BurstSketch: Finding Bursts in Data Streams

Zheng Zhong*
Peking University
Beijing
zheng.zhong@pku.edu.cn

Shen Yan*
Peking University
Beijing
yanshen@pku.edu.cn

Zikun Li*
Peking University
Beijing
lizikun@pku.edu.cn

Decheng Tan*
Peking University
Beijing
dechengex@gmail.com

Tong Yang*†
Peking University & Pengcheng Laboratory
Beijing
yangtongemail@gmail.com

Bin Cui*
Peking University
Beijing
bin.cui@pku.edu.cn

ABSTRACT

[1] *Burst* is a common pattern in data streams which is characterized by a sudden increase in terms of arrival rate followed by a sudden decrease. Burst detection has attracted extensive attention from the research community. In this paper, we propose a novel sketch, namely BurstSketch, to detect bursts accurately in real time. BurstSketch first uses the technique Running Track to select potential burst items efficiently, and then monitors the potential burst items and capture the key features of burst pattern by a technique called Snapshotting. Experimental results show that our sketch achieves a 1.75 times higher recall rate than the strawman solution.

CCS CONCEPTS

• **Theory of computation → Sketching and sampling**; • **Information systems →** *Data stream mining*; Data streams.

KEYWORDS

data stream; sketch; burst

ACM Reference Format:
Zheng Zhong, Shen Yan, Zikun Li, Decheng Tan, Tong Yang, and Bin Cui. 2021. BurstSketch: Finding Bursts in Data Streams. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event , China.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3448016.3452775

1 INTRODUCTION

1.1 Background and Motivation

Burst is a common pattern in data streams, which is characterized by a sudden increase in terms of arrival rate followed by a sudden decrease. The arrival rate of an item refers to its number of appearances in a fixed time window. The pattern of bursts is like a pulse, which can often be observed in the natural world. The appearance of burst often indicates the happening of abnormal or notable events. For example, in financial markets, a burst of trading volume may indicate the happening of financial fraud or illegal market manipulation. For another example, when assigning bandwidth for VIP users, a burst of requests indicate the need for more bandwidth. By detecting this pattern, we can appropriately distribute the limited bandwidth resource,i.e., by assigning more bandwidth at the sudden increase and recovering it to normal level after the sudden decrease. Further, burst detection can be applied in text stream mining [1, 2], web clicks analysis [1], and bursty topic mining in social networks [3–5].

Real-time burst detection is challenging because we need to catch up with high speed (*e.g.*, 1.5M items per second) of data streams while maintaining accuracy. To achieve high speed, it is ideal that only CPU caches are accessed when processing items, which requires the data structure to be small enough to be held in caches. Therefore, the design goal of this paper is to use limited memory to accurately detect bursts in real time.

For real-time burst detection, typical works include CM-PBE [6] and TopicSketch [3]. CM-PBE can not only detect bursty events in real-time, but is also the first work that can detect bursty events from history. It is quite efficient in both time and space. TopicSketch aims at detecting hot topics in text streams. It is simple, fast, and easy to deploy. However, the definitions of bursts in these two works can only detect sudden increases, but cannot detect sudden decreases. In other words, no existing work is capable of detecting bursts in our definition which consists of a sudden increase and a sudden decrease.

1.2 Our Proposed Algorithm

Towards the design goal of this paper, we propose a novel sketch to accurately detect bursts in real time, namely, BurstSketch. To the best of our knowledge, BurstSketch is the first sketch algorithm focusing on detecting bursts in our definition in high-speed data streams. BurstSketch is memory efficient: it is small enough to be

---

*Department of Computer Science and Technology, and National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, China
†PCL Research Center of Networks and Communications, Pengcheng Laboratory, Shenzhen, China
[1] Zheng Zhong, Shen Yan, and Zikun Li contribute equally to this paper, and they together with Decheng Tan complete this work under the guidance of the corresponding author: Tong Yang.

SIGMOD '21, June 18–27, 2021, Virtual Event , China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452775

held in CPU L2 caches. BurstSketch is accurate: it achieves a 97% recall rate (using 60KB memory), which is 1.75 times higher than the strawman solution. BurstSketch is fast: its time complexity is $O(1)$. All related codes of BurstSketch are open-sourced and available at GitHub anonymously [7].

BurstSketch consists of two parts, Stage 1 and Stage 2. For each incoming item, we first check whether it is a potential burst item in Stage 1, if so, it will be sent to Stage 2. The techniques used in Stage 1 and Stage 2 are named Running Track and Snapshotting, respectively. We show the two key techniques below.

Technique I: Running Track. Running Track is used to select potential burst items. It needs to filter out infrequent items as well as items arrive at a steady speed. Running Track works as follows. We use many tracks, each item will be mapped into d tracks by hash functions $h_1(.) \dots h_d(.)$. For each track, we only observe the most frequent item. If it is frequent enough, we consider it as a potential burst item. To find the fastest item in each track, there are several optional strategies: *frequent* [8], *probabilistic decay* [9], *probabilistic replacement* [10]. We choose *frequent* since it is the simplest and fastest which has a comparative accuracy compared to others. In our strategy, high-speed items are unlikely to be filtered out in every track, because it would be selected as long as it becomes the most frequent item in at least one track.

Technique II: Snapshotting. Snapshotting is used to detect bursts from potential bursts. The rationale of Snapshotting is that a burst can be described only with the sudden increase and sudden decrease in arrival rate. Therefore, we do not need to record frequencies of items in every time window. In Snapshotting, we only take two snapshots for the sudden increase and the sudden decrease so that we can confirm whether it is a burst. Snapshotting detects bursts with $O(1)$ memory.

2 PROBLEM STATEMENT & RELATED WORK

2.1 Problem Statement

The symbols frequently used in this paper are shown in Table 2.1.

Table 1: Symbols used in this paper.

Notation	Meaning
$\mathcal{A}_i$	i^{th} bucket array of Stage 1
$\mathcal{B}$	Bucket array of Stage 2
k	parameter for the definition of sudden increase and sudden decrease
L	Maximum width of a burst
T	Burst threshold
H	Running Track threshold
C_{pre}	Frequency in the previous time window
C_{cur}	Frequency in the current time window
t	Time stamp in Stage 2

Burst Detection: *Burst*, in our definition, is a particular pattern of the changing behavior in terms of the arrival rate of an item in a data stream, and the pattern consists of a sudden increase and a sudden decrease. Specifically, we divide the data stream into fixed-width time windows. Given an item e, a sudden increase means that, in two adjacent time windows, the arrival rate of e in the second time window is no less than k times of that in the first time window. Similarly, a sudden decrease is that the arrival rate of e in the second time window is no more than $\frac{1}{k}$ of that in the first time window. Also, we do not consider infrequent bursty items as bursts, for they are not useful in most applications, so the arrival rate of a burst item should exceed a *burst threshold.* In practice, a burst occurs over a short period of time. Therefore, we set a limitation L for the width of a burst, namely, the number of time windows that the burst lasts. The formal definition of a burst is as follows.

Definition: For a time series data stream $S = \{e_{t1},\ e_{t2},\ e_{t3},\ \dots\}$, an item e and a burst threshold T, given that the data stream is divided into fixed time windows $w_1,\ w_2,\ w_3\ \dots$ and the arrival rate of e in the time windows are $r_1,\ r_2,\ r_3,\ \dots$, if there exist four time windows $w_i,\ w_{i+1},\ w_j,\ w_{j+1}$, where

$$r_{i+1} \geqslant k \cdot r_i \wedge r_{j+1} \leqslant \frac{1}{k} \cdot r_j \wedge j > i$$

and

$$r_k \geqslant T, \forall k \in \{i+1,\ \dots,\ j\} \wedge j - i \leqslant L$$

then e is a burst item, the changing process of its arrival rate is a burst, the width of the burst is $j - i$ time windows, window w_{i+1} is the sudden-increase window, and window w_{j+1} is the sudden-decrease window. If multiple sudden-increase windows happen consecutively, we just consider the latest one as the burst's possible beginning. If multiple sudden-decrease windows happen consecutively, we just consider the first one after the sudden increase as the burst's end. It can prevent multiple reports of a single burst.

High-speed Item Detection: A time series data stream S is divided into multiple equal-sized time windows $w_1, w_2, w_3 \dots$, a high-speed item refers to an item whose frequency in a time window exceeds a predefined threshold T.

2.2 The Comparison of the Definitions of Burst

For burst detection, all existing works only focus on the sudden increase of item frequency, but do not care about whether there is a sudden decrease. In this paper, we present a more complete definition with both sudden increase and sudden decrease. In some applications, the occurrence of a sudden decrease is also important. Take the example of assigning limited bandwidth for VIP users. If a VIP user's requests take on a sudden increase, we should assign enough bandwidth to the users. Importantly, we should recover the bandwidth to a normal level for the user when its requests suddenly decrease. There are many more similar examples, such as assigning more computation resources for users with burst requests, assigning more fast memory for the burst of a hot item, and overclocking the CPU frequency for the process with the burst of a computation request. For these examples, the limited resource should be recalled in time when the sudden decrease occurs.

2.3 Prior Work on Burst Detection

Several burst detecting algorithms have been proposed focusing on some specific areas, such as text stream or document stream mining [1–3, 11], astronomical observation [11] and telecommunication traffic management [12]. When it comes to generic burst detection methods [4, 11, 13–16], most of them are based on *Wavelet Tree (WT)* and *Aggregation Tree (AT).* We also survey some typical sketches [8–10, 17–32, 32–50], which we will not discuss here due to the space limitation.

Recently there are two pieces of works concerning burst detection. One is CM-PBE [6], which concentrates on detecting burst from history without storing or querying the whole stream. To identify bursty events in data streams, they propose a concept called $frequency\ curve$, which shows how the frequency of an item grows cumulatively over time. To store the frequency curve, they use dynamic programming, which enables them to approximate the curve with as few points as possible; thus, largely save the storage space. This work is the first to discuss the identification of bursty events in history with high efficiency in both space and query time. Our algorithm differs from this work in two regards. First, the definitions of bursts are different. In their work, an event that witnesses a large acceleration in its arrival rate is considered a bursty event, whereas in our definition, burst consists of a sudden increase and a sudden decrease in its arrival rate. Besides, our algorithm cares more about real-time burst detection in high-speed data streams, while their work puts a premium on bursty events detection in history. Another one is TopicSketch [3] from Wei et al. Their definition of burst is close to the definition from CM-PBE, which is different from ours, as mentioned above. Therefore, they also use the acceleration of items' arrival rate as a metrics of burst. To calculate the acceleration, they incrementally maintain velocities of two time windows. Thus, the acceleration can be derived on the fly. Apart from the definition of bursts, their work differs from ours also in that our algorithm is general-purpose, while they only focus on burst topic detection.

3 BURSTSKETCH ALGORITHM

3.1 The Strawman Solution

The strawman solution is based on CM sketch. CM sketch consists of k counter arrays, each associated with a hash function. For each incoming item, the hash function is calculated to map it to a mapping bucket in each array, then all the mapping buckets of the item is increased by 1. To report the estimated frequency of an item, the CM sketches output the minimum value among the mapping buckets. In the strawman solution , we construct $L+2$ CM sketches to store the estimated frequencies of the latest $L+2$ time windows to detect burst whose width no larger than L. We use a queue to store potential burst items. Whenever the frequency of an item in a window is larger than the burst threshold, we insert its flow ID into the queue. At the end of each time window, for potential burst items, we query their frequencies from CM sketches to find burst patterns. Although the strawman solution is capable to detect bursts, it is memory consuming and inaccurate. Because it stores information of $L+2$ windows and takes into account many items that are not potential bursts.

3.2 The Burst Sketch

Rationale: In this paper, we propose a novel sketch, namely BurstSketch. BurstSketch consists of two stages. To avoid recording unnecessary information, the first stage checks whether an incoming item is a potential burst item. We only send the potential items to the second stage for burst detection. To detect a burst, rather than recording the frequencies of $L+2$ time windows for each item, Stage 2 only records the frequencies of 2 adjacent time windows for potential burst items to detect whether there exists sudden increase or sudden decrease, and we use a timestamp to snapshot it. In summary, compared to the strawman solution, our BurstSketch filters out much more unnecessary information.

Data Structure: As shown in Figure 1, BurstSketch has two stages: Stage 1 using **Running Track** to filter low arrival rate items, and Stage 2 using **Snapshotting** to find burst patterns. Stage 1 consists of d bucket arrays $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_d$, and each array consists of m buckets. There are d hash functions $h_1(.), h_2(.), \dots, h_d(.)$ associating with d bucket arrays respectively. Each bucket has two fields: item ID (key) and frequency. We have a Running Track threshold H to determine whether the item is a potential burst item. It is worth noting that the number of tracks determines the maximum number of bursts our BurstSketch can detect simultaneously. A single track takes up only several bytes, but more tracks enable us to detect more bursts simultaneously, and also lessens hash collisions. Therefore, we recommend using enough tracks to achieve higher accuracy. Stage 2 is a bucket array $\mathcal{B}[1], \mathcal{B}[2], \dots, \mathcal{B}[M]$ associated with a hash function $g(.)$. Each bucket has s cells. Each cell has four fields: item ID (key), two counters C_{pre} and C_{cur}, timestamp t. C_{pre} is used to record the frequency of the item in the previous time window, while C_{cur} is used to record the frequency of the item in the current time window. The timestamp records the time window in which the latest sudden increase happened. If the timestamp is equal to 0, it means no sudden increase occurred.

Algorithm 1: Insertion-BurstSketch

```
Input: an item $e$; $H$, the Running Track threshold;
if $e$ is in $\mathcal{B}[g(e)]$ then
    $e.C_{cur} \leftarrow e.C_{cur} + 1$;
else
    for each $i \in [1, d]$ do
        if $e$ is in $\mathcal{A}_i[h_i(e)]$ then
            increase the frequency of $e$ by 1;
            if the frequency of $e \geq H$ then
                if Insert_Stage2 ($e$, the frequency of $e$) then
                    clear $\mathcal{A}_i[h_i(e)]$ to empty;
        else if $\mathcal{A}_i[h_i(e)]$ is empty then
            insert $e$ into $\mathcal{A}_i[h_i(e)]$ and set the frequency of $e$ to 1;
        else if $e$ is not in $\mathcal{A}_i[h_i(e)]$ and $\mathcal{A}_i[h_i(e)]$ is not empty then
            decrease the frequency of $\mathcal{A}_i[h_i(e)]$ by 1;
            if the frequency of $\mathcal{A}_i[h_i(e)]$ is 0 then
                clear $\mathcal{A}_i[h_i(e)]$ to empty;
Function Insert_Stage2($e, C$):
    if $C >$ the $C_{cur}$ of the smallest item in $\mathcal{B}[g(e)]$ then
        use $e$ to replace the smallest item;
        $e.C_{cur} \leftarrow C$; $e.C_{pre} \leftarrow 0$;
        return 1;
    return 0;
```

Insertion: Given an incoming item e, if e is in Stage 2, we increment $e.C_{cur}$ by 1. Otherwise, we insert it into Stage 1: we hash e into d mapping buckets of Stage 1 $\mathcal{A}_1[h_1(e)], A_2[h_2(e)], \dots, A_d[h_d(e)]$. For each bucket, there are 3 cases.

Case 1: e is not in the bucket, and the bucket is empty. In this case, we insert e into the bucket with the frequency of 1.

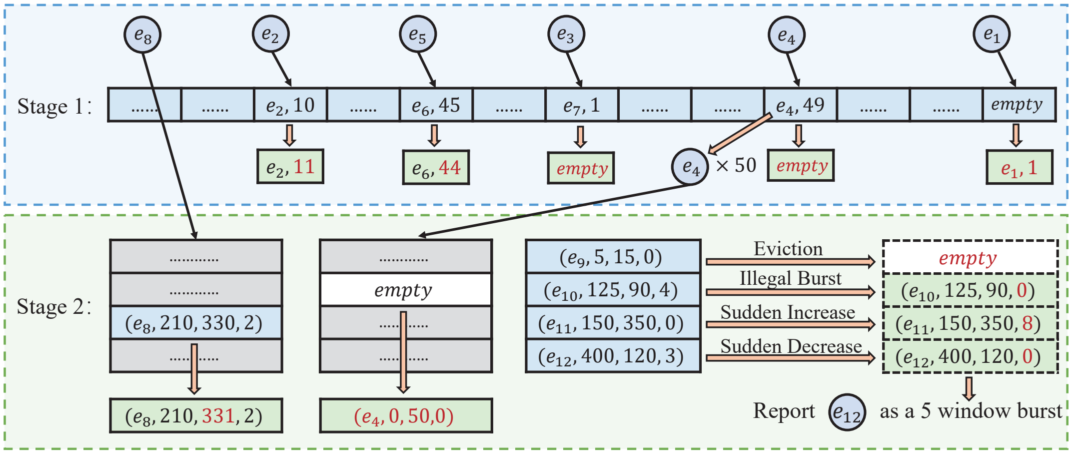


Figure 1: An example of BurstSketch using one hash function.

Case 2: e is not in the bucket, and the bucket is not empty. In this case, we decrement the frequency of the bucket by 1. If the frequency is decreased to 0, we empty the bucket. We need a replacement strategy to allow a new potential burst to get in when it is hashed into a full bucket. There are three typical replacement strategies, namely, Frequent [8], probabilistic decay [9], and probabilistic replacement [10]. We choose Frequent for it is fast and easy to implement.

Case 3: e is in the bucket. We just increment the frequency of e by 1. If the frequency of e is equal to or larger than the Running Track threshold H, we try inserting e into Stage 2 (because e is frequent enough): if we find an empty cell in the bucket $\mathcal{B}[g(e)]$, we insert e in it with its frequency. Otherwise, we try evicting the smallest item whose timestamp t is 0: if the frequency of the item is smaller than the frequency of e, we evict the item and insert e with its frequency. If all the items' t are not 0, we try evicting the smallest item in the bucket with the same method. Stage 2 stores and monitors potential bursts. The space in Stage 2 is limited, so we need to evict the items that are not likely to become a burst when the corresponding bucket is full.

Detection: Stage 2 uses Snapshotting to capture the sudden increase and the sudden decrease for each item, and reports bursts in the end of each time window. For item e, suppose the max width of a burst is L. First we detect if there is a sudden increase or sudden decrease: we check the frequencies of e in the latest two time windows. If $\frac{e.C_{cur}}{e.C_{pre}} \geq 2$, the sudden increase happens. Then we update the current time window into t. Specially, if e has been inserted into Stage 2 in the current time window (which means we do not know $e.C_{pre}$), we regard $e.C_{pre}$ as 0. If $\frac{e.C_{cur}}{e.C_{pre}} \leq \frac{1}{2}$, a sudden decrease happens. Then we check whether there has been a sudden increase and whether the difference between t and the current time window is no more than L. If so, BurstSketch reports a burst which has t as its sudden-increase window and the current time window as its sudden-decrease window, then we clean $e.t$ to 0. Otherwise, no burst is reported and t remains unchanged.

Cleaning Policy: In Stage 1, we clean all arrays at the end of each time window. In Stage 2, we evict the items whose arrival rates are always low. Specifically, at the end of every time window, we check if the frequencies of the latest two time windows are both lower than H. If so, we evict the item. We also clean illegal potential bursts, whose frequency is smaller than T in the current time window. If so, we clean t of the item to 0.

An running example: Figure 1 show an running example of BurstSketch. In this example, in Stage 2, given a bucket with $(e_{10}, 125, 90, 4)$, e_{10} is the item ID, 125 is e_{10}'s frequency in the previous time window C_{pre}, 90 is e_{10}'s frequency in the current time window C_{cur}, 4 is the time when the latest sudden increase happens. Suppose the Running Track threshold $H = 50$, the burst threshold $T = 100$, and the time of this example is at the end of time window 8. 1) To insert e_8, we find it in Stage 2, so we just increment $e_8.C_{cur}$ by 1. 2) To insert e_2, we find it in Stage 1, so we just increment the frequency of e_2 by 1. 3) To insert e_5, we do not find it in both stages, so we decrement the frequency of the item in the mapped bucket by 1. 4) To insert e_3, we decrement the frequency of e_7 from 1 to 0, then we evict e_7. 5) To insert e_4, we find it in Stage 1, so we increment e_4 by 1. After the increment, the frequency of e_4 reaches the Running Track threshold and we find an empty cell in Stage 2. Then we clean e_4 in Stage 1 and insert it into Stage 2 with the frequency of 50. 6) To insert e_1, we find an empty bucket in Stage 1, so we insert e_1 with the frequency of 1. At the end of every time window, we check if there is any sudden increase, sudden decrease, illegal burst, or legal burst. At the same time, we evict the items which are not potential anymore. 7) For e_9, both $e_9.C_{pre}$ and $e_9.C_{cur}$ are below 50, so we evict e_9 from Stage 2. 8) For e_{10}, $e_{10}.C_{cur}$ is below 100, it means it is an illegal burst, so we clean its timestamp to 0. 9) For e_{11}, $\frac{e_{11}.C_{cur}}{e_{11}.C_{pre}} = \frac{350}{150} \geq 2$, it means a sudden increase happens. Therefore, we record the current time window 8 into the timestamp field. 10) For e_{12}, $\frac{e_{12}.C_{cur}}{e_{12}.C_{pre}} = \frac{120}{400} \leq \frac{1}{2}$, it means a sudden decrease happens. And we find the width of the burst (*i.e.*, $8 - 3 = 5$) is legal. Therefore, we report e_{12} as a burst with a width of 5. Then we clean the timestamp of e_{12}.

Bursts inside bursts: We have an extended version to detect bursts inside bursts. The definition of bursts inside bursts is similar to *bracket matching*: sudden increase corresponds to left bracket and sudden decrease corresponds to right bracket. To detect bursts inside bursts, the ideal algorithm works as follows. We add a stack for each item in Stage 2. When a sudden increase happens, we push a timestamp with current time into the stack. If the stack is full, we delete the oldest timestamp, which is at the bottom of the stack. We use an array with two pointers (a header and a tail) to implement the stack, and thus can delete the timestamp from the bottom of the stack. When a sudden decrease happens, we pop the timestamp (the most recent sudden increase) from the top of the stack, and report the pair of sudden increase and sudden decrease as bursts inside bursts. If the stack is empty, we do nothing.

3.3 Optimization: Deduplication

In Stage 1, we record the fastest item (whose arrival rate is fastest) in each bucket. However, a high-speed item may occupy more than one bucket, which is redundant. Therefore, reducing copies of high-speed items can save memory for BurstSketch. Therefore, we modify the insertion of Stage 1. Given an incoming item e, if we do not find e in Stage 2, we map e into d mapping buckets of Stage 1 according to three cases:

Case 1: e is not in any bucket, and none of the buckets is empty. In this case, we decrement the frequency of each bucket by 1. If the frequency is decreased to 0, we empty the bucket.

Case 2: e is not in any bucket, and at least one of the buckets is empty. In this case, we insert e into one of the empty buckets.

Case 3: e is in a bucket. We just increment the frequency of e by 1. If the frequency of e is equal to or larger than the Running Track threshold H after the increment, we try inserting e into Stage 2 the same as in the basic version.

3.4 BurstSketch Can Do More

Except for finding bursts, BurstSketch can also find high-speed items. We divide the data streams into short time windows to detect its speed. For items in Stage 1, the frequency of the item reports its arrival rate. For items e in Stage 2, $e.C_{cur}$ reports its arrival rate. There is no overestimation error in the arrival rate. Suppose the threshold of a high-speed item is K. We check every bucket in Stage 1 and Stage 2 at the end of each time window, if the arrival rate of an item is higher than K, we report it as a high-speed item.

4 MATHEMATICAL ANALYSIS

In this section, we provide theoretical analysis for BurstSketch. First, we derive the error bound of Stage 1 in Section 4.1. Then we show an upper bound of the number of distinct items in Stage 2 in Section 4.2. Finally, we show that there is no overestimation error in Section 4.3.

4.1 The Error Bound of Stage 1

LEMMA 4.1. *Given a time series data stream S which has fixed window size. In a window w, for item e_i, suppose e_i does not in Stage 2. Let $F_{i,j,k}$ be the number of items mapping to bucket $\mathcal{A}_j[k]$ in w except for item e_i, f_i be the frequency of e_i in w, $\mathcal{A}_j[k].ID$ be the ID of bucket $\mathcal{A}_j[k]$, $\mathcal{A}_j[k].count$ be the frequency of bucket $\mathcal{A}_j[k]$. Suppose $f_i > F_{i,j,k}$, which means e_i is in the majority in this bucket, we have $\mathcal{A}_j[k].ID = e_i$ and $f_i - F_{i,j,k} \le \mathcal{A}_j[k].count \le f_i$.*

PROOF. Since each item which is not e_i can at most counteract one e_i, so there at least remains $f_i - F_{i,j,k}$ numbers of e_i. Therefore, $\mathcal{A}_j[k].ID = e_i$ and $f_i - F_{i,j,k} \le \mathcal{A}_j[k].count$. $\mathcal{A}_j[k].count \le f_i$ is obvious because $\mathcal{A}_j[k].count$ increases only when the item is equal to $\mathcal{A}_j[k].ID$. □

THEOREM 4.2. *Given a time series data stream S which has fixed window size W. In a window w, suppose $\mathcal{A}_j[k].ID = e_i$, let f_i be the frequency of item e_i in w. For $0 < \varepsilon < f_i$, we have*

$$Pr\{f_i - \mathcal{A}_j[k].count \ge \varepsilon\} \le \frac{W - f_i}{m\varepsilon} \tag{1}$$

PROOF. By the linearity of the expectation and the pairwise independence of the hash functions, we have

$$E[F_{i,j,k}] = E[\sum_{e \neq e_i} f_e I_{h_j(e)=h_j(e_i)}] = \sum_{e \neq e_i} f_e \frac{1}{m} = \frac{W - f_i}{m}$$

where f_e is the frequency of item e in the window. By Markov inequality, we have

$$Pr\{F_{i,j,k} < \varepsilon\} = 1 - Pr\{F_{i,j,k} \ge \varepsilon\} \ge 1 - \frac{W - f_i}{m\varepsilon}$$

Therefore, according to the lemma above,

$$\begin{aligned} Pr\{f_i - \mathcal{A}_j[k].count \ge \varepsilon\} &= 1 - Pr\{f_i - \mathcal{A}_j[k].count < \varepsilon\} \\ &\le 1 - Pr\{f_i > F_{i,j,k} \wedge F_{i,j,k} < \varepsilon\} \\ &= 1 - Pr\{F_{i,j,k} < \varepsilon\} \\ &\le \frac{W - f_i}{m\varepsilon} \end{aligned}$$

□

4.2 Upper Bound of the Number of Distinct Items in Stage 2

THEOREM 4.3. *Given a data stream S. We assume each window has W items. In each window, S obeys an arbitrary distribution. Let n be the number of distinct items in Stage 2, H be the Running Track threshold. Then, we have*

$$n \le \frac{3W}{H} \tag{2}$$

PROOF. For an item, it is in Stage 2 either because it has already been in Stage 2 before this window or because it passes through Stage 1 in this window. We denote f_0 the frequency of the item in the current window, f_1 the frequency of the item in the previous window, f_2 the frequency of the item in the window before the previous window. In the case of the item that has already been in Stage 2, because of the cleaning policy, we have $f_1 \ge H \vee f_2 \ge H$. In another case, the item passes through Stage 1, which means $f_0 \ge H$. In summary, for an item in Stage 2, it satisfies $f_0 \ge H \vee f_1 \ge H \vee f_2 \ge H$. For each window, the number of items whose frequency is not less than the threshold is no more than $\frac{W}{H}$. We add up it and derive the upper bound $\frac{3W}{H}$. □

4.3 Proof of no Overestimation Error

THEOREM 4.4. *For any item e_i in Stage 2, let $\hat{f_i}$ be the estimated frequency of item e_i in Stage 2, f_i be the real frequency, then*

$$\hat{f_i} \le f_i$$

PROOF. For item e_i, if it has already been in Stage 2 before the current window, it is obvious that estimated frequency $\hat{f_i}$ is equal to the real frequency f_i. If it passes through Stage 1 in the current window, the frequency before being stored in Stage 2 should not be less than the Running Track threshold. Because we set the threshold as the initial value of $\hat{f_i}$, we have $\hat{f_i} \le f_i$. □

Corollary 4.5. *The arrival rates of output items in finding high-speed items are definitely higher than K.*

5 EXPERIMENTAL RESULTS

In this section, we show the experimental results of BurstSketch. First, we describe the experimental setup in Section 5.1. Second, we show how parameter settings affect BurstSketch's performance in Section 5.2. Third, we evaluate the performance of BurstSketch on different datasets and provide some analysis on BurstSketch in Section 5.4 and Section 5.5, respectively. Finally, we compare BurstSketch with prior works on burst detection and finding high-speed items in Section 5.6.

5.1 Experimental Setup

Datasets: We use the following datasets in our experiments and divide them into count-based windows and time-based windows.

1) IP Trace Dataset: As many papers [9, 24] do, we use anonymized IP trace streams from CAIDA [51]. CAIDA identifies each flow of IP trace streams by the five-tuples: source and destination IP address, source, and destination port, protocol. We use the source and destination IP address in the five-tuples as ID. We use 20M items. The number of bursts of this dataset is 19551 when we set the window size as 40K items. The duration in which the data was collected is 44.02s.

2) Web Page Dataset: The web page dataset is built from a collection of web pages, which were downloaded from a website [52]. Each item is 4 bytes long, representing the number of distinct items in a web page. We use 20M items. The number of bursts of this dataset is 6861 when we set the window size as 70K items.

3) Network Dataset: The network dataset contains users' posting history on the stack exchange website [53]. 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. We use 3M items. The number of bursts of this dataset is 989 when we set the window size as 70K items.

Implementation: BurstSketch and the strawman solution is implemented in C++. We run the programs on a server with dual 6-core CPUs (12 threads, Intel Xeon CPU E5-2620 @2.00 GHz) and 64GB DRAM memory. In all experiments, we use BOB Hash [54]to implement the hash functions.

Metrics:

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

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

3) F1 Score: $\frac{2 \cdot RR \cdot PR}{RR+PR}$. It is calculated from the precision and recall of the test, and it is also a measure of a test's accuracy.

4) Throughput: Million insertions per second (MIPS). We repeat the experiments 5 times and average the results.

5.2 Experiments on Parameter Settings

In this section, we measure the effects of some key parameters of BurstSketch, namely, the number of hash functions d, the ratio of the size of Stage 1 to the size of Stage 2 $\frac{md}{Ms}$, the number of cells in a bucket s, the ratio of the Running Track threshold to the burst threshold l, and the ratio between two adjoin windows for sudden increase or sudden decrease detection k in Stage 2. We also consider the replacement strategy in Stage 1 a parameter. We use the CAIDA dataset in these experiments, and RR and PR to evaluate the effects.

Effects of d (Figure 2(a)): *In the basic version, the best d is 1. In the optimized version, d = 6 is the best.* In this experiment, we fix the size of Stage 1 and Stage 2 to 2000. The number of hash functions d varies from 1 to 6. The results show that in the basic version, when $d = 1$, the RR is the highest. As d grows larger than 1, RR decreases evidently. In the optimized version, RR increases as d becomes larger, so the optimal d is 6. Thus, we set $d = 1$ for the basic version and $d = 6$ for the optimized version in the following experiments. Users can tune the parameter d to make a trade off between accuracy and speed depending on the application requirements. A larger value of d for the optimized version will slow down its speed because we have to check $d - 1$ more buckets for each insertion (Figure 4(e)). In other words, increasing d from 1 to the optimal value means higher accuracy but will lower speed.

Effects of $\frac{md}{Ms}$ (Figure 2(b)): *The experimental results show that the best value for $\frac{md}{Ms}$ is from 2.25 to 3.25.* In this experiment, we set the total memory size to 3 different values: 40KB, 60KB, and 80KB and vary $\frac{md}{Ms}$ from 1.25 to 3.75. The results show that when memory size is 60KB or 80KB, RR increases as $\frac{md}{Ms}$ increase. When memory size is 40KB, RR peaks while $\frac{md}{Ms} = 2.25$. Therefore, the optimal value of $\frac{md}{Ms}$ is from 2.25 to 3.25, and we choose $\frac{md}{Ms} = 3.25$.

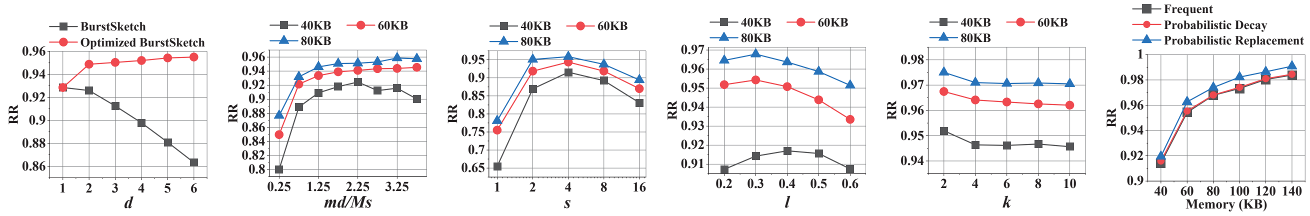
Effects of s (Figure 2(c)): *The experimental results show that BurstSketch achieves the best accuracy when the number of cells in a bucket is 4.* We compare the effects of different values of s and find that 4 is always the optimal value for s in 3 different memory cases, as shown in the figure. So we set s to 4 in our experiments.

Effects of l (Figure 2(d)): *The experimental results show that the optimal value for l is from 0.3 to 0.4.* In this experiment, we compare the performance of BurstSketch when l varies from 0.2 to 0.6. When the memory size is 40KB, PR peaks when $l = 0.4$. When the memory size is 60/80KB, PR reaches the peak point while $l = 0.3$. Thus, the optimal value of l is from 0.3 to 0.4, and we set $l = 0.3$.

Effects of the ratio k (Figure 2(e)): *Our experimental results show that BurstSketch performs well even when the ratio k is very high.* As the ratio k varies, the RR of BurstSketch is stable, which indicates that the performance of BurstSketch is insensitive to k. For simplicity, we set k to 2 in our experiments in this paper.

Effects of replacement strategy (Figure 2(f)): *Our experimental results show that the RR of BurstSketch under the three replacement strategies are close.* Although the RR of probabilistic replacement is often slightly higher, it is slow, complex, and unstable, while Frequent is fast and easy to implement. Therefore, we choose frequent as the replacement strategy in this paper.

Analysis: The optimal value of d in the basic version is small because higher d results in more copies of high-speed items, which is memory consuming. This is consistent with our analysis in Section 3.3. After the deduplication, the value of d of the optimized version is larger because each potential burst item has more opportunities to be selected into Stage 2. For $\frac{md}{Ms}$, as md becomes larger, hash collisions are reduced. If Ms is larger, more potential burst items can be monitored at the same time. Therefore, the optimal ratio balances two stages. For l, if it is smaller, items in Stage 1 is easier to be inserted into Stage 2, so that the arrival rate of the item will be more accurate. However, as l grows larger, the number of items monitored in Stage 2 grows larger. making Stage 2 easier to be full. Therefore, the optimal ratio balances these two situations.



(a) d (b) $\frac{md}{Ms}$ (c) s (d) l (e) k (f) replacement strategy

Figure 2: Evaluation on Parameter Settings.

Concrete Steps for Choosing Parameters: For parameter d in the basic version, we find that $d = 1$ is optimal in most cases. For parameter d in the optimized version, increasing d will increase accuracy and decrease speed. Therefore, users can adjust d to strike a good trade off between accuracy and speed. For parameter $\frac{md}{Ms}$, the optimal $\frac{md}{Ms}$ is always large than 0.75 in general. Therefore, we can try increasing $\frac{md}{Ms}$ to find the optimal $\frac{md}{Ms}$. For parameter s, the optimal s is always in the range of $2 - 16$ in general. Therefore, we can try setting s from 2 to 16 to find the optimal s. For parameter l, the optimal l is always in the range of $0.2 - 0.6$ in general. Therefore, we can try setting l from 0.2 to 0.6 to find the optimal value.

5.3 Experiments on Different Datasets

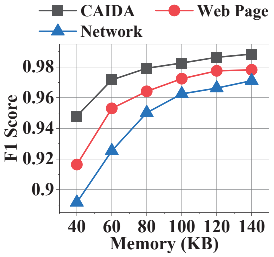


Figure 3: Burst Detection on Different Datasets.

In this section, we conduct experiments on three real-world datasets: CAIDA, Web Page, and Network, and evaluate BurstSketch's performance with F1 score. The F1 score exceeds 94% with 20KB of memory, which indicates that BurstSketch works well with very limited memory. When the memory size exceeds 100KB, the F1 score on three datasets all exceeds 97%. In addition, the reason why BurstSketch performs best in CAIDA dataset is that the skewness of CAIDA is the highest. It means that it is easier to filter out non-burst item in Stage 1 in CAIDA. *The results show that BurstSketch achieves high F1 score on every datasets. The performance of BurstSketch in three datasets are only slightly different, and the trends are very similar.* The results show the robustness of BurstSketch, so in the following experiments, we only use CAIDA dataset.

5.4 Comparison with Strawman Solution

In this section, we compare BurstSketch's performance with the strawman solution and the optimized version in the metrics below.

RR (Figure 4(a)): *This experiment shows that RR of the optimized BurstSketch is slightly higher than BurstSketch, and the RR of BurstSketch greatly outperforms the strawman solution.* Compared to the strawman solution, RR of BurstSketch is about 25% higher in average. The optimized BurstSketch improve the RR by about 2% compared to the basic version.

PR (Figure 4(b)): *This experiment shows that the PR of the optimized BurstSketch is higher than the basic version and is much higher than the strawman solution.* The results show that BurstSketch's PR is about 40% higher than the strawman solution. The PR of the optimized version is 0.1% higher than the basic version.

Throughput (Figure 4(c)): *Our results show that the insertion throughput of the BurstSketch is always higher than that of the strawman solution.* The throughput of optimized BurstSketch is 3.2 times higher than that of the strawman solution. The throughput of the basic version is 1.34 times higher than that of the optimized version ($d = 6$). However, if the optimized version's d is smaller, the throughput is higher. It also shows that decreasing d is a trade off between accuracy and speed.

Analysis: The experiment results show that BurstSketch greatly outperforms the strawman solution. The results are consistent with our analysis in Section 3.2. The main reason is that the strawman solution stores frequencies of all items in $n + 2$ time windows (n is the max width of a burst) to detect bursts, which have enormous redundancy. In contrast, first, BurstSketch uses Running Track to filter out infrequent items and frequent item with a steady arrival rate, which are not potential bursts. Second, BurstSketch uses snapshotting to snapshot two key feathers of a burst: sudden increase and sudden decrease, to detect bursts from the potential burst items.

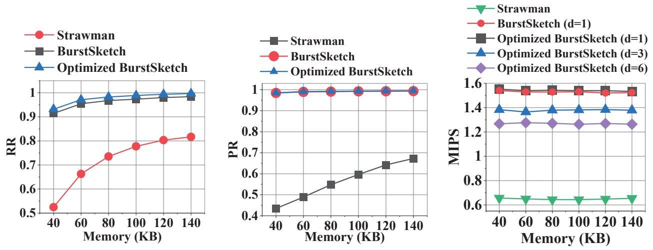
5.5 Analysis on BurstSketch

In this section, we analyse BurstSketch from several aspects. First, we compare its performance in time-based windows and count-based windows. To show Stage 1's effectiveness, we measure the number of data streams that pass through Stage 1. Also, we evaluate the minimal memory usage to achieve an acceptable performance in data streams of different speeds. Finally, we test BurstSketch's performance in detecting bursts inside bursts.

Performance under time-based and count-based windows (Figure 5(a)): Different from count-based windows, the number of items per window could vary a lot in time-based windows. The experimental results show that BurstSketch's performance under count-based windows is slightly higher than its performance under time-based windows. This reveals that the accuracy of our BurstSketch is insensitive to whether the number of items in each window is equal. The reason behind is that, no matter whether the number of items in each window is equal, after the items are filtered by Stage 1, the number of items (potential bursts) that reach Stage 2 varies a lot per window.

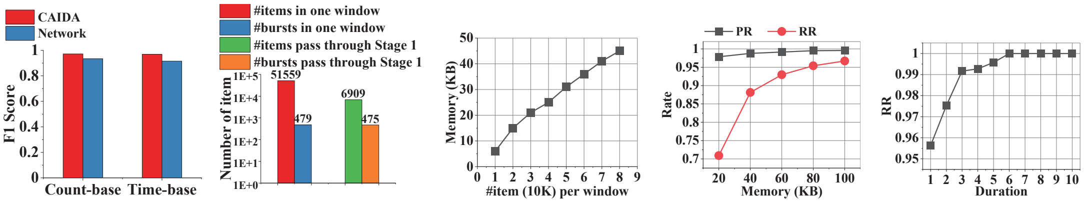
The number of items that pass through stage 1 (Figure 5(b)): The experimental results show that Stage 1 is highly effective in filtering out non-burst items, since about 87% of the items in the data stream are filtered out. Only 4 (0.8%) items that are filtered out are bursts, which shows that Stage 1 has a very high recall rate.

Memory usage in burst detection in data streams of different speed(Figure 5(c)): In this experiment, we vary the speed of the input data stream (from 10K items to 90K items per window), and



(a) RR (b) PR (c) Throughput

Figure 4: Comparison between BurstSketch and Strawman Solution and The Optimized Version



(a) window type (b) effectiveness of Stage 1 (c) memory usage (d) bursts inside bursts (e) bursts' duration

Figure 5: Analyses on BurstSketch.

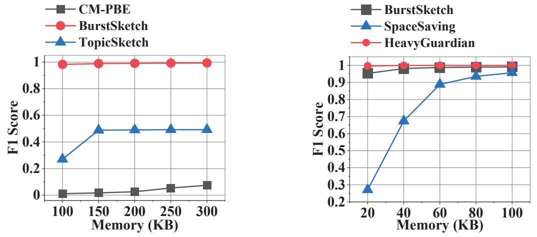
check how much memory BurstSketch has to use to achieve an F1 score of 0.9. The experimental results show that the memory usage to achieve an F1 score of 0.9 grows linearly with the increase of the speed of the data stream.

Bursts inside bursts (Figure 5(d)): The results show that BurstSketch performs well in detecting bursts inside bursts. The PR in finding bursts inside bursts exceeds 97% with 20KB memory. The RR is 70% with 20KB memory but grows rapidly.

The influence of the duration of bursts(Figure 5(e)): The experimental results reveal that as the duration of burst grows larger, the RR of BurstSketch increases. The reason is that the streams with larger duration tend to be stable, and our algorithm detects this kind of bursts more effectively.

5.6 Comparison with Prior Work

In this section, we compare BurstSketch with prior works in burst detection and finding high-speed items. In burst detection, we compare it with CM-PBE-1 [6] and TopicSketch [3] using the ground truth by our definition. In finding high-speed items, we compare it with HeavyGuardian [9] and SpaceSaving [25].



(a) burst detection (b) finding high-speed items

Figure 6: Comparison with Prior Works.

Comparison on Burst Detection (Figure 6(a)): *The experimental results show that BurstSketch largely outperforms TopicSketch and CM-PBE.* BurstSketch achieves an F1 score very close to 1, while TopicSketch's F1 score failed to exceed 0.5 with the memory usage of 300KB, and the F1 score achieved by CM-PBE failed to exceed 0.08 with the memory usage of 300KB. As mentioned above, our definition is different from others, which means the applications are also different. Therefore, the experimental comparison of different definitions does not mean that the performance of BurstSketch is much better than other burst algorithms. It means that other algorithms are inappropriate for our applications.

F1 for finding high-speed items (Figure 6(b)): *This experiment shows that BurstSketch achieves high F1 score in finding high-speed items.* The F1 score of BurstSketch reaches 0.95 even if the memory size is only 20KB. As the memory size exceeds 40KB, the F1 score of BurstSketch is very close to 1. The F1 score of BurstSketch is a little lower than that of HeavyGuardian, but they are very close. The F1 score of BurstSketch is averagely 1.32 times higher than SpaceSaving, and is 3.5 times higher under the memory size of 20KB. The results show that BurstSketch performs well in finding high-speed items, which is consistent with our analysis in Section 3.2. In fact, Running Track is designed to filter out slow-speed items and select high-speed items. Therefore, BurstSketch is efficient in finding high-speed items.

6 CONCLUSION

Real-time burst detection in high-speed data streams is important in many applications. This paper proposes a novel algorithm called BurstSketch for real-time burst detection, which is fast, memory-efficient, and accurate. Experimental results show that the BurstSketch can achieve high accuracy with fairly limited memory usage in real-time burst detection and finding high-speed items.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China (NSFC) (No. U20A20179), and the project of "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications" (No. LZC0019).

REFERENCES

[1] Jon Kleinberg. Bursty and hierarchical structure in streams. *KDD*, 2003.

[2] Zhijian Yuan, Yan Jia, and Shuqiang Yang. Online burst detection over high speed short text streams. In *ICCS*, 2007.

[3] Wei Xie, Feida Zhu, Jing Jiang, Ee-Peng Lim, and Ke Wang. Topicsketch: Real-time bursty topic detection from twitter. *TKDE*, 2016.

[4] Ryohei Ebina, Kenji Nakamura, and Shigeru Oyanagi. A real-time burst detection method. In *ICTAI*, 2011.

[5] Chuangying Zhu, Junping Du, Qiang Zhang, Ziwen Zhu, and Lei Shi. Burst topic detection in real time spatial–temporal data stream. *IEEE Access*, 2019.

[6] Debjyoti Paul, Yanqing Peng, and Feifei Li. Bursty event detection throughout histories. In *ICDE*, 2019.

[7] Source code related to BurstSketch. https://github.com/BurstSketch/BurstSketch.

[8] G. Lukasz, D. David, D. Erik D, L. Alejandro, and M. J Ian. Identifying frequent items in sliding windows over on-line packet streams. In *Proc. ACM IMC*, 2003.

[9] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. Heavyguardian: Separate and guard hot items in data streams. In *KDD*, 2018.

[10] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Randomized admission policy for efficient top-k and frequency estimation. In *INFOCOM*, 2017.

[11] Yunyue Zhu and Dennis Shasha. Efficient elastic burst detection in data streams. In *SIGKDD*, 2003.

[12] Rafal Maison and Maciej Zakrzewicz. Prediction-based load shedding for burst data streams. *Bell Labs Technical Journal*, 2011.

[13] Xin Zhang and Dennis Shasha. Better burst detection. In *ICDE*, 2006.

[14] Shouke Qin, Weining Qian, and Aoying Zhou. Approximately processing multi-granularity aggregate queries over data streams. In *ICDE*, 2006.

[15] Tingting Chen, Yi Wang, Binxing Fang, and Jun Zheng. Detecting lasting and abrupt bursts in data streams using two-layered wavelet tree. In *AICT-ICIW*, 2006.

[16] Aoying Zhou, Shouke Qin, and Weining Qian. Adaptively detecting aggregation bursts in data streams. In *DASFAA*, 2005.

[17] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 2005.

[18] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. *ACM SIGMCOMM CCR*, 2002.

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

[20] Tong Yang, Yang Zhou, Hao Jin, Shigang Chen, and Xiaoming Li. Pyramid sketch: A sketch framework for frequency estimation of data streams. *Proc. VLDB Endow.*, 2017.

[21] Anshumali Shrivastava, Arnd Christian Konig, and Mikhail Bilenko. Time adaptive sketches (ada-sketches) for summarizing data streams. In *SIGMOD*, 2016.

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

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

[24] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: adaptive and fast network-wide measurements. In *SIGCOMM*, 2018.

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

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

[27] Graham Cormode and Shanmugavelayutham Muthukrishnan. What's new: Finding significant differences in network data streams. *IEEE/ACM Transactions on Networking*, 2005.

[28] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: a better netflow for data centers. In *Proc. USENIX NSDI*, 2016.

[29] Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter A Dinda, Ming-Yang Kao, and Gokhan Memik. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Transactions on Networking*, 2007.

[30] K. Balachander, S. Subhabrata, Z. Yin, and C. Yan. Sketch-based change detection: methods, evaluation, and applications. In *SIGCOMM*, 2003.

[31] Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *SIGMOD*, 2016.

[32] Kai Sheng Tai, Vatsal Sharan, Peter Bailis, and Gregory Valiant. Sketching linear classifiers over data streams. In *SIGMOD*, 2018.

[33] Graham Cormode. Sketch techniques for approximate query processing. *TRDB*, 2011.

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

[35] Tong Yang, Alex X Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. A shifting bloom filter framework for set queries. *VLDB Endowment*, 2016.

[36] Bofang Li, Aleksandr Drozd, and et al. Scaling word2vec on big corpus. *DSE*, 2019.

[37] Stephen Bonner, Ibad Kureshi, and et al. Exploring the semantic content of unsupervised graph embeddings: An empirical study. *DSE*, 2019.

[38] Yinghui Wang, Peng Lin, and Yiguang Hong. Distributed regression estimation with incomplete data in multi-agent networks. *Science China Information Sciences*, 2018.

[39] Tongya Zheng, Gang Chen, and et al. Real-time intelligent big data processing: technology, platform, and applications. *Science China Information Sciences*, 2019.

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

[41] Haipeng Dai, Muhammad Shahzad, Alex X Liu, and Yuankun Zhong. Finding persistent items in data streams. *VLDB Endowment*, 2016.

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

[43] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *SIGMOD*, 2017.

[44] Alex D. Breslow and Nuwan S. Jayasena. Morton filters: Faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proc. VLDB Endow.*, 2018.

[45] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters. *Proc. VLDB Endow.*, 2019.

[46] Jiawei Jiang, Fangcheng Fu, Tong Yang, and Bin Cui. Sketchml: Accelerating distributed machine learning with data sketches. In *SIGMOD*, 2018.

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

[48] Zhewei Wei, Ge Luo, Ke Yi, Xiaoyong Du, and Ji-Rong Wen. Persistent data sketching. In *SIGMOD*, 2015.

[49] Yikai Zhao, Liu Zirui, Tong Yang, and etal. Lightguardian: A full-visibility, lightweight, in-band telemetry system using sketchlets. In *NSDI*, 2021.

[50] Peiqing Chen, Dong Chen, Lingxiao Zheng, and etal. Out of manywe are one: Measuring item batch with clock-sketch. In *SIGMOD*, 2021.

[51] The caida anonymized 2016 internet traces. http://www.caida.org/data/overview/.

[52] Real-life transactional dataset. http://fimi.ua.ac.be/data/.

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

[54] Hash website. http://burtleburtle.net/bob/hash/evahash.html.