scenarios [2]. Besides classic tasks like measuring item frequency [3] and finding frequent items [4], recent years have witnessed a surge of interest in measuring new attributes in data streams like cardinality [5], quantile [6], entropy [7], This paper studies an unexplored attribute in data streams: item freshness. We define the freshness of an item as the interval between its last arrival time and the present time. This paper aims at real-time estimating the freshness (or equivalently, the last arrival time) of any item in a data stream.

The information of item freshness is important in many scenarios. 1) In cache design, many studies have suggested that Least Recently Used (LRU) cache performs best in most scenarios [8], while devising an accurate and efficient LRU cache is notably challenging [9]. The information of item freshness can help to determine which items to evict when the cache is full, so as to guide the design of an efficient LRU cache. 2) In online advertising, item freshness helps in understanding user behavior, allowing for the timely delivery of relevant advertisements based on the most recent user interactions, such as clicks or purchases. For example, some users tend to frequently visit certain kinds of products, leading their corresponding freshness to stay small. Therefore, the ad system can recommend similar products to these users. Based on item freshness, we can also further investigate user's periodic browsing or buying behavior [10], so as to attain better-targeted recommendations and higher engagement. 3) In computer networks, item freshness can improve network measurement and management. Given that network traffic consists of numerous packets from many flows, item freshness can help identify active flows and estimate the number of current active flows, whose increase may indicate potential attacks. Additionally, item freshness can also aid in recognizing flowlets [11], which is defined as a burst of packets from the same flow followed by an idle interval, thereby further supporting network management and load balancing [12, 13].

To our best knowledge, there is no existing work tailored for measuring item freshness. A straightforward solution is to use a hash table to store all item IDs and their last arrival time. This solution is memory inefficient considering the large volume of data streams. On the other hand, we can adapt the algorithms designed for other problems [14–17] to freshness measurement. The most related problem is item batch detection [15], which is equivalent to determining whether the freshness of each item exceeds a fixed threshold. As item batch detection is essentially an existence detection problem, its state-of-the-art (SOTA) solutions only need coarse-grained time

information, while estimating the exact freshness for each item requires fine-grained time information. Therefore, existing solutions fall short in accuracy when directly applied to freshness estimation, which will be further discussed in § 2.2. This paper aims at devising a memory-efficient data structure that does not store item IDs while enabling accurate freshness estimation.

In this paper, we propose a time-aware sketch algorithm, called RingSketch, to accurately measure item freshness. The key idea of RingSketch is the integration of time-agnostic sketch algorithms with time-aware CLOCK [18] algorithm, and we further propose the key technique called *pointer tracing* to enhance the estimation granularity of CLOCK algorithm. Considering the large volume of data streams, it is inefficient to accurately record the IDs and the last arrival time for all items. Therefore, employing probabilistic data structures (sketches) to approximately summarize the freshness information is desirable. However, existing sketches mainly focus on measuring item frequency and they are unable to capture temporal information. To enable time-agnostic sketches to be aware of time, we combine sketch algorithms with CLOCK [18] algorithms. As shown in Figure 1, the basic data structure of RingSketch is a circular counter array. Like existing sketches [3, 19, 20], each incoming item is hashed into multiple counters within this array, and the incoming item sets its hashed counters to the maximum value. Here, larger counter value indicates that the item hashed into the counter is fresher. A CLOCK pointer periodically sweeps across the circular array and decreases each of its passing counter. To estimate the freshness of an item, RingSketch analyzes the values of its hashed counters to estimate the pointer position at the item's last arrival. Then RingSketch traces the trajectory that the pointer has traveled since the item's last arrival, thereby calculating the item's freshness based on the length of the trajectory and the pointer's sweeping speed.

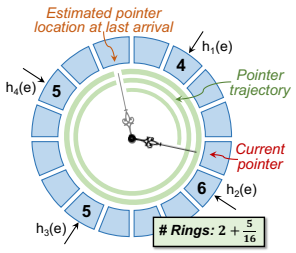


Figure 1: An estimation example of RingSketch.

As shown in Figure 1, the process of estimating item freshness with RingSketch is akin to observing the length of growth rings in a tree trunk. We calculate the freshness based on the length of the growth rings and the growth speed (pointer sweeping speed). We theoretically analyze the estimation error of RingSketch and conduct extensive experiments. The results show that RingSketch achieves $< 10^{-3}$ average relative error and $> 11.4\ M/s$ update throughput, outperforming the baseline solutions by at least 13.3× and 1.5× respectively. All codes are open-sourced [1].

This paper makes the following key contributions.

- We are the first to formulate the attribute of item freshness.
- We propose RingSketch to accurately and efficiently measure item freshness with theoretical guarantees.
- We extensively evaluate RingSketch, showing its accuracy and speed outperform baselines by 13.3× and 1.5× respectively.

Table 1: Symbols frequently used in this paper.

Symbols	Meaning
n	Number of distinct items in data stream
e_i	An item in data stream
F_i	Freshness of item e_i
d	Number of parts (hash functions) in RingSketch
m	Number of counters in each part of RingSketch
s	Each counter in RingSketch consists of s bits
$\mathcal{A}_i$	The i^{th} part (counter array) in RingSketch
$h_i(\cdot)$	The i^{th} hash function mapping items into $\mathcal{A}_i$
$\mathcal{V}$	Scanning speed of pointer (counters per time unit)
$\mathcal{T}_0$	Time for pointer to complete one cycle ($\mathcal{T}_0 = dm/\mathcal{V}$)
$\mathcal{T}$	Time for pointer to complete $2^s - 1$ cycles ($\mathcal{T} = \frac{dm\cdot(2^s-1)}{\mathcal{V}}$)

2 Background and Related Work

2.1 Problem Statement

Data stream: A data stream σ is defined as an unbounded sequence $\{e_i\}_{i=1,2,\ldots}$ of items drawn from the universe $[n] := \{1, 2, \ldots, n\}$. Each item e_i in σ is associated with a timestamp t_i indicating its arrival time.

Item freshness: Given an item $e_i \in [n]$ with the last arrival time t_i, we define its freshness as $F_i = |t_{now} - t_i|$, which is the time gap between its last appearance and current moment. We aim at estimating the freshness for any item, which is equivalent to estimating the last arrival time.

2.2 Related Work

Although there is no specialized work for freshness estimation, we can measure item freshness by adapting some algorithms for sliding window measurement [14] and batch detection [15–17]. Below we introduce four algorithms and discuss how to adapt them to measure item freshness.

1) SWAMP [14]: Sliding Window Approximate Measurement Protocol (SWAMP) is an algorithm designed for measuring various attributes for recent items. It uses a cyclic array to record the fingerprints of the items in current sliding window, and uses a hash table called TinyTable to record the frequencies of these fingerprints. To measure item freshness, we can record the last arrival time of these fingerprints in the TinyTable. However, this solution is memory inefficient because it stores metadata of fingerprints and their frequencies.

2) TOBF [17]: Time-Out Bloom Filter (TOBF) is a variant of the classic Bloom filter [21], which was initially proposed to improve the sampling performance of small flows in network traffic. A TOBF consists of an array of timestamps. For each incoming item, TOBF hashes it into d timestamps in the array, and sets the d hashed timestamps to the current time t_{now}. To estimate the freshness of an item, TOBF finds the oldest timestamp t_{old} among its d hashed timestamps, and returns $\hat{F} = |t_{now} - t_{old}|$. TOBF is also memory inefficient because it stores the raw timestamps, which is typically of 64-bit. By contrast, our RingSketch will use small counter (4-bit) to approximately record time information.

3) HyperBF [16]: Hyper Bloom Filter (HyperBF) is another variant of the Bloom filter algorithm, which was proposed to detect item batches in data streams. A HyperBF cyclically divides the timeline

into $2^s - 1$ epochs of length δ, and it maintains an array of s-bit cells to record temporal information of items. For each incoming item, HyperBF first updates its d hashed cells to the current epoch, and then incidentally cleans some adjacent outdated cells. Similar to TOBF, to estimate the freshness of an item, HyperBF checks the d hashed cells and finds the oldest one. Suppose there are x epochs between the oldest cell and current moment, HyperBF estimates the freshness of the item as $x \times \delta$. Note that the estimation granularity of HyperBF can only be the multiple of epoch length δ. Thus, when directly applied to freshness estimation, its accuracy is unsatisfactory.

4) ClockSketch [15]: ClockSketch is a recent algorithm proposed to detect item batches, which is very similar to our RingSketch but can only capture coarse-grained time information. It also consists of an array of s-bits cells. Each incoming item sets its d hashed cells to the maximum value $2^s - 1$. ClockSketch uses a separate thread to cyclically sweep the cell array at a constant speed and decrease each passing non-zero cell by one. To query the freshness of an item, ClockSketch also looks for the oldest cell among the d hashed cells. Let x be the value of the oldest hashed cell. ClockSketch reports item freshness as $(2^s - 1 - x) \times \delta$, where δ is the time taken by the pointer to complete one rotation. Similar to HyperBF, the estimation granularity of ClockSketch can also only be the multiple of epoch length δ. RingSketch aims at breaking the granularity limitation of ClockSketch and HyperBF.

As a class of spatiotemporal efficient probabilistic data structures, sketches are widely used in a variety of data stream mining tasks [3, 6, 7, 19, 20, 22–29]. Typical sketches work by compactly recording the approximate statistics of data streams in a summary. However, most existing sketches only focus on recording item frequency, and they are unable to capture the temporal information of items. This paper enables time-agnostic sketches to be aware of time, and extends their application to a new task of freshness estimation.

3 The RingSketch Algorithm

Design rationale: For large-scale data streams, it is challenging to accurately track the freshness of all items within limited memory. Like many other sketches, RingSketch also uses sublinear space to approximate the key temporal information of critical items. As the temporal information of fresh items are more important than stale items [9, 15], we prioritizes the accuracy of fresh items by allowing the temporal data of stale items to be gradually overwritten. Specifically, we use a sketch to record the temporal information of each item, and use a pointer to cyclically scan the sketch according to CLOCK algorithm [18]. To query the freshness of an item, we trace the pointer trajectory since the item's last arrival. We calculate the item's freshness based on the length of the pointer trajectory and the pointer scanning speed.

Data structure: As shown in Figure 2, the data structure of RingSketch consists of a cyclic array of s-bit counters. The cyclical array is partitioned into d parts $\mathcal{A}_1, \cdots, \mathcal{A}_d$, each of which has m counters. Each part of the counter array $\mathcal{A}_i$ is associated with one pairwise independent hash function $h_i(\cdot)$ that maps items into one counter in it. A clock pointer cyclically sweeps the counters in a clockwise direction at a velocity of $\mathcal{V}$ (counters per time unit), and decreases each passing counter by 1. Under such configuration, the pointer sweeps a counter from its maximum value $(2^s - 1)$ to 0 in $\mathcal{T} = md \cdot (2^s - 1)/\mathcal{V}$ time units. Therefore, for a stale item that has not appeared for more than $\mathcal{T}$ time units, its temporal information will be cleaned. In practice, we can arbitrarily set $\mathcal{T}$ by adjusting the parameters s and $\mathcal{V}$.

Update: For item e_i, we calculate hash functions to locate d hashed counters $\mathcal{A}_1[h_1(e_i)], \cdots, \mathcal{A}_d[h_d(e_i)]$. We update the d hashed counters to the maximum value $2^s - 1$ to overwrite the temporal information of stale items. The update procedure can be accelerated with multi-threading by creating d threads to update the counters in the d parts, and another thread for the clock pointer. We can also use SIMD instructions to sweep multiple counters simultaneously.

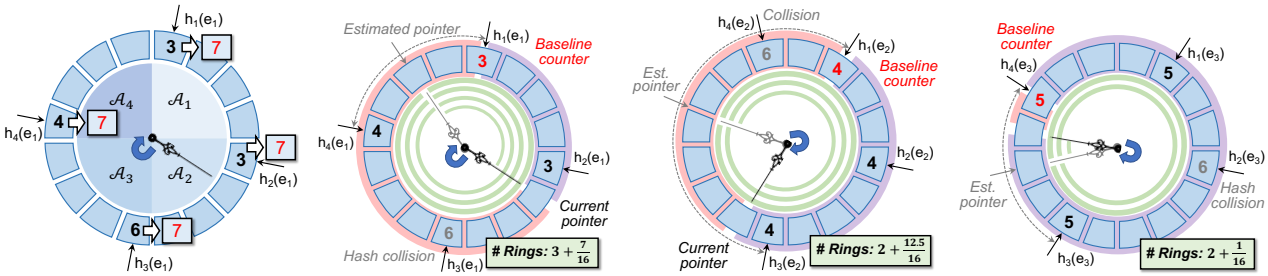
Query: Given an item e_i, we first identify which hashed counters of e_i are the invalid counters that have been overwritten (Step 1). Afterwards, we estimate the pointer location at e_i's last arrival (Step 2). Next, we trace the pointer trajectory since e_i's last arrival based on the estimated pointer position and the values of the hashed counters (Step 3). We finally calculate the freshness of e_i according to the pointer trajectory length and the pointer sweeping speed (Step 4). The overall *design principle* of our query operation is to use the pointer's rotation as an implicit clock, estimating elapsed time by tracking its trajectory since an item's last arrival. By filtering out collision-corrupted counters and tracing the pointer's movement from the remaining valid hashed counters, RingSketch converts spatial rotation into temporal duration without explicitly storing timestamps, achieving high accuracy and minimal memory overhead at the same time.

Below we elaborate on the four steps:

1) Collision Identification. We first identify the counters among the d hashed counters that are overwritten by fresher items. Specifically, we find the counter with the smallest value Min among the d hashed counters, and we call this counter the *baseline counter*. If there are multiple counters having the same smallest value, we select the one that is the farthest from the current pointer (in counter-clockwise direction) to be the *baseline counter*. Subsequently, we inspect all the hashed counters in a counter-clockwise manner. We begin at current pointer position, proceed counter-clockwise to examine each hashed counter until the baseline counter (the counters with purple background in Figure 2(b)-2(d)). If a hashed counter in question is larger than Min, a hash collision is detected and the counter will be marked invalid. Then, we continue to examine the remaining hashed counters in the same direction (the counters with red background in Figure 2(b)-2(d)), from the baseline counter back to the current pointer, in which case any counter larger than Min+1 will be marked as invalid. In the following steps, we will consider these overwritten counters as invalid.

2) Pointer Estimation. We use the remaining valid hashed counters to locate the pointer position at e_i's last arrival. Starting from the *baseline counter*, we find the next valid hashed counter in counter-clockwise direction. We estimate the pointer position at e_i's last arrival to be the midpoint between the baseline counter and this valid counter. Note that when no other valid hashed counter exists, the baseline counter itself serves as the next valid hashed counter. In such case, we estimate the pointer position at the opposite position of the baseline counter (180° apart).

3) Trajectory Tracing. We determine the trajectory that the pointer has traveled since e_i's last arrival. This is done by rotating the



(a) Update example (b) Query example 1 (c) Query example 2 (d) Query example 3

Figure 2: Examples of RingSketch ($s = 3$, $d = 4$, $m = 4$, $\mathcal{T}_0 = 1$ second).

pointer counter-clockwise from its current position, incrementing 1 to each valid hashed counter as the pointer passes them, until the pointer finally stops at the estimated position and the values of all valid hashed counters become $2^s - 1$. Specifically, in counter-clockwise direction, if current pointer is closer to the baseline counter than the estimated pointer (Figure 2(b)-2(c)), we estimate the number of rings to be $\mathcal{R} = 2^s - 2 - Min + \frac{C}{dm}$, where C is the distance between current pointer and the estimated pointer (in the number of counters). Otherwise, if current pointer is closer to the estimated pointer (Figure 2(d)), we estimate the number of rings to be $\mathcal{R} = 2^s - 1 - Min + \frac{C}{dm}$.

4) Freshness Calculation. We finally calculate the estimated freshness of item e_i as $\hat{F}_i = \mathcal{R} \cdot \mathcal{T}_0$, where $\mathcal{T}_0 = \frac{dm}{V}$ is the time for the pointer to complete one cycle.

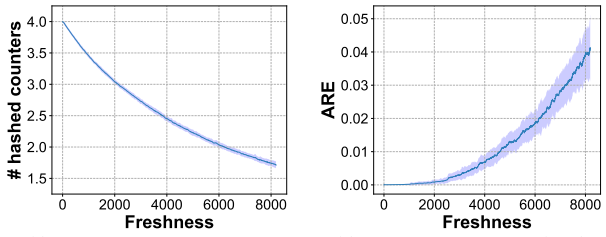
Examples (Figure 2): Below we use four examples to illustrate the update and query operations of RingSketch, where we set $s = 3$, $d = 4$, and $\mathcal{T}_0 = 1$ second.

Update example (Figure 2(a)): For the incoming item e_1, we first calculate hash functions to locate its d hashed counters. Then we update the value of the d hashed counters to 7.

Query example 1 (Figure 2(b)): *1) Collision Identification:* As there are two hashed counters with the smallest value $Min = 3$, we select the counter that is farther from the current pointer in counter-clockwise direction as *baseline counter* ($\mathcal{A}_1[h_1(e_1)]$). Next, starting from the current pointer, we examine each hashed counter in counter-clockwise direction. As $\mathcal{A}_3[h_3(e_1)] > Min + 1$, we mark it as invalid. *2) Pointer Estimation:* We estimate the pointer position at e_1's last arrival to be the midpoint between *baseline counter* $\mathcal{A}_1[h_1(e_1)]$ and its next valid counter $\mathcal{A}_4[h_4(e_1)]$. *3) Trajectory Tracing:* As the current pointer is closer to the *baseline counter* than the estimated pointer in counter-clockwise direction, we calculate the number of rings as $\mathcal{R} = 2^s - 2 - Min + \frac{C}{dm} = 3 + \frac{7}{16}$. *4) Freshness Calculation:* We estimate freshness as $\hat{F}_1 = \mathcal{R} \cdot \mathcal{T}_0 = 3 + \frac{7}{16}$ second.

Query example 2 (Figure 2(c)): *1) Collision Identification:* As there are three hashed counters with the smallest value $Min = 4$, we select the counter that is farther from the current pointer as *baseline counter* ($\mathcal{A}_1[h_1(e_2)]$). As $\mathcal{A}_4[h_4(e_2)]$ is larger than $Min + 1$, we consider it to have a hash collision. *2) Pointer Estimation:* We estimate the pointer position at e_2's last arrival to be the midpoint between $\mathcal{A}_1[h_1(e_2)]$ and $\mathcal{A}_3[h_3(e_2)]$. *3) Trajectory Tracing:* As current pointer is closer to $\mathcal{A}_1[h_1(e_2)]$ than the estimated pointer, we calculate the number of rings as $\mathcal{R} = 2^s - 2 - Min + \frac{C}{dm} = 2 + \frac{12.5}{16}$. *4) Freshness Calculation:* We estimate freshness as $\hat{F}_2 = \mathcal{R} \cdot \mathcal{T}_0 = 2 + \frac{12.5}{16}$ second.

Query example 3 (Figure 2(d)): *1) Collision Identification:* We first select $\mathcal{A}_4[h_4(e_3)]$ as *baseline counter*. As $\mathcal{A}_2[h_2(e_3)]$ is between the current pointer and the *baseline counter* and $\mathcal{A}_2[h_2(e_3)] > Min$, we consider it as invalid. *2) Pointer Estimation:* We estimate the pointer position at e_3's last arrival to be the midpoint between $\mathcal{A}_4[h_4(e_3)]$ and $\mathcal{A}_3[h_3(e_3)]$. *3) Trajectory Tracing:* As current pointer is closer to the estimated pointer than $\mathcal{A}_4[h_4(e_3)]$, we calculate the number of rings as $\mathcal{R} = 2^s - 1 - Min + \frac{C}{dm} = 2 + \frac{1}{16}$. *4) Freshness Calculation:* We estimate freshness as $\hat{F}_3 = \mathcal{R} \cdot \mathcal{T}_0 = 2 + \frac{1}{16}$ second.



(a) Number of hashed counters (b) Average Relative Error (ARE)

Figure 3: Memory allocation and average relative error for items with different freshness.

Discussion: As discussed above, RingSketch aims at sacrificing the accuracy of stale items in favor of fresh items. Specifically, in RingSketch, the hashed counter(s) of a stale item can be overwritten by fresher items. Furthermore, if a stale item has not arrived for $\mathcal{T}$ time units, its freshness information will be cleaned. In this way, RingSketch automatically allocates more memory to fresher items. To better illustrate this behavior, we conduct experiments using CAIDA dataset [30], where we use a 16KB RingSketch with $d = 4$ and $s = 12$ and we set $\mathcal{T} = 8192$. In Figure 3, we plot the average number of hashed counters (±2std) and the average relative error (ARE) (±2std) for the items with different freshness. The results show that for the freshest items, RingSketch allocates $d = 4$ hashed counters to them, achieving nearly 0 ARE. As an item becomes stale (with larger freshness), RingSketch automatically reduces the number of hashed counters allocated to it, resulting in an increase in the ARE. This behavior is consistent with the design rationale of RingSketch, showcasing its effectiveness in dynamically allocating more memory resources to fresher items.

Task extension: We extend RingSketch to the following three tasks to further demonstrate its versatility.

Estimating fresh item cardinality: Fresh item cardinality refers to the number of distinct items that have appeared in recent $\mathcal{T}_c$ time units. We aim at estimating the fresh item cardinality for any sliding window size $\mathcal{T}_c$ ($\mathcal{T}_c < \mathcal{T}$). This allows us to answer such questions as "How many users have visited a website in the past hour?", or "How many connections (flows) have been in the network in the past minute?" To our best knowledge, no prior work has addressed the problem of estimating the cardinality of fresh items.

We combine RingSketch with the method of *linear counting* [31] to give the maximum likelihood estimation for fresh item cardinality. Given the sliding window size $\mathcal{T}_c$ ($\mathcal{T}_c < \mathcal{T}$), we rotate the pointer counter-clockwise from its current position at speed $\mathcal{V}$ for $\mathcal{T}_c$ time units, incrementing 1 to each scanned counter. Afterwards, we count the number of counters with the values equal to or larger than $2^s - 1$ in each part $\mathcal{A}_i$. Let x_i be the number counters in $\mathcal{A}_i$ whose values are $\geqslant 2^s - 1$. According to the method of *linear counting* [31], $\hat{C}_i = -m \cdot \log(1 - \frac{x_i}{m})$ gives a maximum likelihood estimation for the fresh item cardinality C in $\mathcal{T}_c$. Finally, we estimate the cardinality in $\mathcal{T}_c$ as the average of $\hat{C}_i$ over the d parts, *i.e.*, $\hat{C} = \frac{1}{d}\sum_{i=1}^{d} \hat{C}_i$.

Mining batches with variable thresholds: Item batch is an important pattern in data streams, which is defined as a group of identical items that arrive closely. Two adjacent batches of the same item are spaced by a predefined time threshold $\mathcal{T}_b$. For each incoming item e, the goal of item batch detection is to report whether it is the start of a new batch, which is equivalent to reporting whether the interval between its last arrival time and the present moment exceeds $\mathcal{T}_b$. Existing algorithms for batch detection require all items to use the same time threshold $\mathcal{T}_b$, and this threshold should be known in advance so as to configure the algorithm's parameters. For example, ClockSketch [15] uses $\mathcal{T}_b$ to configure the pointer speed, and HyperBF [16] uses $\mathcal{T}_b$ to set the length of time epoch. However, in many applications, different items call for different batch threshold $\mathcal{T}_b$ and this threshold can vary with time. For example, in network traffic management, the operator might want to use different $\mathcal{T}_b$ for the flows with different service level (SLA), so as to manage different flows at different granularity (*e.g.*, performing scheduling, load balancing, and measurement), and the threshold should vary with current network load. We should use smaller batch threshold to achieve finer-grained load balancing when the network is under high load [32]. By acquiring the freshness of each item with RingSketch, we can easily set different $\mathcal{T}_b$ for different items, which extends the definition and application of item batches.

Mining periodic items: Periodic items refer to the items that arrive with a fixed time interval, which have wide applications in networks [33, 34], financial markets [35], and recommendation systems [10]. For a group of periodic items of item e_i and time interval V, we define its frequency as the number of intervals of e_i that falls in the range $[V - \Delta V, V + \Delta V)$ where ΔV is the allowable error. Top-k periodic items refer to k groups of periodic items with the k largest frequencies. To mine top-k periodic items, for each incoming item e_i, we first query RingSketch to acquire its freshness $\hat{F}_i$, and then combine its ID and its interval $V = \hat{F}_i$ to form an element $E_i = \langle e_i, V \rangle$, where the interval V is rounded according to ΔV to tolerate allowable error. Then we insert E_i into another top-k sketch (*e.g.*, GSU sketch [36], Space-Saving [4]), which will report the elements with top-k frequencies, *i.e.*, top-k periodic items.

4 Mathematical Analysis

We mathematically analyze the average error of RingSketch, where we focus on the fresh items that have appeared within recent $\mathcal{T}$ time units. We first derive the approximate average absolute error (AAE) under the assumption of no hash collision in Theorem 4.1. Then we derive the upper bound of the estimation error for a certain item in Theorem 4.2. We derive the AAE upper bound for a data stream following Zipf distribution and uniform distribution in Theorem 4.3 and Theorem 4.4, respectively. We also conduct experiments validating that our theoretical analyses are consistent with the experimental results. The detailed proof can be found in our supplementary materials [37].

Theorem 4.1. *Let $\mathcal{E}$ be the average absolute error (AAE) for estimating the freshness of all items that have appeared within recent $\mathcal{T}$ time units. Under the assumption of no hash collision, we have $\mathbb{E}(\mathcal{E}) \approx \frac{\mathcal{T}_0}{3d}$.*

Theorem 4.2. *Consider a certain item e_i that has appeared within recent $\mathcal{T}$ time units. Suppose there are w distinct items in the data stream between the last arrival time of e_i and the present moment. Let $\Delta F_i = \left|F_i - \hat{F}_i\right|$ be the estimated error of RingSketch for the freshness of e_i. We have*

$$\mathbb{E}\left(\Delta F_i\right) \leqslant \frac{\mathcal{T}_0}{3d} + (1 - \mathcal{P}) \cdot \frac{5\mathcal{T}_0}{12d} + (1 - \mathcal{P})^d \cdot \frac{\mathcal{T}}{2}$$

where $\mathcal{P} = \left(1 - \frac{1}{m}\right)^w \approx e^{-\frac{w}{m}}$ is the probability that e_i does not collide with other item in any of the d parts, and $\mathcal{T}_0 = \frac{dm}{\mathcal{V}}$ is the time for the pointer to complete one cycle.

Zipf distribution: We derive the AAE upper bound for a data stream following Zipf distribution [38]. In a Zipf distribution with N items derived from n distinct items, the k_{th} most frequent item shows up $\frac{N}{k^\alpha \zeta(\alpha)}$ times, where α is the parameter of Zipf distribution and $\zeta(\alpha) = \sum_{i=1}^{n} \frac{1}{i^\alpha}$.

Lemma 4.1. *Consider a data stream following Zipf distribution with N items derived from n distinct items and parameter α. Let e_k be the k_{th} most frequent item, and w_k be the number of distinct item in data stream since e_k's last arrival. We have*

$$\mathbb{E}\left(w_k\right) = \sum_{j=1}^{n} \frac{j^{-\alpha}}{j^{-\alpha} + k^{-\alpha}} - \frac{1}{2}$$

.

In particular, let $N_{\mathcal{T}}$ be the number of items that have appeared within recent $\mathcal{T}$ time units. When $\alpha = 1.0$, we have $\mathbb{E}\left(w_k\right) \leqslant \min\left(k \cdot \ln\left(\frac{n}{k} + 1\right) - \frac{1}{2}, \frac{N_{\mathcal{T}}}{2}\right)$. When $\alpha = 1.5$, we have $\mathbb{E}\left(w_k\right) \leqslant \min\left(2.42k - \frac{1}{2}, \frac{N_{\mathcal{T}}}{2}\right)$.

Theorem 4.3. *Consider a data stream following Zipf distribution with N items derived from n distinct items and parameter α. Let $\mathcal{E}$ be the average absolute error (AAE) for estimating the freshness of all items that have appeared within recent $\mathcal{T}$ time units. The upper bound of $\mathbb{E}(\mathcal{E})$ satisfies*

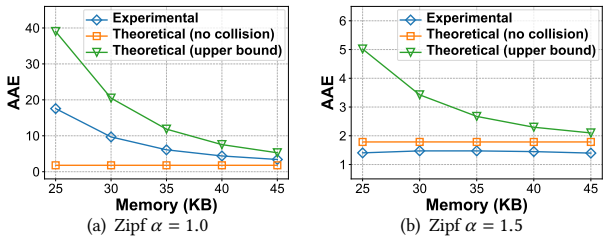
$$\mathbb{E}(\mathcal{E}) \leqslant \frac{\mathcal{T}_0}{3d} + \sum_{k=1}^{n} \frac{k^{-\alpha}}{\zeta(\alpha)} \cdot \left(\frac{5\mathcal{T}_0 w_k}{12md} + \left(\frac{w_k}{m}\right)^d \cdot \frac{\mathcal{T}}{2}\right)$$

where w_k is the number of distinct item in data stream since e_k's last arrival, which can be substituted by the expectation (or upper bound) in Lemma 4.1 to attain an upper bound.

Uniform distribution: We further analyze the performance of our RingSketch on uniform-distributed data streams. Although uniform distribution is rare in real-world streaming data, we still derive the algorithm's error bound under this distribution in Theorem 4.4 (with the same form as Theorem 4.3). We can see that under uniform distribution, RingSketch requires a larger size ($m > n$) to achieve strong error guarantees.

Theorem 4.4. *Consider a data stream following uniform distribution with N items derived from n distinct items and parameter α. Let $\mathcal{E}$ be the average absolute error (AAE) for estimating the freshness of all items that have appeared within recent $\mathcal{T}$ time units. The upper bound of $\mathbb{E}(\mathcal{E})$ satisfies*

$$\mathbb{E}(\mathcal{E}) \leqslant \frac{\mathcal{T}_0}{3d} + \frac{5\mathcal{T}_0}{24d} \cdot \frac{n-1}{m} + \left(\frac{n-1}{m}\right)^d \cdot \frac{\mathcal{T}}{4}$$



(a) Zipf $\alpha = 1.0$ (b) Zipf $\alpha = 1.5$

Figure 4: Comparison of theoretical and experimental AAE.

Experimental analysis (Figure 4): We conduct experiments to validate the effectiveness of our theoretical AAE (Theorem 4.1) and AAE upper bound (Theorem 4.3). The experiments are conducted under two data streams following Zipf distribution with $\alpha = 1.0$ and 1.5, where the items arrive at a constant speed (1 item per time unit). We set $s = 8$, $d = 6$, and $\mathcal{T} = 8192$. We can see that on both datasets, the experimental AAE is always bounded by our theoretical upper bound. As the memory usage increases, the gap between the experimental AAE and our two theoretical AAE becomes smaller because of fewer hash collisions. When using 45KB memory, the difference between the experimental and theoretical AAE will be smaller than 1.85/0.4 time unit on the two datasets, showing that our theoretical analyses are highly consistent with the experimental results.

Note that in Figure 4(b), the experimental AAE consistently exceeds the theoretical bound. This is because our theoretical lower bound in Theorem 4.1 is an approximate bound, where we use "$\approx$" instead of "=" to indicate the approximations made during the integral-based derivation (proofs detailed in [37]). On Zipf 1.5 data stream, RingSketch has so small empirical error that it may occasionally surpass the approximate bound under certain configuration.

5 Experimental Results

5.1 Experimental Setup

Platform and implementation: We conducted all CPU experiments on a server with one 18-core 4.2GHz CPU (Intel i9-10980XE), 128 GB 3200MHz DDR4 memory, and 24.75MB L3 cache. We use 32-bit Bob Hash [39] with different seeds as hash functions.

Baselines: We compare RingSketch with four algorithms: SWAMP [14], TOBF [17], HyperBF [16], and ClockSketch [15]. We adapt these algorithms to freshness estimation as described in § 2.2. We manually tune their parameters so that they can achieve high accuracy. For RingSketch, we set $d = 4$, $s = 16$ by default.

Datasets: We use two real-world datasets (CAIDA [30], Criteo [40]) and one synthetic dataset (Zipf). CAIDA and Criteo exhibit highly skewed distribution, where top-5% frequent items account for 71% and 98% total occurrences, respectively. We assume the data stream arrives at a constant speed, with one item arriving at each time unit. We measure the freshness for the items that have appeared within recent $\mathcal{T} = 8192$ time units, and report the average results.

1) CAIDA dataset is a collection of IP trace datasets. We treat each packet as one item, and use its 13-byte 5-tuple (src/dst IP, src/dst port number, protocol type) as ID. We use a small-scale 1-minute trace containing about 30M items (1.3M distinct items).

2) Criteo dataset is an advertising click data stream consisting of feature values and click feedback for many ads. For each ad, we use the hash value (8 bytes) of its categorical features as ID. We use a dataset containing 48M items (2.4M distinct items).

3) Zipf dataset: We generate multiple datasets following Zipf distribution [38] (described in § 4) with different α, which reflects the skewness degree. Each dataset has 32M items.

Metrics:

1) Average Relative Error (ARE): $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}\left|F_i - \hat{F}_i\right|/F_i$. F_i is the real freshness of e_i, $\hat{F}_i$ is its estimated freshness, and Ψ is constructed by randomly selecting items that have arrived in recent $\mathcal{T}$ time units.

2) Average Absolute Error (AAE): $\frac{1}{|\Psi|}\sum_{e_i\in\Psi}\left|F_i - \hat{F}_i\right|$.

3) Throughput: Million operations per second (M/s).

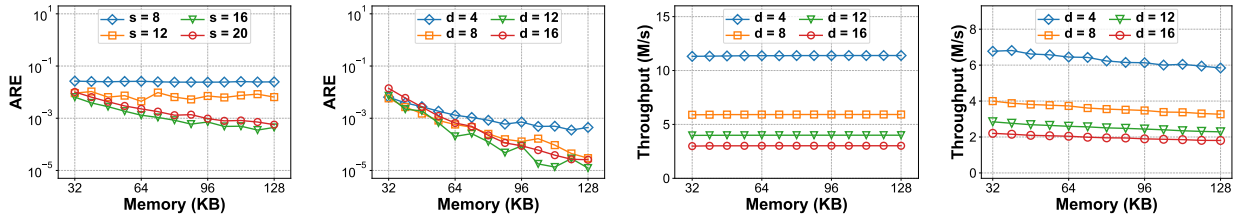
5.2 Impact of RingSketch Parameters

We first evaluate the impact of RingSketch parameters and provide a recommended setup for RingSketch. By default, we set $s = 16$, $d = 4$, and enable the multi-threading acceleration and SIMD acceleration. The experiments are conducted on CAIDA dataset.

Impact of counter size (s) on accuracy (Figure 5(a)): We find that the RingSketch using $s = 16$ bit counters can achieve satisfactory accuracy. The results show that larger counter size goes with smaller relative error of RingSketch, and the absolute error is not significantly affected by the counter size. When using 128KB memory, the RingSketch using $s = 16$ bit counters achieves 4.4×10^{-4} ARE, which is very accurate. In practice, we recommend to set $s = 16$, so that RingSketch can achieve high accuracy while being friendly to SIMD instructions.

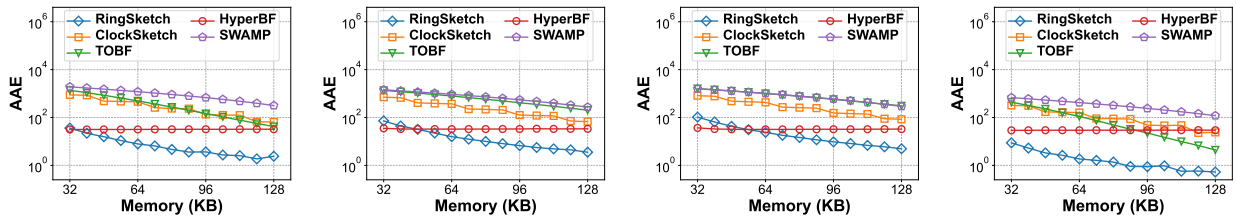
Impact of hash number (d) on accuracy (Figure 5(b)): We find that the RingSketch using $d = 4$ parts (or hash functions) can achieve satisfactory accuracy. The results show that in general, larger d goes with smaller ARE. But when $d = 4$, RingSketch can already achieve quite small error. When using 128KB memory, the RingSketch using $d = 4$ parts achieves 4.4×10^{-4} ARE, which is very accurate. In practice, we recommend to set $d = 4$ to simultaneously achieve high accuracy and fast speed.

Impact of hash number (d) on update throughput (Figure 5(c)): We find that smaller d goes with higher update throughput, and the update throughput is nearly not affected by memory usage. The



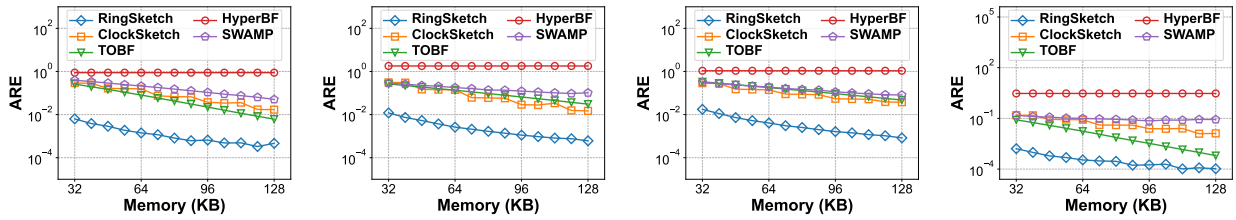
(a) ARE *vs.* counter size (s) (b) ARE *vs.* # parts (d) (c) Update throughput *vs.* # parts (d) (d) Query throughput *vs.* # parts (d)

Figure 5: Impact of RingSketch parameters (CAIDA).



(a) AAE (CAIDA) (b) AAE (Criteo) (c) AAE (Zipf 1.0) (d) AAE (Zipf 1.5)

Figure 6: Comparison of Average Absolute Error (AAE) with prior art.



(a) ARE (CAIDA) (b) ARE (Criteo) (c) ARE (Zipf 1.0) (d) ARE (Zipf 1.5)

Figure 7: Comparison of Average Relative Error (ARE) with prior art.

update throughput of the RingSketch using $d = 4, 8, 12, 16$ parts is about 11.4, 5.9, 4.0, 3.0 M/s, respectively.

Impact of hash number (d) on query throughput (Figure 5(d)): We find that smaller d goes with higher query throughput, and the query throughput is nearly not affected by memory usage. The query throughput of the RingSketch using $d = 4, 8, 12, 16$ parts is about 6.7, 4.0, 2.9, 2.2 M/s, respectively.

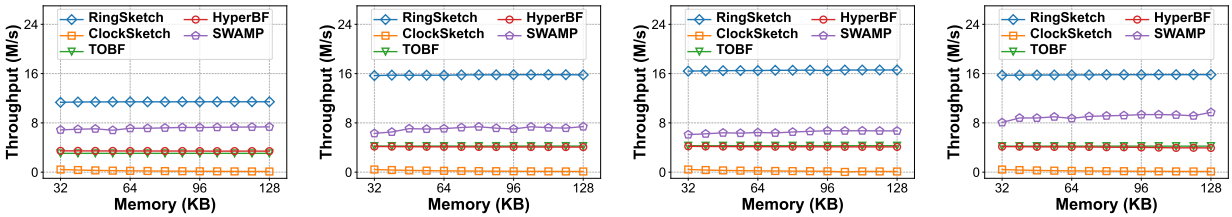
5.3 Comparison of RingSketch with Prior Art

Average Absolute Error (AAE) (Figure 6): We find that compared to prior art, RingSketch achieve about 13.5× ∼ 131.9× smaller average absolute error. On CAIDA dataset, when using 128KB memory, the AAE of RingSketch, ClockSketch, TOBF, HyperBF, SWAMP is 2.4, 64.7, 41.9, 32.3, 316.6, respectively. For RingSketch, ClockSketch, TOBF, and SWAMP, larger memory usage goes with smaller absolute error because larger memory leads to fewer hash collisions. For HyperBF, its absolute error is not significantly affected by the memory usage because its coarse-grained estimation is more affected by the epoch length than by hash collisions.

Average Relative Error (ARE) (Figure 7): We find that compared to prior art, RingSketch achieves 13.3× ∼ 1899.1× smaller average relative error. On CAIDA dataset, when using 128KB memory, the ARE of RingSketch, ClockSketch, TOBF, HyperBF, SWAMP is 4.6×10^{-4}, $1.6{\times}10^{-2}$, $6.2{\times}10^{-3}$, $8.8{\times}10^{-1}$, $5.1{\times}10^{-2}$, respectively. Similar to AAE, except for HyperBF, larger memory goes with smaller relative error. The relative error of HyperBF is not significantly affected by the memory usage because its estimation accuracy is limited by epoch length rahter than hash collisions.

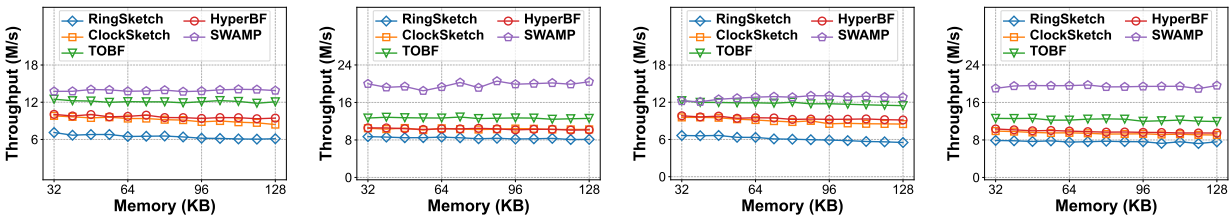
Update throughput (Figure 8): We find that compared to prior art, RingSketch achieves about 1.5× ∼ 57× higher update throughput. The update throughputs of all algorithms are not significantly affected by memory usage. On CAIDA dataset, when using 128KB memory, the update throughput of RingSketch, ClockSketch, TOBF, HyperBF, SWAMP is 11.4, 0.2, 3.1, 3.4, 7.4 M/s, respectively. ClockSketch has low update throughput because in order to achieve satisfactory accuracy, its pointer needs to scan a large number of counters per time unit, which is time consuming.

Query throughput (Figure 9): We find that the query throughput of RingSketch is 1.3× ∼ 1.9× slower than that of prior art. On CAIDA dataset, when using 128KB memory, the query throughput of RingSketch, ClockSketch, TOBF, HyperBF, SWAMP is 7.1, 9.8, 12.4, 10.0, 13.7 M/s, respectively. RingSketch has the lowest query throughput because it requires complex computations for collision identification and pointer tracing. By contrast, other algorithms are not tailored for freshness estimation. When directly adapted to freshness estimation, their query operations do not involve complex computations, resulting in faster speeds.



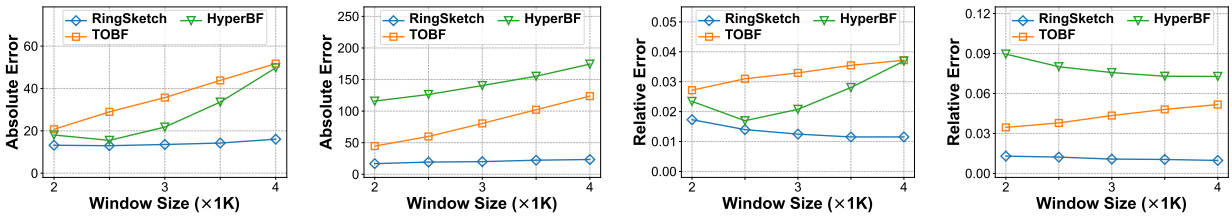
(a) Update throughput (CAIDA) (b) Update throughput (Criteo) (c) Update throughput (Zipf 1.0) (d) Update throughput (Zipf 1.5)

Figure 8: Comparison of update throughput with prior art.



(a) Query throughput (CAIDA) (b) Query throughput (Criteo) (c) Query throughput (Zipf 1.0) (d) Query throughput (Zipf 1.5)

Figure 9: Comparison of query throughput with prior art.



(a) Absolute Error (CAIDA) (b) Absolute Error (Zipf 1.0) (c) Relative Error (CAIDA) (d) Relative Error (Zipf 1.0)

Figure 10: Comparison of average error on fresh item cardinality estimation with prior art.

5.4 Performance on Extended Tasks

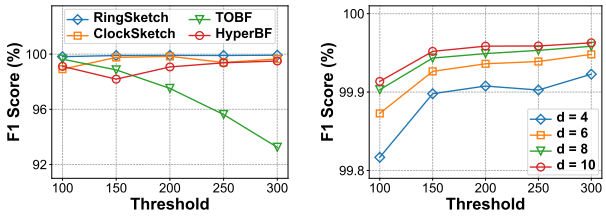
Estimating fresh item cardinality: We evaluate the performance of RingSketch on estimating fresh item cardinality. As there are no existing works designed for this task, following the idea of RingSketch, we adapt HyperBF [16] and TOBF [17] to this task by combining them with *linear counting* [31]. Specifically, to estimate the cardinality for the items in sliding window $\mathcal{T}_c$, we count the number of cells x whose epochs are less than $\mathcal{T}_c$ time units ahead of the current moment, and estimate the cardinality as $\hat{C}_i = -m \cdot \log(1 - \frac{x}{m})$, where m is the number of cells in HyperBF. For TOBF, we count the number of timestamps x within $\mathcal{T}_c$ time units ahead of the current moment, and estimate the cardinality as $\hat{C}_i = -m \cdot \log(1 - \frac{x}{m})$, where m is the number of timestamps in TOBF. For RingSketch, we set $d = 1$ and $s = 16$. To demonstrate the high efficiency of our solution, we only use 5KB memory.

1) Absolute Error on cardinality estimation (Figure 10(a)-10(b)): We find that compared to prior art, RingSketch achieves up to about 100× smaller absolute error on cardinality estimation. On CAIDA dataset, the absolute error of RingSketch, HyperBF, and TOBF is 16.04, 49.72, 51.72 when window size $\mathcal{T}_c = 4000$ and 13.54, 21.77, 35.69 when window size $\mathcal{T}_c = 3000$. We can see that larger window size goes with higher absolute error of TOBF and HyperBF. This is because when using larger window size, the number of items within the window increases, resulting in more hash collisions. By contrast, the Absolute Error of RingSketch is almost unaffected by window size, which is significantly smaller than that of HyperBF and TOBF.

2) Relative Error on cardinality estimation (Figure 10(c)-10(d)): We find that compared to prior art, RingSketch achieves up to about 10^5× smaller relative error on cardinality estimation. On CAIDA dataset, the relative error of RingSketch, HyperBF, and TOBF is $1.15 \times 10^{-2}, 3.69 \times 10^{-2}, 3.71 \times 10^{-2}$ when window size $\mathcal{T}_c = 4000$ and $1.25 \times 10^{-2}, 2.08 \times 10^{-2}, 3.29 \times 10^{-2}$ when window size $\mathcal{T}_c = 3000$. On Criteo dataset, the relative error of RingSketch, HyperBF, and TOBF is 4.38×10^{-4}, 108.14, 3.83×10^{-3} when $\mathcal{T}_c = 4000$ and 3.76×10^{-4}, 96.15, 3.23×10^{-3} when $\mathcal{T}_c = 3000$. The error of HyperBF fluctuates as the window size increases, because when the window size happens to be an exact multiple of its time epoch length δ, HyperBF performs better with reduced error.

The results in Figure 10 show that RingSketch achieves up to about 100× smaller absolute error and 10^5× smaller relative error compared to prior works. In addition, existing works suffer larger error as window size increases because of more hash collisions. By contrast, the error of RingSketch is almost unaffected by window size, which is significantly smaller than that of prior works.

Mining item batches with variable thresholds: We evaluate the performance of RingSketch on detecting item batches with variable thresholds. We adapt HyperBF [16], TOBF [17], and ClockSketch

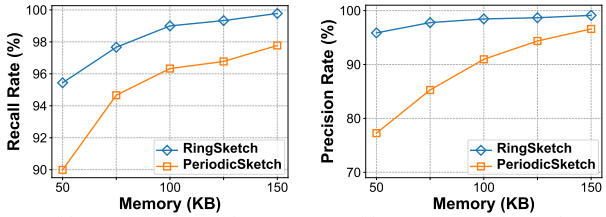


(a) F1 Score (CAIDA) (b) F1 Score (CAIDA)

Figure 11: Performance on mining item batches with variable thresholds.

[15] to this task by first using them to report item freshness as described in § 2.2, and then check whether the reported freshness exceeds current batch threshold $\mathcal{T}_b$. For RingSketch, we set $s = 8$, $d = 4$ by default. To demonstrate the high efficiency of our solution, we only use 5 KB memory for all algorithms. The experiments are conducted under CAIDA dataset, where we evaluate the F1 score on detecting batches with different threshold $\mathcal{T}_b$.

As shown in Figure 11(a), RingSketch achieves nearly 100% F1 Score on finding batches with variable threshold, which is higher than prior art. The F1 Score of RingSketch, ClockSketch, TOBF, HyperBF on detecting batches with $\mathcal{T}_b = 150$ is 99.90%, 99.76%, 98.85%, 98.18%. For TOBF, larger batch threshold goes with smaller F1 Score because of more hash collisions. The F1 Score of ClockSketch and HyperBF fluctuates with batch threshold because when $\mathcal{T}_b$ happens to be an exact multiple of their epoch length δ, their freshness estimation has minimal error. Figure 11(b) further illustrate the impact of hash number d in RingSketch on batch detection, showing that larger d goes with higher F1 Score.



(a) Recall Rate (CAIDA) (b) Precision Rate (CAIDA)

Figure 12: Performance on mining periodic items.

Mining periodic items: We evaluate the performance of RingSketch on detecting periodic items in data streams. We first use RingSketch to report item freshness, and then use a GSU sketch [36] to report top-k periodic items (ID and interval pairs). We compare our solution with PeriodicSketch [36], which is the SOTA solution for finding periodic items. For RingSketch, we set $s = 8$ and $d = 6$ by default. The experiments are conducted under CAIDA dataset, where we evaluate the Recall/Precision Rate on finding top-3000 periodic items.

As shown in Figure 12(a), RingSketch achieves > 95% Recall Rate, which is at most 40% higher than SOTA PeriodicSketch. When using 50KB memory, the Recall Rate of RingSketch and PeriodicSketch is 95.44%, 89.99%. RingSketch has higher Recall Rate than SOTA PeriodicSketch because its freshness estimation is more accurate, thereby it sends more accurate elements to the top-k sketch. As shown in Figure 12(b), RingSketch achieves > 95% Precision Rate, which is at most 60% higher than SOTA PeriodicSketch. When using 50KB memory, the Precision Rate of RingSketch and PeriodicSketch is 95.87%, 77.26%. RingSketch has higher Precision Rate because of its more accurate freshness estimation.

6 Conclusion

Item freshness is an important attribute in data streams, which can play an important role in many applications. This paper proposes RingSketch, a time-aware sketch algorithm to real-time measuring item freshness. As the first streaming algorithm tailored for measuring item freshness, RingSketch simultaneously achieves high accuracy and fast update speed. We mathematically derive the average error for RingSketch and validate the theoretical results with experiments. Extensive experiments demonstrate that RingSketch significantly outperforms the baseline solutions in terms of accuracy and speed, and RingSketch also performs well in three extended tasks. In the future, we plan to apply RingSketch to more scenarios and use our results to improve the performance of recommendation systems and network traffic management systems.

Acknowledgments

This work is supported by National Key R&D Program of China (No. 2024YFB2906603), Beijing Natural Science Foundation (Grant No. QY23043), and National Natural Science Foundation of China (NSFC) (No. 62372009).

References

[1] Ringsketch codes. https://github.com/RingSketch/RingSketch.

[2] Jonathan A Silva, Elaine R Faria, Rodrigo C Barros, Eduardo R Hruschka, André CPLF de Carvalho, and João Gama. Data stream clustering: A survey. *ACM Computing Surveys*, 46(1):1–31, 2013.

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

[4] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT 2005*, pages 398–412. Springer, 2005.

[5] Jakub Lemiesz. On the algebra of data sketches. *Proceedings of the VLDB Endowment*, 14(9):1655–1667, 2021.

[6] Rana Shahout, Roy Friedman, and Ran Ben Basat. Together is better: Heavy hitters quantile estimation. *Proceedings of the ACM on Management of Data (SIGMOD 23)*, 1(1):1–25, 2023.

[7] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *SIGCOMM 16*, pages 101–114, 2016.

[8] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd annual Southeast regional conference*, pages 267–272, 2004.

[9] Yikai Zhao, Wenrui Liu, Fenghao Dong, Tong Yang, Yuanpeng Li, Kaicheng Yang, Zirui Liu, Zhengyi Jia, and Yongqiang Yang. P4lru: Towards an lru cache entirely in programmable data plane. In *SIGCOMM 23*, pages 967–980, 2023.

[10] Tong Chen, Hongzhi Yin, Hongxu Chen, Hao Wang, Xiaofang Zhou, and Xue Li. Online sales prediction via trend alignment-based multitask recurrent neural networks. *Knowledge and Information Systems*, 62(6):2139–2167, 2020.

[11] Shan Sinha, Srikanth Kandula, and Dina Katabi. Harnessing tcp's burstiness with flowlet switching. In *Hotnets 2004*. Citeseer, 2004.

[12] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *NSDI 17*, pages 407–420, 2017.

[13] Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. Flowtune: Flowlet control for datacenter networks. In *NSDI 17*, pages 421–435, 2017.

[14] 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 2018*, pages 2204–2212. IEEE, 2018.

[15] Peiqing Chen, Dong Chen, Lingxiao Zheng, Jizhou Li, and Tong Yang. Out of many we are one: Measuring item batch with clock-sketch. In *SIGMOD 21*, pages 261–273, 2021.

[16] Zirui Liu, Chaozhe Kong, Kaicheng Yang, Tong Yang, Ruijie Miao, Qizhi Chen, Yikai Zhao, Yaofeng Tu, and Bin Cui. Hypercalm sketch: One-pass mining periodic batches in data streams. In *2023 IEEE 39th International Conference on Data Engineering. IEEE*, 2023.

[17] 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*, pages 590–599. Springer, 2006.

[18] Fernando J Corbato et al. A paging experiment with the multics system. *Massachusetts Institute of Technology*, 1968.

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

[20] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *ICALP*, pages 693–703, 2002.

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

[22] 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 *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 25–33, 2019.

[23] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. Toward nearly-zero-error sketching via compressive sensing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 1027–1044, 2021.

[24] Pinghui Wang, Yitong Liu, Zhicheng Li, and Rundong Li. An ldp compatible sketch for securely approximating set intersection cardinalities. *Proceedings of the ACM on Management of Data*, 2(1):1–27, 2024.

[25] Peng Jia, Pinghui Wang, Rundong Li, Junzhou Zhao, Junlan Feng, Xidian Wang, and Xiaohong Guan. A compact and accurate sketch for estimating a large range of set difference cardinalities. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 1338–1351. IEEE, 2024.

[26] Pinghui Wang, Dongdong Xie, Junzhou Zhao, Jinsong Li, Zhicheng Li, Rundong Li, Yang Ren, and Jia Di. Half-xor: A fully-dynamic sketch for estimating the number of distinct values in big tables. *IEEE Transactions on Knowledge and Data Engineering*, 36(7):3111–3125, 2024.

[27] Peng Jia, Pinghui Wang, Jing Tao, and Xiaohong Guan. A fast sketch method for mining user similarities over fully dynamic graph streams. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1682–1685. IEEE, 2019.

[28] Pinghui Wang, Yiyan Qi, Yu Sun, Xiangliang Zhang, Jing Tao, and Xiaohong Guan. Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. *Proceedings of the VLDB Endowment*, 11(2):162–175, 2017.

[29] Yiyan Qi, Rundong Li, Pinghui Wang, Yufang Sun, and Rui Xing. Qsketch: An efficient sketch for weighted cardinality estimation in streams. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 2432–2443, 2024.

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

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

[32] Zirui Liu, Yikai Zhao, Zhuochen Fan, Tong Yang, Xiaodong Li, Ruwen Zhang, Kaicheng Yang, Zihan Jiang, Zheng Zhong, Yi Huang, et al. Burstbalancer: Do less, better balance for large-scale data center traffic. *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[33] Eric Cole. *Advanced persistent threat: understanding the danger and how to protect your organization*. Newnes, 2012.

[34] Z-L Zhang, Vinay J Ribeiro, Sue Moon, and Christophe Diot. Small-time scaling behaviors of internet backbone traffic: An empirical study. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, volume 3, pages 1826–1836. IEEE, 2003.

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

[36] 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 22)*, pages 96–109. IEEE, 2022.

[37] Supplementary materials. https://github.com/RingSketch/RingSketch/blob/main/RingSketch_Supplementary.pdf.

[38] David MW Powers. Applications and explanations of Zipf's law. In *EMNLP-CoNLL*. ACL, 1998.

[39] Bobhash. http://burtleburtle.net/bob/hash/evahash.html.

[40] Criteo click feedback logs dataset. Available: https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/.

[41] Murmurhash. https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.

[42] Cityhash. https://github.com/google/cityhash.

[43] Farmhash. https://github.com/google/farmhash.

[44] Hailin Zhang, Zirui Liu, Boxuan Chen, Yikai Zhao, Tong Zhao, Tong Yang, and Bin Cui. Cafe: Towards compact, adaptive, and fast embedding for large-scale recommendation models. *Proceedings of the ACM on Management of Data*, 2(1):1–28, 2024.

[45] Zirui Liu, Hailin Zhang, Boxuan Chen, Zihan Jiang, Yikai Zhao, Yangyu Tao, Tong Yang, and Bin Cui. Cafe+: Towards compact, adaptive, and fast embedding for large-scale online recommendation models. *ACM Transactions on Information Systems*.

[46] Yao Tian, Tingyun Yan, Ruiyuan Zhang, Kai Huang, Bolong Zheng, and Xiaofang Zhou. A learned cuckoo filter for approximate membership queries over variable-sized sliding windows on data streams. *Proceedings of the ACM on Management of Data*, 1(4):1–26, 2023.

A Additional Experimental Results

A.1 Performance on More Hash Functions

We evaluate the performance of RingSketch on more hash functions in Table 2: BobHash [39], MurmurHash [41], CityHash [42], and FarmHash [43]. The results show that ARE remains nearly unchanged across different hash functions, and faster hash functions lead to higher throughput.

Table 2: Performance of RingSketch on more hash functions.

Hashes	ARE	Throughput (M/s)	
		Update	Query
BobHash	0.032	12.01	6.10
MurmurHash	0.036	15.91	7.05
CityHash	0.034	22.86	6.54
FarmHash	0.034	25.58	6.65

A.2 Performance on Different Pointer Speed

We conduct experiments evaluating the freshness measurement accuracy under different pointer rotation speed (32K memory, $s = 16$, $d = 4$). The results in Table 3 show that higher rotation speed goes with smaller error. This is because higher rotation speed correlates with smaller measurement window size, thereby simplifying the measurement problem.

Table 3: Performance of RingSketch under various pointer scanning speed (number of counters per time unit).

Speed	ARE	AAE
64K	1.3×10^{-2}	162.29
128K	2.5×10^{-3}	15.36
256K	3.5×10^{-4}	1.13
512K	3.4×10^{-5}	0.053
1024K	5.2×10^{-6}	0.0050

A.3 Performance on Different Data Skewness

Like many other sketches, RingSketch also uses sublinear space to approximate the key (temporal) information of hot items (fresh items) in data stream. In RingSketch, hot items retain more valid counters, enabling more accurate freshness estimation. By contrast, cold items retain fewer valid counters, and thus yield coarser-grained estimation. When data stream contains more cold items (i.e., lower skewness), RingSketch's overall accuracy decreases under fixed memory usage.

In practice, Zipf distribution is widely used for modeling data streams, which aligns with the heavy-tailed phenomena observed in real-world scenarios. We evaluate the accuracy of RingSketch (under fixed memory of 64KB) on Zipf data streams with varying skewness in Table 4. The results confirm that cold items indeed affect the accuracy of RingSketch.

Table 4: Performance of RingSketch on different data skewness (α of Zipf distribution).

Skewness (α)	AAE	ARE
Zipf 1.0	7.80	1.2×10^{-3}
Zipf 1.1	1.53	2.4×10^{-4}
Zipf 1.2	0.23	3.6×10^{-5}
Zipf 1.3	0.023	3.5×10^{-6}
Zipf 1.4	0.0017	3.9×10^{-7}
Zipf 1.5	5.9×10^{-6}	7.0×10^{-10}
Zipf 1.6	4.5×10^{-6}	6.0×10^{-10}
Zipf 1.7	3.6×10^{-6}	4.0×10^{-10}
Zipf 1.8	7.2×10^{-7}	1.0×10^{-10}

We also evaluate the accuracy of RingSketch and compare it with prior art in uniform-distributed data streams. The results in Table 5 show that RingSketch also outperforms other solutions under uniform-distributed data.

Table 5: Comparison of accuracy on uniform distribution.

Methods	AAE	ARE
RingSketch	9.15	0.0013
ClockSketch	12.09	0.0014
TOBF	425.47	0.068
HyperBF	43.97	0.048
SWAMP	3141.23	0.62

A.4 Performance on End-to-end Applications

To better demonstrate RingSketch's effectiveness in real-world application scenarios, we use it to enhance the performance of two end-to-end applications: caches and recommendation systems.

Cache: RingSketch enables a space-time efficient approximate-LRU cache that uses freshness to guide LRU replacement, where we directly implement the rotating-pointer in set-associative cache lines. The results in Table 6 (4K cache, Zipf1.2) show our solution achieves almost the same hit rates as standard LRU (implemented with hash table and doubly-list), while delivering 3.2× faster update and 5.9× space reduction.

Table 6: RingSketch in optimizing LRU cache (Zipf 1.2).

Methods	Hit Rate	Speed (M/s)	Space (KB)
Standard LRU	83.93%	4.35	224
LRU w/ RingSketch	83.91%	14.02	38

We also evaluate the hit rate on varying Zipf skewness in Table 7. The results confirm that our solution consistently achieves similar

Table 7: Performance of RingSketch in optimizing LRU cache (various Zipf skewness).

Skewness (α)	Standard LRU	LRU w/ RingSketch
Zipf 0.5	1.25%	1.25%
Zipf 0.6	3.08%	3.03%
Zipf 0.7	7.63%	7.47%
Zipf 0.8	16.87%	16.61%
Zipf 0.9	32.01%	31.59%
Zipf 1.0	51.01%	50.70%
Zipf 1.1	69.90%	69.44%
Zipf 1.2	83.93%	83.91%
Zipf 1.3	92.17%	92.12%
Zipf 1.4	96.39%	96.24%
Zipf 1.5	98.38%	98.31%

hit rates as standard LRU across all skewness levels. Note that the speed and space results do not vary with the skewness levels.

Recommendation system: We integrate RingSketch into a recent deep learning recommendation model (DLRM) framework called CAFE [44, 45]. We use RingSketch to report the freshness of features, and we periodically demote stale hot features and promote fresh ones. This enables model to focus more on recent features and adapt to real-time market dynamics. The results in Table 8 show RingSketch improves test AUC by 0.108 (CriteoTB, 100x CR).

Table 8: Performance of RingSketch in optimizing deep learning recommendation systems.

Methods	Test AUC	Train Loss
Standard CAFE	72.123	0.12623
CAFE w/ RingSketch	72.231	0.12610

A.5 Comparison with More Baselines

Besides the four baselines in our paper, there are also some works that use learning-based methods to evaluate the time gap between the current and last occurrence of an item. For example, the Learned Cuckoo Filter (LCF) [46] uses a pre-trained model (called learned oracle) to predict items' frequencies within a fixed time window, and uses the predicted frequencies to infer last-arrival time. However, as stated in its original paper, this approach assumes data distribution remain stable over time. By contrast, our RingSketch does not rely on this assumption.

Table 9: Comparison between the learned oracle in LCF [46].

Methods	ARE
LCF	45.8945
RingSketch	1.2×10^{-3}

We have also evaluated LCF using its original code and configuration, and compared it with our RingSketch on Zipf 1.0 data stream. The results in Table 9 show that LCF's freshness estimation errors are notably large. This large error implies that such learning models may face great challenges in practical applications. This is because LCF and our RingSketch have different design goals. LCF targets at the membership query problem, which only needs approximate last-arrival time. By contrast, RingSketch aims at directly report the accurate last-arrival time, which is more difficult.

B Additional Discussions

B.1 Distinction Between Item Freshness and Item Batch Time Span

We discuss the distinction between the *Item Freshness* attribute proposed in this paper and the *Item Batch Time Span* attribute proposed in ClockSketch [15]. Actually, they are two completely distinct attributes. 1) *Item Freshness* measures the time since an item's last arrival, requiring fine-grained last-arrival-time tracking. 2) *Item Batch Time Span* measures the time since an item's current batch start. To estimate *Item Batch Time Span*, ClockSketch first detects batches and then explicitly records batches' start time. Detecting batches only requires coarse-grained last-arrival-time. As RingSketch also supports batch detection (Figure 11(a)), it can directly extend to estimate *Item Batch Time Span*.

B.2 Distinction Between RingSketch and ClockSketch

We discuss the distinction between RingSketch and ClockSketch [15]. ClockSketch uses only counter values to get approximate last-arrival-time and detect batch. By contrast, freshness estimation requires exact per-item last arrival time, which is more challenging. By leveraging sketch's internal structure as an implicit clock, RingSketch innovatively proposes pointer-tracking that converts spatial rotation into temporal duration, achieving finer-grained time tracking. RingSketch is compatible with all use cases of ClockSketch, and can attain higher accuracy.

To be more specific, RingSketch differs from ClockSketch for the following two reasons. 1) RingSketch and ClockSketch have different design goals. ClockSketch is designed for detecting batch with approximate time information, while RingSketch aims at measuring freshness with accurate time information, which is more difficult. Actually, RingSketch is more general than ClockSketch. RingSketch can replace ClockSketch in all its applications and achieve better accuracy. 2) RingSketch's algorithmic design is more sophisticated than ClockSketch. ClockSketch only leverages the information of counter values. By contrast, RingSketch fully exploits sketch's internal structure as an implicit clock, converting spatial rotation into temporal duration. This novel design fully unlocks the potential of circular sketches (e.g., ClockSketch), and we hope it could inspire more new methods for recording temporal information in streaming data.