Counting Distinct Objects over Sliding Windows

Wenjie Zhang [1,2] Ying Zhang [1] Muhammad Aamir Cheema[1] Xuemin Lin [1,2]

[1] the University of New South Wales {xxx,lxue,zzz}@cse.unsw.edu.au [2] NICTA

Abstract

Aggregation against distinct objects has been involved in many real applications with the presence of duplicates, including real-time monitoring moving objects. In this paper, we investigate the problem of counting distinct objects over sliding windows with arbitrary lengths. We present novel, time and space efficient, one scan algorithms to continuously maintain a sketch so that the counting can be approximately conducted with a relative error guarantee ϵ in the presence of object duplicates. Efficient query algorithms have also been developed by effectively utilizing the skyband property. Moreover, the proposed techniques may be immediately applied to the range counting aggregation and heavy hitter problem against distinct elements. A comprehensive performance study demonstrates that our algorithms can support real-time computation against high speed data streams.

1 Introduction

Many recent applications in real-time monitoring moving objects require on-line counting of distinct objects. For instance, in real-time traffic management it is desirable to monitor the traffic volume of an area over a time frame; this is usually done by counting the number of distinct objects. In wireless communication management, the number of distinct users at a station is a key measurement of "popularity" of the area covered by the station. Counting distinct objects is also required in many other applications. For instance, in a stock market surveillance system, it is important to monitor the number of distinct clients in real-time in addition to the total number of transactions, the prices, etc. Moreover, counting distinct objects is a key component in an estimation of join results in join processing optimization; consequently on-line counting distinct objects may be applied to the join processing optimization among data streams.

In the above applications, datasets may be massive in size and fast in update speed. Therefore, in the context of real-time monitoring regarding the applications above, it is desirable to read database only once (i.e., one scan). Nevertheless, the challenge is that it is impossible to count the number of distinct elements by only one-scan of a dataset unless the whole dataset fits in memory. This makes it impractical to exactly counting distinct objects in real-time.

Flajolet and Martin [7] provide the first one-scan technique to build a sketch so that approximately counting distinct objects has the precision guarantee $\frac{|n'-n|}{n} \leq \epsilon$ (ϵ-*approximation*) with the confidence $1-\delta$ ($\delta > 0$) and the space of $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\log m)$ bits where n' is an approximation of n distinct elements, and m is the object domain size. Bar-Yossef *et. al.* in [1] improves the efficiency of the technique of [7].

While the techniques cited above have been shown very space and time efficient to achieve ϵ-approximation, they do not deal with the concept of *aging*. They are not applicable to *sliding windows*. On the other hand, there are a broad spectrum of applications where data objects observed early could be outdated and counting the most recently observed (sliding windows) distinct data elements is more important. For instance, in a real-time traffic management, counting distinct vehicles over the most recent traffic may provide more accurate prediction towards traffic volume changes than that over all observed vehicle movements as traffic changes from time to time. Similarly, sliding windows are also crucial in the other two applications above.

Datar *et. al.* [5] provide the technique of *exponential histogram* to approximately counting the number of objects over sliding windows with variable lengths. It requires the space $O(\frac{1}{\epsilon}\log w)$ to enforce ϵ-approximation where w is the maximum number of objects encountered by a sliding window. While space and time efficient, the technique is not applicable to counting distinct elements.

An efficient off-line technique is developed by Tao *et. al.* in [16] to build space effective techniques to approximately count distinct objects against time intervals. In order to accommodate any time interval, it requires to create a sketch at each time-stamp which has a different sketch than the last sketch. Consequently, a huge storage space may still be required if there are massive sketch changes; thus, the techniques are applicable only to off-line computation against historical data. Gibbon [8] develops a novel one-scan *distinct sampling technique* to estimate the result size of an arbitrary query. Applying it to counting distinct elements over sliding windows requires a pre-fixed sample size of $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\sqrt{m})$ to enforce ϵ-approximation with $(1-\delta)$ confidence, according to Theorem in [8].

Motivated by this, in this paper we present novel, time and space efficient, one-scan algorithms to continuously maintain sketches to approximately count the number of distinct objects against sliding windows of arbitrary lengths with ϵ-approximation. Our contributions may be summarized as follows:

1. We develop novel, one-scan, space and time efficient sketch techniques, based on FM algorithms [7]. Our techniques guarantee ϵ-approximation by $1-\delta$ confidence with the space of $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\log m)$ machine words; this greatly reduces the space requirement in [8]. Consider that approximately counting distinct elements over sliding windows is much more sophisticated than that over whole streams. The space requirement of our techniques is near optimal as even counting a whole stream needs the space of $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\log m)$ bits [1, 7] to ensure ϵ-approximation with $1-\delta$ confidence.
2. The sketch technique above has $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$ subsketches and a query algorithm has to process every sub-sketch. Consequently, our query algorithm, against FM-based sketches above runs in $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\log\log m)$ time. To speed up query processing time, we develop another time and space efficient sketch techniques based on the algorithm in [1]. To enforce ϵ-approximation with $1-\delta$ confidence, our second sketch technique only requires $O(\log\frac{1}{\delta})$ subsketches. By converting the sketch problem to the "k-skyband" problem [14], novel and efficient techniques have been developed to continuously maintain k-skyband in a 2-d space when k is large. By effectively utilizing the property of a skyband, we show that our query algorithm runs in time $O(\log\frac{1}{\delta}\log M)$ where M, the maximum size of a subsketch, is (expected) $O(\frac{1}{\epsilon^2}\log n)$. Consequently, our second query algorithm runs in (expected) $O(\log\frac{1}{\delta}(\log\frac{1}{\epsilon}+\log\log n))$ time.

A comprehensive performance study demonstrates that our techniques can approximately counting distinct objects in real-time over high speed data streams (updates) with high accuracy and a small space requirement.

The rest of the paper is organised as follows. Section 2 presents problem definitions and related work. In Section 3, we present our first contribution of the paper. Section 4 presents the second contribution. In Section 5, we report our experiment results. Section 6 concludes the paper.

2 Background Information

We first state the problem. Then we present the related work.

2.1 Problem Statement

We model a set of observations by data streams. Each observation is treated as an element in a data stream and is represented by $e = (x, t)$ where x is an object ID and t is the time-stamp. Note that for two data elements $e = (x, t)$ and $e' = (x', t')$, $t > t'$ iff e comes later than e'; that is, e is *younger* than e'. Although $e \neq e'$, x could be the same as x'; that is, the same object is observed twice at t and t', respectively.

In a collection S of data elements, there may be many elements with the same object ID (e.g., objects are moving around and observed multiple times). D_S denotes the set of distinct object IDs in S. In this paper, we study the following counting problem.

Distinct Counting against Sliding Window(DCSW). For a given t, let S_t denote the data elements observed after (inclusive) t and $D_{S,t}$ denote the set of distinct object IDs contained in S_t. We compute $|D_{S,t}|$ for any t and denote $|D_{S,t}|$ by $n_{S,t}$.

We investigate the problem of answering DCSW queries with the precision guarantee of ϵ-*approximation*; that is, to enforce $\frac{|n'-n|}{n} \leq \epsilon$ where n' is an approximate solution of n.

Problem Description. We investigate the problem of continuously maintaining a sketch (consisting of several sub-sketches) over a data stream S such that for any given t, the sketch can be used to return an ϵ-approximate answer to $n_{S,t}$. The aim is to minimize the *maximum memory space required in such a continuous computation*, as well as to process high-speed data streams in real time.

2.2 Related Work

We briefly overview the techniques in [1, 7, 16]. They are most closely related to our work.

2.2.1 FM Algorithm

Suppose that S is a collection of elements whose domain is $\mathcal{D}$. FM algorithm [7] proceeds as follows.

Let B be a bitmap of length k with subindexes $[0, k-1]$. Suppose that $h()$ is a randomly generated hash function $\mathcal{D} \to B$, such that $\forall x \in \mathcal{D}$, 1) for each bit, $h(x)$ has the equal opportunity to have 0 or 1, 2) $h(x)$ is enforced to have one and only one bit with value 1, and 3) $h(x)$ assigns the last bit (the bit with subindex $k-1$) with value 1 iff the first $k-1$ bits (from left) take value 0. To enforce property 2), $h(x)$ may be interpreted as a serial binary hash functions that start from the first bit and terminate once the current bit is assigned by value 1. It can be immediately shown [2] that on average, $h()$ runs in time $O(1)$ (two calls of a binary hash function) per data element and the probability of having the ith bit with value 1 is $\frac{1}{2^{i+1}}$. In our implementation, we use the public code from Massive Data Analysis Lab [11] to randomly generate such hash functions.

A FM sketch on S is defined as $FM(S) = \bigvee_{x \in S} h(x)$, where $FM(S)$ is a bitmap with length k and the ith bit of $FM(S)$ takes value 1 iff $\exists x \in S$ such that $h(x)$ assigns the value 1 to the ith bit. We define $FM_{min}(S)$ as follows:

- If i is the least bit (from left) with value 0, $FM_{min}(S)$ is defined as i.
- Otherwise, $FM_{min}(S)$ is defined as ∞ (in our implementation, we define $FM_{min}(S)$ as k).

To improve the accuracy of FM algorithm, multiple copies (say, l) of FM sketches are constructed. Therefore, each data element is hashed into l FM sketches, $FM_1(S)$, $FM_2(S)$, ... , $FM_l(S)$, respectively. The number n_S of distinct elements in S is estimated by:

$$A_S = \frac{1}{\varphi} 2^{\sum_{i=1}^{l} FM_{i,min}(S)/l}. \qquad (1)$$

Here, $\varphi \overset{\text{def}}{=} 2^{E(FM_{1,min}(S))}/n_S$,[1] and each $FM_{i,min}(S)$ related to $FM_i(S)$ is defined in the same way as $FM_{min}(S)$ related to $FM(S)$. As shown in [7], $E(FM_{i,min}(S)) = E(FM_{j,min}(S))$ $(1 \leq i < j \leq l)$. From the insight in Section 3.2 in [4], Theorem 2 in [7], and the *Central Limit Theorem* (pp 229 in [6]), the following lemma can be immediately verified using the independence assumption.

Lemma 1 *Suppose that A_S is returned by FM algorithm as shown in (1). Then, $P(|A_S - n_S| < \epsilon n_S) \geq 1 - \delta$, for any given $0 < \delta < 1$ and $0 < \epsilon < 1$, if $n_S > \frac{1}{\epsilon}$, $k = O(\log m + \log \epsilon^{-1} + \log \delta^{-1})$, and $l = O(\frac{1}{\epsilon^2} \log \delta^{-1})$, where $m = |\mathcal{D}|$.*

2.2.2 BJKST algorithm

In [1], a novel variation of FM algorithm, BJKST algorithm, has been proposed to speed-up the computation, while the accuracy and the space-efficiency can be retained. It proceeds as follows. First, we pick at random a pairwise independent hash function h to hash $\mathcal{D}$ to $[1, m^3]$ where $\mathcal{D}$ is the domain of data elements x and $|\mathcal{D}| = m$. The following Lemma has been shown as folklore.

Lemma 2 *If $m \geq \delta^{-1}$ then h is injective over S with probability at least $1 - \delta$.*

Based on this, BJKST algorithm always keeps the k smallest elements (i.e. with the k smallest distinct hash values) and uses the following A_S to estimate n_S

$$A_S = \frac{k \times m^3}{f_{k\_\min}}. \qquad (2)$$

Here, $f_{k\_\min}$ is the kth smallest distinct hash value. If there are less than k distinct values, then $A_S = \infty$ (in our implementation, we assign A_S as k' ($k' \leq k$) if there are only k' distinct values). To improve the accuracy, BJKST algorithm picks at random l pairwise independent hash functions h_i (hashing $\mathcal{D}$ to $[1, m^3]$), and outputs $A_{i,S}$ for each h_i where $A_{i,S}$ (for $1 \leq i \leq l$) related to h_i is defined in the same way as A_S related to h. BJKST algorithm outputs A_S as the median of these $A_{i,S}$ to estimate n_S. BJKST algorithm keeps only k elements with the k smallest distinct hash values. The following Lemma 3 has been proved in [1].

Lemma 3 *Suppose that $0 < \epsilon, \delta < 1$. If $m \geq \delta^{-1}$, $k = O(\frac{1}{\epsilon^2})$, $l = O(\log \delta^{-1})$, and $n_S \geq k$, then $P(|A_S - n_S| < \epsilon n_S) \geq 1 - \delta$.*

2.2.3 Spatio-Temporal Aggregation

Regarding counting distinct spatial objects intersecting a spatial region, Tao *et. al.* [16] observe that the space required to exactly count distinct spatial objects intersecting a spatial region is $\Omega(N)$ where N is the number of observations. To reduce space, they proposed to use FM algorithm to build space-efficient sketches to approximately conduct the computation.

To support an arbitrary time interval, an FM sketch is constructed against a batch of new observations with the same time stamp. A sketch at time-stamp t_2 is kept if it is different than that generated at time t_1 where t_1 is largest time-stamp but smaller than t_2 (i.e. t_1 and t_2 are consecutive). Consequently, $O(\mathcal{N})$ FM sketches are required to be maintained with the total space $O(\mathcal{N} \frac{1}{\epsilon^2} \log \frac{1}{\delta} \log m)$

[1] As $E(FM_{1,min}(S))$ cannot be explicitly represented and n_S is unknown, in our implementation we approximately choose φ as 0.775351 according to the approximate results in [7].

to guarantee ϵ-approximation where $\mathcal{N}$ is the number of observation batches. To support efficient off-line computation, an aRB like [13] index is developed in [16]. Clearly, it is quite space efficient when $\mathcal{N} \ll N$. Nevertheless, the technique is not applicable to data streams where $\mathcal{N}$ often equals $O(N)$.

3 FM-based Sketches

Our techniques are based on the following observation. For a dataset S, if we first select the data elements from S with time-stamp values not smaller than a given t (the result is denoted by $S|_{t+}$) and apply FM Algorithm on $S|_{t+}$, then the obtained estimation $A_{S,t}$ of the number $n_{S,t}$ of distinct objects in $S|_{t+}$ follows Lemma 1.

Below, we first present a novel, space-efficient data structure (sketch) by using FM algorithm. Then, we present space- and time-efficient algorithms to continuously maintain such sketches to achieve an ϵ-approximation.

3.1 The Framework

The following example illustrates the basic idea in our framework based on FM algorithm.

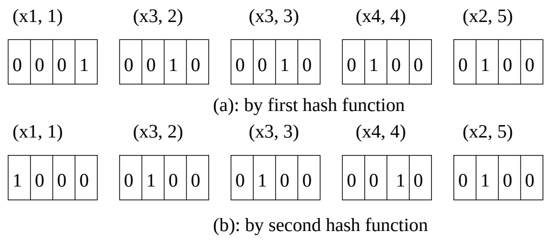


Figure 1: An Example

As shown in Figure 1, there are 5 data elements (x, t) where the second and the third represent two occurrences of the same object x_3. Suppose that in FM algorithm $l = 2$ and $k = 4$. Two hash functions h_1 and h_2 are randomly picked to hash each element, respectively. The total 10 bitmaps with length 4 are generated, respectively, by h_1 and h_2, as depicted in Figure 1(a)-(b).

To effectively keep values information, we map a bitmap into an array by replacing the bit with value 1 by its corresponding time-stamp. Figure 2 shows the corresponding arrays converted from the bitmaps in Figure 1.

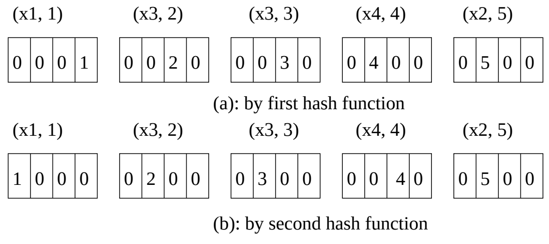


Figure 2: Transformed from Figure 1

For a time t and a hash function j (jth subsketch), to estimate $n_{S,t}$ (number of distinct elements arriving no earlier than t) by using FM algorithm we first select the arrays generated by h_j such that their corresponding non-zero values not smaller than t, then find the left-most common element with value 0 and return its subindex as $f_{j,t}$.

Example 1 *Let $t = 2$ and $j = 1$. Regarding Figure 2 (a), the 2nd array, 3rd array, 4th array, and 5th array are selected. Then, $f_{1,2} = 0$ (i.e. the subindex of the left-most common element, 1st element, in these arrays). Here, the 2nd and 4th arrays are redundant.*

Clearly, computing $f_{j,t}$, by this way, is equivalent to what have been discussed in the beginning of this section; that is, we do a selection on S to output $S|_{t+}$; then apply h_j on $S|_{t+}$ and use FM Algorithm to get $FM_{j,min}(S|_{t+})$ ($= f_{j,t}$). Moreover, this example also demonstrates that if two arrays have non-zero values allocated in the same position, the one with smaller time-stamp value (older) will never be used in any query (i.e., regarding any t); consequently, this redundant array should be removed. Therefore, in the worst case we only keep k arrays where k is the length of bitmaps in the hash functions. Furthermore, after removing redundant arrays the remaining arrays generated by h_j can be merged into one array with non-zero values remain in the same positions, respectively.

Example 2 *Regarding the example in Figure 2(a), 2nd and 4th arrays are redundant and thus, are removed. The merged result is depicted in Figure 3(a). For the example in Figure 2(b), 2nd and 3rd arrays are redundant. The merged result is depicted in Figure 3(b).*

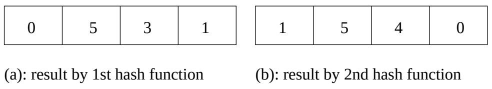

0 5 3 1
1 5 4 0
(a): result by 1st hash function
(b): result by 2nd hash function

Figure 3: Compressed from Figure 2

Below, we present our continuous sketch construction and maintenance algorithm in Algorithm 1. We maintain l arrays $\{s_i : 1 \le i \le l\}$ each of which is generated, as described above, by a randomly picked hash function h_i, and has k elements with subindexes from 0 to $k-1$. Note that the time-stamp of an element takes a positive value. Thus, each array s_i can be initialized to $(0, 0, ..., 0)$. For every $h_i(x)$ $(1 \le i \le l)$, $\rho(h_i(x))$ denotes the position (subindex) of the bit, with value 1, in $h_i(x)$. Note that $s_i[\rho]$ is the ρ-th element in s_i. Moreover, to ensure ϵ-approximations for sliding windows with the number of distinct objects less than $\frac{1}{\epsilon}$ precise answers are the only possibility; consequently, we always keep the L most recent "distinct" objects (i.e., with the largest time-stamp values) in $\mathcal{L}$ in addition to $\{s_i : 1 \le i \le l\}$, so that counting the number of most recent distinct objects, which is smaller than L, can be answered exactly. We use t_{sml} to denote the smallest time-stamp value in $\mathcal{L}$ and x_{sml} is the element with the time-stamp. Note that in $\mathcal{L}$ we keep each element (x, t).The following theorem is immediate.

Algorithm 1 Space-Efficient Sketches (**SE-FM**)

```
Input: $l$, $k$, $L$, a stream $S$ of $(x, t)$.
Output: $\mathcal{L}$: the set of $L$ most recent distinct objects;
    $\{s_i : 1 \le i \le l\}$: each $s_i$ is an array with $k$ elements.
Description:
 1: Initialize $\{s_i : 1 \le i \le l\}$; $\mathcal{L} \leftarrow \emptyset$; $j \leftarrow 0$;
 2: Generate $l$ hash functions $\{h_i() : 1 \le i \le l\}$;
 3: for each new $(x, t)$ do
 4:   if $\exists (x_1, t_1) \in \mathcal{L}$ s.t. $x_1 = x$ then
 5:     replace $(x_1, t_1)$ by $(x, t)$;
 6:   else
 7:     if $j < L$ then $\mathcal{L} \leftarrow \mathcal{L} \cup \{(x, t)\}$; $j \leftarrow j + 1$;
 8:     else remove $(x_{sml}, t_{sml})$ and add $(x, t)$ in $\mathcal{L}$;
 9:   for $i = 1$ to $l$ do
10:     $\rho \leftarrow \rho(h_i(x))$;
11:     $s_i[\rho] \leftarrow t$;
12: Return $\mathcal{L}$ & $\{s_i : 1 \le i \le l\}$.
```

Theorem 1 *Algorithm 1 requires the space of $L + l \times k$ elements.*

To estimate $n_{S,t}$ for a given t, our query algorithm proceeds as follows. If $t > t_{sml}$ then we only query $\mathcal{L}$. Otherwise, in the light of earlier discussions we first select the elements in s_i with positive time-stamps (corresponding to objects in $S|_{t+}$) not smaller than t; the result is denoted by $s_i|_{t+}$. Then, we return the location of the left-most element in s_i that is not included in $s_i|_{t+}$. If such a left-most element does not exist, we return k (corresponding to the situation ∞ when we presented FM Algorithm). Let Π denote a subset of elements in an array and $I(\Pi)$ denote the set of subindexes of the elements in Π. Our query algorithm is presented in Algorithm 2.

Algorithm 2 Approximating $n_{S,t}$

```
Input: $t$, $\mathcal{L}$, $\{s_i : 1 \leq i \leq l\}$ generated by Algorithm 1;
Output: $A_{S,t}$;
Description:
1: get $t_{sml}$ from $\mathcal{L}$;
2: if $t_{sml} < t$ then
3:     $A_{S,t} \leftarrow |\mathcal{L}|_{t^+}|$;
4: else
5:     for $i = 1$ to $l$ do
6:         if $[0, k-1] - I(s_i|_{t^+}) \neq \emptyset$ then
7:             $f_{i,t} \leftarrow \min\{j : j \in [0, k-1] - I(s_i|_{t^+})\}$;
8:         else
9:             $f_{i,t} = k$;
10:    $A_{S,t} \leftarrow \frac{1}{\varphi} 2^{\sum_{i=1}^{l} f_{i,t}/l}$;
11: Return $A_{S,t}$.
```

Similar to Lemma 1, the following Lemma holds for every pair of $A_{S,t}$ and $n_{S,t}$ if $L = \frac{1}{\epsilon}$.

Lemma 4 *For a given t, ϵ, and δ, $A_{S,t}$ returned by Algorithm 2 against the output of Algorithm 1 has the property that $P(|A_{S,t} - n_{S,t}| < \epsilon n_{S,t}) \geq 1 - \delta$ if $L = \frac{1}{\epsilon}$, $l = O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$, and $k = O(\log m + \log \delta^{-1} + \log \epsilon^{-1})$.*

Proof 1 *If $A_{S,t}$ is returned from only counting $\mathcal{L}$, it is the exact answer. The lemma is immediate.*

Consider that $A_{S,t}$ is returned from $\{s_i : 1 \leq i \leq l\}$. It can be immediately verified that Algorithm 2, in this case, is equivalent to: 1) doing a select on S to output $S|_{t^+}$, and then 2) applying FM algorithm on $S|_{t^+}$, 3) $n_{S,t} > \frac{1}{\epsilon}$. According to Lemma 1, this lemma is also immediate.

Lemma 4 and Theorem 1 together imply that Algorithm 1 and Algorithm 2 guarantee the ϵ-approximation with probability (confidence) $1 - \delta$ if $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta} \log m)$ space is allocated when $m > \epsilon^{-1}$ and $m > \delta^{-1}$.

3.2 Time Complexity

Suppose that we treat an element with value 0 in an s_i as having the time-stamp 0. The query algorithm (Algorithm 2), to compute $f_{i,t}$ for each i and a given t, selects the sub-index of the left-most element, from s_i, with the time-stamp smaller than t. Below we show it can be conducted in a logarithmic time if a min-heap is maintained.

As illustrated in Figure 4, each s_i is organized by a *min-heap* [3] built against the time-stamp values on a binary balanced tree. Then, we search the heap according to the order of depth first following the left-most path from the root satisfying the criterion - the search key value is smaller than t. For instance, for $t = 3$ the search returns the first element; thus, $f_{i,t} = 0$ (the subindex of 1st element). It can be immediately verified that such a search can be done in $O(logk)$. Consequently, Algorithm 2 can run in time $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta} \log \log m)$ if $m > \epsilon^{-1}$ and $m > \delta^{-1}$ since there are $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ subsketches and $k = O(\log m)$.

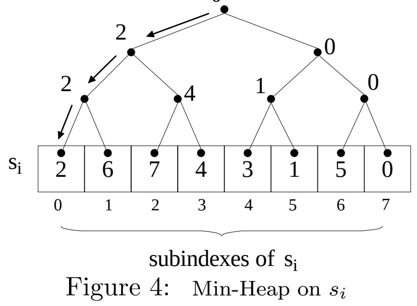


Figure 4: Min-Heap on s_i

Algorithm 1 runs in time $O(\log \frac{1}{\epsilon})$ per element to dynamically maintain $\mathcal{L}$ if we maintain a search tree on $\mathcal{L}$. As discussed earlier, each $h_j()$ $(1 \leq j \leq l)$ takes constant time on average to hash a data element. Thus, Algorithm 1 runs in time $O(\frac{1}{\epsilon^2} \log \delta^{-1} \log \log m)$ on average per data element, given there are $O(\frac{1}{\epsilon^2} \log \delta^{-1})$ such arrays and maintaining such a min-heap takes $O(\log \log m)$ time.

3.3 PCSA-like Algorithm

While Algorithm 1 is space-efficient and guarantees a probabilistic relative ϵ-approximate, each element is hashed into $\Omega(\frac{1}{\epsilon^2} \log \delta^{-1})$ arrays (subsketches). This potentially makes the algorithm less efficient. Our performance study in Section 6 demonstrates it can only handle a medium speed data stream in real-time.

Consider that in many recent applications, to support on-line computation of high speed data streams is a crucial requirement. In this subsection, we propose a time-efficient algorithms following the framework in the last section. It retains the space requirement but there is no theoretical guarantee of accuracy with confidence $1 - \delta$ though the expected accuracy is ϵ-approximate. Our performance study, nevertheless, indicates the algorithm is practically very space-efficient and highly accurate, and it is able to support on-line computation of high-speed data streams.

The algorithm is an immediate application of PCSA technique [7] to our algorithm, Algorithm 1. The basic idea is to hash each data element randomly to ζ arrays (subsketches) among the l arrays (subsketches). Algorithm 1 may be modified as follows.

- First, we pick at random another ζ hash functions: $\{H_i : 1 \leq i \leq \zeta\}$ besides the l hash functions in Algorithm 1, where each H_i hashes the element domain $\mathcal{D}$ to $[1, l]$.
- Then, in Algorithm 1 instead of the iteration (in line 9) from $i = 1$ to l, we do the iteration for each $i \in \{H_1(x), H_2(x), ..., H_\zeta(x)\}$. The others in Algorithm 1 remain the same.

We call such a modified Algorithm 1 "Algorithm SE-PCSA". Suppose that all the parameters are selected as those in Lemma 4. It is immediate Algorithm SE-PCSA runs in time $O(\zeta \log \log m)$ for each data element.

In the light of PCSA technique, Algorithm 2 is modified accordingly as follows to estimate a $n_{S,t}$. We change line 10 in Algorithm 2 to $A_{S,t} \leftarrow \frac{l}{\zeta\varphi} 2^{\sum_{j=1}^{l} f_{j,t}/l}$. It can be implemented in the same way as what we described in Section 3.2 with the same time complexity. These, together with the facts in [7], immediately imply that the expected accuracy of Algorithm SE-PCSA is ϵ-approximate. Note that in our implementation, we use a pairwise independent hash function for H_i and our experiment indicates that when ζ approaches 100, its accuracy remains quite stable.

4 K-skyband Technique

While the PCSA-like algorithm speeds up the sketch maintenance algorithm by hashing each element into a small number ζ ($\zeta \leq 100$ in our implementation) of randomly selected FM-based subsketches, it still retains $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$ subsketches to pursue ϵ-approximation. Consequently, the query algorithm runs in $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta} \log \log m)$ time; it is not very efficient when ϵ is small. This prevents us from processing a large number of queries simultaneously in real-time.

Our second sketch technique is based on BJKST algorithm since BJKST requires only $O(\log \frac{1}{\delta})$ sub-sketches to achieve ϵ-approximation with $1 - \delta$ confidence. We will show that applying BJKST algorithm to our problem DCSW leads to the k-skyband problem. Consequently, we show our sketch technique requires the (expected) space $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta} \log n)$. Our experiment indicates that in practice, it requires much less space than that required by SE-FM (Algorithm 1) or PCSA-like technique. Moreover, this sketch technique enables us to develop an efficient query algorithm with $O(\log \frac{1}{\delta}(\log \frac{1}{\epsilon} + \log \log n))$ (expected) time in contrast to $O(\frac{1}{\epsilon^2} \log \frac{1}{\delta} \log \log m)$.

The rest of the section is organized as follows. We first outline the algorithm. Then, we provide the details towards space minimization, querying k-skyband, and efficient maintenance techniques of a k-skyband.

4.1 Outline

We maintain l subsketches. Without loss of generality, we assume the domain $\mathcal{D}$ of object IDs is $[1, m]$. For each sketch i, a randomly generated pairwise hash function h_i hashes $[1, m]$ to $[1, m^3]$ so that each data element (x, t_x) is hashed to s_i by the form $(x, h_i(x), t_x)$ for all i.

Query. Similar to our FM-based algorithm, we may divide a query algorithm logically into two parts. For a given t, we first select all elements with time-stamps not smaller than t; that is, get $S|_{t^+}$. Then, we apply BJKST algorithm to $S|_{t^+}$. More specifically, in each s_i we choose the set $s_i|_{t^+}$ of elements with time-stamps not smaller than t. Then, we apply the BJKST query method, as described in (2) and the last paragraph of Section 2.2.2 on all $s_i|_{t^+}$ for $1 \leq i \leq l$.

To address the accuracy guarantee not covered by Lemma 3 when $n_{S,t} < k$ (e.g. a sliding window with the number of distinct objects less than k), we globally maintain a $\mathcal{L}$ to store the most recent k distinct objects, that is, the k distinct objects with the largest time-stamp values. Then, for each t we first find out if $t > t_{sml}$ where t_{sml} is the smallest time-stamp (oldest object) among the elements in $\mathcal{L}$. If $t > t_{sml}$, then query $\mathcal{L}$; otherwise query all s_i as above.

Let the query result obtained above be denoted by $A_{S,t}$. We have the following Theorem - ϵ-approximation.

Theorem 2 *Suppose that* $0 < \epsilon, \delta < 1$. *If* $m \geq \delta^{-1}$, $k = O(\frac{1}{\epsilon^2})$, $l = O(\log \delta^{-1})$, *then* $P(|A_{S,t} - n_{S,t}| < \epsilon n_S) \geq 1 - \delta$.

Proof 2 *If $A_{S,t}$ is obtained from $\mathcal{L}$, then it is the exact solution. The Theorem holds.*

If $A_{S,t}$ is obtained from s_i, then it can be viewed as if we applied BJKST algorithm on $S|_{t^+}$. Therefore, the theorem holds according to Lemma 3.

Sketch Maintenance. As with Algorithm 1, once a new element comes we first examine if an element (x, t_x) in $\mathcal{L}$ needs to be replaced by the new element. Meanwhile, we hash the new element (x, t_x) into $\{s_i : 1 \leq i \leq l\}$ by the form $(x, h(x), t_x)$.

Below, we show that in each s_i, we do not have to keep every element. We only need to keep "k-skyband".

4.2 Space Minimization

A hashed data element $(x, h_i(x), t_x)$ is kept in the sketch of s_i (for $1 \leq i \leq l$) if s_i does not "k-dominate" $(x, h_i(x), t_x)$. An s_i *k-dominates* $(x, h_i(x), t_x)$ iff there are k data elements in s_i with distinct hash values not greater than $h_i(x)$ and their time-stamp values not smaller (not older) than t_x.

The *distinct value based k-skyband* of s_i is the set, denoted by $SK(s_i)$, of data elements in s_i that are not k-dominated by s_i.[2] In our sketch algorithm, we maintain the distinct value based k-skyband $SK(s_i)$, instead of s_i, for $1 \leq i \leq l$. To simplify the notation, we abbreviate "distinct value based k-skyband" to "k-skyband" hereafter in this paper.

Theorem 3 *Each current $SK(s_i)$ for $1 \leq i \leq l$ has the following properties.*

P1: *If s_i currently k-dominates an element e ($\in s_i$), e will never be used by our query algorithm above for any t.*

P2: *For each element $e = (x, h_i(x), t_x) \in SK(s_i)$, either*

[2]The problem is the same as "k-Skyband" in [14] if we focus on $(h(x), t_x)$ only, except we enforce distinct values.

P2a: *there is a t_0 such that $h(x)$ is the kth smallest among the elements in D_{s_i,t_0^+} where D_{s_i,t_0^+} denotes the set of elements in s_i with distinct hash values and time-stamp values at least t_0, or*

P2b: *$h(x)$ is one of the $k-1$ smallest distinct hash values in s_i.*

Proof 3 *We prove P1 and P2 as follows.*

Proof of P1. *According to the definition, if $e = (x, h_i(x), t_x)$ is k-dominated by s_i then we have the property that there are at least k elements in s_i with distinct hash values not greater than $h_i(x)$ and come after (inclusive) t. This property will be retained regardless how many new elements come. Consequently, our query algorithm will never choose $h_i(x)$. Thus, P1 holds.*

Proof of P2. *If $(x, h_i(x), t_x)$ does not belong to category P2b, then there are λ ($\lambda > k-1$) elements in s_i with distinct hash values smaller than $h_i(x)$.*

Let t_0 is the time-stamp of the element with the $(k-1)$th largest time-stamp value among these λ elements. Since e is in $SK(s_i)$, among these λ elements there are only λ_1 ($\lambda_1 \leq k-1$) elements with time-stamp values larger than t_x. Therefore, $t_0 \leq t_x$; that is, e belongs to category P2a.

Note that P1 in Theorem 3 implies that we only need to maintain $SK(s_i)$. Clearly, an element in the category P2a will be used in a DCSW query with time t_0. Moreover, any element with one of the $k-1$ smallest distinct hashed values (category P2b) may be used in processing DCSW for the whole stream once future elements have hashed values smaller than the current $k-1$ smallest values; thus, it needs to be kept. Therefore, Theorem 3 implies that $SK(s_i)$ is the minimum number of elements we should keep to achieve ϵ-approximation. In fact, we can also show that $SK(s_i)$ (for $1 \leq i \leq n$) has the following (expected) space.

Theorem 4 *In a 2-d set $s = \{(x_i, y_i) : 1 \leq i \leq n\}$ with n elements, assume all x and y values are unique, x and y are independent, each x follows a same distribution, and each y also follows a same distribution. Then, the k-skyband $SK(s)$ has the expected number of elements $O(k \ln(\frac{n}{k}))$ where x_i corresponds to a hashed value and y_i corresponds to a time-stamp.*

Proof 4 *Without loss of generality, we assume that $y_i > y_j$ if $i < j$.*

For $1 \leq i \leq n$, let the random variable $X_i = 1$ if (x_i, y_i) is a k-skyband element, otherwise, $X_i = 0$. The expected number of k-skyband elements is $E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} P(X_i = 1)$ where P denotes the probability.

Clearly, the value (0 or 1) of each X_i (for $1 \leq i \leq n$) depends on $\{(x_j, y_j) : 1 \leq j \leq i-1\}$ as y_j is decreasingly ordered and any element (x_j, y_j) for $j > i$ does not dominate (x_i, y_i).[3] Note that every element (x_i, y_i) when $i \leq k$ belongs to $SK(s)$; thus, $P(X_i = 1) = 1$ when $i \leq k$.

For $i > k$, (x_i, y_i) is a k-skyband element iff x_i is one of the k smallest values in $\{y_j : 1 \leq j \leq i\}$. Note that each y_j has the same probability to fall into the k smallest values as each y_j follows the same distribution, and we assume the independence among all y_j and between x and y. Thus, $P(X_i = 1) = \frac{k}{i}$.

It can be immediately verified

$$E(\sum_{i=1}^{n} X_i) = k + \sum_{i=k+1}^{n} P(X_i = 1) = k \times (1 + H_{1,n} - H_{1,k}).$$

Here, $H_{1,n} = \ln(n)$, the Theorem immediately follows.

[3]*(x, y) dominates (x', y')* iff $x \leq x'$ and $y \geq y'$.

From Theorems 2 and 4, it follows that to achieve ϵ-approximation we need (expected) $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\log n)$ space with the high probability (confidence) $1-\delta$ if all time-stamps are unique, objects and time-stamps are independent, object ID of each element follows the same distribution, and the time-stamp of each element follows the same distribution. Our experiment demonstrates that in practice, this algorithm requires a smaller space than the FM-based techniques in Section 3.

4.3 Query Algorithm

Keeping only $SK(s_i)$ ($1 \le i \le l$) not only minimize the information kept but also enables us to develop an efficient query algorithm. Below, we show that querying each s_i for t by searching $SK(s_i)$ takes $O(\log|SK(s_i)|)$ time only.

For a given t, s_i has two cases: A) s_i has at least k elements with distinct hashed values and time-stamp values not smaller than t; B) not A). The theorem below is a key.

Theorem 5 *For a given t and i, suppose that e_1 is the element in $SK(s_i)$ with the smallest time-stamp among all elements arriving no earlier than t, and the time-stamp of e_1 is the r_1th smallest in $SK(s_i)$. Then, s_i belongs case A) if and only if the element e_2 with the (r_1+k-1)th smallest hashed values in $SK(s_i)$ has the kth smallest hashed values among the elements in $SK(s_i)$ coming no earlier than t.*

Proof 5 *First, we can show that for any element $e \in SK(s_i)$ its hashed value always falls in the $(r+k-1)$th smallest values of all hashed values in $SK(s_i)$ where e has the rth smallest time-stamp in $SK(s_i)$. This can be immediately shown by mathematic induction from the element with the smallest time-stamp in $SK(s_i)$ based on the fact that e is not k-dominated by $SK(s_i)$.*

The above fact immediately implies that in $SK(s_i)$ if we remove from $SK(s_i)$ all elements (r_1-1 in total) with the time-stamp values smaller than t_1 then e_2 has the kth smaller values among the remaining elements.

Based on Theorem 5 and in the light of our BJKST-based query algorithm, for any t we only need to find such an e_1 against time-stamps to determine r_1 and then find e_2 against hashed values. If such e_2 does not exist (i.e., less than k elements in $SK(s_i)$ with time-stamps not smaller than t), then we return $(n-r_1+1)$ according to BJKST query algorithm in Section 2.2.2.

To speed-up the search, we continuously maintain a binary search tree eT on t and a binary search tree eV on $h(x)$ with the information of the number of elements in each subtree, respectively, for the elements in $SK(s_i)$ for $1 \le i \le l$. It is clear with such search trees, the query algorithm above can be done in $O(\log|SK(s_i)|)$ time. Regarding the expected size of $SK(s_i)$, to achieve ϵ-approximation with confidence $1-\delta$, our query algorithm runs in (expected) time $O(\log\frac{1}{\delta}(\log\frac{1}{\epsilon}+\log\log n))$ as there are $O(\log\frac{1}{\delta})$ subsketches.

Below we describe our query algorithm in Algorithm 3.

4.4 Sketch Maintenance

We present our techniques to continuously maintain each $SK(s_i)$ for $1 \le i \le l$, as well as eT_i and eV_i. Clearly, if we have determined which element should be added or removed from $SK(s_i)$, the eT_i and eV_i can be updated (insertion or deletion) in time $O(\log|SK(s_i)|)$ per element by using the standard tree rebalancing technique in [3]. While every new element has to be added into $SK(s_i)$, below we present an efficient technique, by effectively utilizing eV_i, to determine whether or not elements in $SK(s_i)$ should be removed.

An immediate way is to compute the *total dominance count* for each element e; that is, count the number of elements in $SK(s_i)$ that dominate e. Nevertheless,

Algorithm 3 Query Algorithm

```
Input: Query time $t$, $\{SK(s_i), eT_i, eV_i : 1 \le i \le l\}$;
Output: $A_{S,t}$;
Description:
1:  get $t_{sml}$ from $\mathcal{L}$;
2:  if $t > t_{sml}$ then
3:      return $|\mathcal{L}|t^+|$;
4:  else
5:      for $i = 1$ to $l$ do
6:          compute $r_1$ regarding $t$ against $eT_i$;
7:          if $r_1 + k - 1 \le |s_i|$ then
8:              compute the $(r_1+k-1)$th smallest hashed value $v_i$ against $eV_i$;
9:              $A_i \leftarrow \frac{k \times m^3}{v_i}$;
10:         else
11:             $A_i \leftarrow (|s_i| - r_1 + 1)$;
12:     return the median of $\{A_i\}$ as $A_{s,t}$;
```

in our problem setting there may be many elements in each $SK(s_i)$ since k has to be $O(\frac{1}{\epsilon^2})$ to guarantee ϵ-approximation. In addition, an element may dominate many other elements. Therefore, simply computing dominance count for each element is too expensive; our experiment in Section 6 confirms this.

In fact, we do not have to count dominance for each element; instead, many times we can record the dominance count for a group of elements.

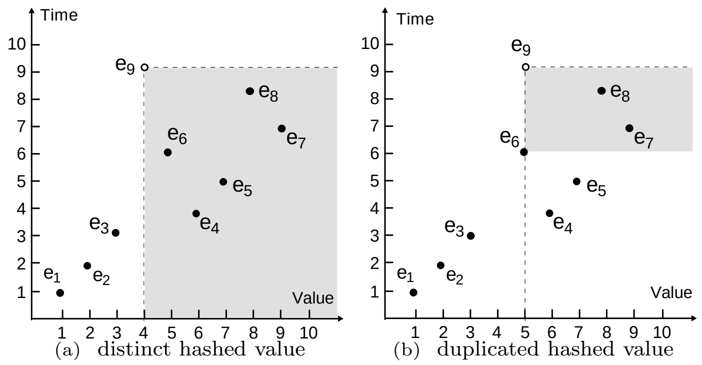


Figure 5: Example for two kind of dominate

Example 3 *Let $k=2$ and $l=1$. As depicted in Figure 5(a), suppose the elements $e_1, e_2, \cdots, e_8$ have been hashed into s_1 where x-coordinators are hashed values $h(e_i)$ and y-coordinators are time-stamps (the larger, the younger). It can be verified that before e_9 arrives, these 8 elements form 2-skyband $SK(s_1)$. Once e_9 is in, according to the definition it dominates the elements from e_4 to e_8. Instead of updating dominance count of each element from e_4 to e_8, it is possible to group them into several groups and then adding dominance counts on each group to avoid visit each individual element.*

Below, we first present an augmentation of eV_i to speed-up our computation of continuously maintaining $SK(s_i)$ for $1 \le i \le l$.

4.4.1 Dominance Aggregation Search Tree

We augment an eV_i ($1 \le i \le l$) tree to a dominance aggregation search tree (dAs-tree). As depicted in Figure 6, a *balanced* binary search tree is maintained on hashed values over the 8 elements from e_1 to e_8. Since the number of elements in a subtree is irrelevant to this part and they can be maintained in a similar way as search keys, we omit them in our discussions here.

To simplify the presentation, we enforce the constraint that all elements are at the leaf-levels, and each intermediate node has two children; that is, an AVL-like tree but all elements allocated at leaf nodes. At each node, we keep 5 search values h, α, β, t_{min}, and t_{max}. Here, h stores the search key value built on all hashed values, t_{min} stores the minimum time-stamp, and t_{max} stores the maximum time-stamp in the subtree. In addition, at each node j, α_j

denotes the captured *dominance count* of this node,[4] β_j is defined as follows.

$$\beta_j = \max\{\sum_{e \in p} \alpha_e : \ \forall p \in \mathcal{P}_j\} \tag{3}$$

Here, $\mathcal{P}_j$ is the set of all paths from node (exclusive) j to a leaf, and p is such a path. At leaf-node, we do not need to count β (thus it is omitted in our implementation), and $t_{min} = t_{max} = t$ where t is the time-stamp of the element kept there. It can be immediately verified that if E_L and E_R are the two children of node j, then

$$\beta_j = \max\{(\alpha_{E_L} + \beta_{E_L}), (\alpha_{E_R} + \beta_{E_R})\} \tag{4}$$

Note that the dominance relationship of "e dominating e'" is captured either at e' or at an ancestor of e'.

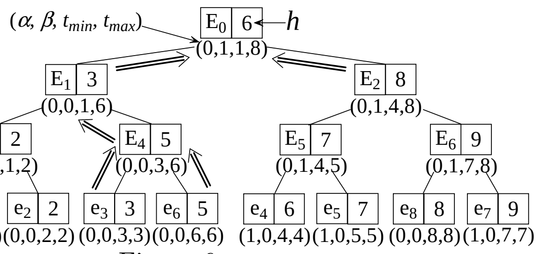


Figure 6: a dAs-tree

In Figure 6, we assume that the dAs-tree is built by the bulk-loading technique [15] once e_8 comes and all dominance counts have been done on each node. Note that e_6 dominates both e_4 and e_5 as e_6 comes later with smaller hashed values. In this case, we could record either $\alpha = 1$ at node E_5 or $\alpha = 1$ at both e_4 and e_5. Later option is used in Figure 6.

A dAs-tree is *balanced* if it satisfies the criteria for an AVL-tree. In the next section, we will show how a dAs can be rebalanced by applying standard rebalancing technique in [3].

4.4.2 Algorithm

Our algorithm to continuously maintaining $SK(s_i)$ is outlined in Algorithm 4. Below we describe the 3 steps in Algorithm 4 in details.

Algorithm 4 Continuously Maintaining k-Skyband

```
Input: k, l hash functions $\{h_j : 1 \le j \le l\}$,
       dAs-trees $eV_j$ for $1 \le j \le l$;
Output: $SK(s_j)$ for $1 \le j \le l$;
Description:
1: for a new $(x, t_x)$ do
2:   for $i = 1$ to $l$ do
3:     STEP 1. Update dominance counts in $eV_i$
              against $(x, h_i(x), t_x)$;
4:     STEP 2. (Possibly) delete the nodes
              not in $SK(s_i)$ from $eV_i$ and $eT_i$;
5:     STEP 3. Insert $(x, h(x), t_x)$ into $eV_i$ and $eT_i$;
```

Step 1: Update Dominance Counts. There are two cases once $(x, h_i(x), t)$ is obtained.

Case 1. There is no element in the current $SK(s_i)$ with the same hashed value as $(x, h_i(x), t)$.

Case 2. There is an element in the current $SK(s_i)$ with the same hashed value as $(x, h_i(x), t)$.

Case 1. Regarding the example in Figure 5(a), no element dominates e_9 as e_9 is the most recent element; nevertheless, e_9 dominates the elements from e_4 to e_8. In this example, we need to increase the α at e_6 by 1. In addition, the dominance counts of e_4, e_5, e_7, and e_8 can be "globally" updated at E_2 by increasing its α by 1 without having to visit any of its descendants to save computation costs. We can immediately verify that at each node i,

Max-Count. $(\beta_i + \alpha_i + \sum_{j \in \pi_i} \alpha_j)$ is the maximum total dominance count among elements in its subtree, where π_i is the set of ancestors of node i.

This, together with (3), require that in our algorithm, any update of the dominant count α at a node e has to be propagated to its root by updating all β values, by using (4), on the path. We describe our algorithm below in Algorithm 5.

Algorithm 5 UpdatedAs $(eV_i, (x, h_i(x), t_x))$

```
Input: $eV_i$ and $(x, h_i(x), t_x)$;
Output: Updated $eV_i$;
Description:
1:  get the root $E$ of $eV_i$;
2:  if $E$ is a leaf then
3:    if $h(x) < E.h$ then $E.\alpha := E.\alpha + 1$
4:  else
5:    if $h(x) < E.h$ then
6:      $E2.\alpha := E2.\alpha + 1$; ($E2$ is the right-child)
7:      UpdatedAs (dAs.$E1$,$(x, h_i(x), t_x)$);
        ($E1$ is the left-child)
8:    else
9:      UpdatedAs (dAs.$E2$,$(x, h_i(x), t_x)$);
10: $E.\beta := \max\{E1.\alpha + E1.\beta, E2.\alpha + E2.\beta\}$;
```

In Algorithm 5, $E.h$ denotes the search key value of the tree at node E, $E.\alpha$ and $E.\beta$ denote the α and β values, respectively at E, dAs.$E1$ denotes the subtree with root $E1$. It can be immediately verified that the algorithm visit at most two nodes at each level; thus the algorithm runs in $O(\log |SK(s_i)|)$ $(= O(\log |eV_i|))$ time per element.

Example 4 *Regarding the example in Figure 5(a), once e_9 arrives, the updates to dominance counts of the tree in Figure 6 follow the arrows as illustrated. In Figure 7, we illustrate the result that only contains the nodes with some changes.*

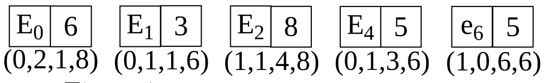


Figure 7: Updates of a dAs-tree: Case 1

Case 2. Regarding the example in Figure 5(b), the hashed value $h(x)$ of e_9 is the same as that of e_6. Thus e_6 has to be removed. In this case, if we update the existing dAs-tree in Figure 6 in the same way as Case 1. We over-count the dominance count of e_4 and e_5 as they have already counted by e_6. To resolve this issue, we only update the dominance count of elements with time-stamp values greater than 6 - the time-stamp of e_6.

The algorithm has the same traversal paradigm as that in Case 1 (Algorithm 5) except we need to add the constraint - the time-stamps greater than t' where t' is the time-stamp of the element with the same hashed value as a new element. We can modify Algorithm 5 as follows.

- For the situation at line 3, we add the constraint $t' \le E.t_{min}$.
- For the situation at line 5, we also add the condition $t' \le E2.t_{min}$. Then, we do line 7 if another condition $t' \le E1.t_{max}$.
- We change the whole "else part" at line 8 and line 9 to "UpdatedAs (dAs.$E1$,$(x, h_i(x), t)$) if $h(x) < E.h$ and $E1.t_{min} < t' \le E1.t_{max}$, UpdatedAs (dAs.$E1$,$(x, h_i(x), t)$) if $t' \le E2.t_{max}$".

Note that after adding these constraints, the algorithm no longer guarantees logarithmic time complexity and runs in $O(TR)$ where TR is the size of the tree spanning the nodes visited. Although the algorithm may require linear time for one new element in the worst case, our experiment demonstrates that it is very efficient in practice.

Example 5 *Regarding the example in Figure 5(b), once e_9 arrives, the updates of dominance counts of the tree in Figure 6 are illustrated in Figure 8 where only the nodes with some changes are shown.*

[4]A *dominance count* at a node e is the number of other elements captured that dominate all elements in the subtree with root e.

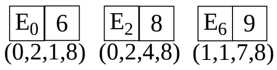


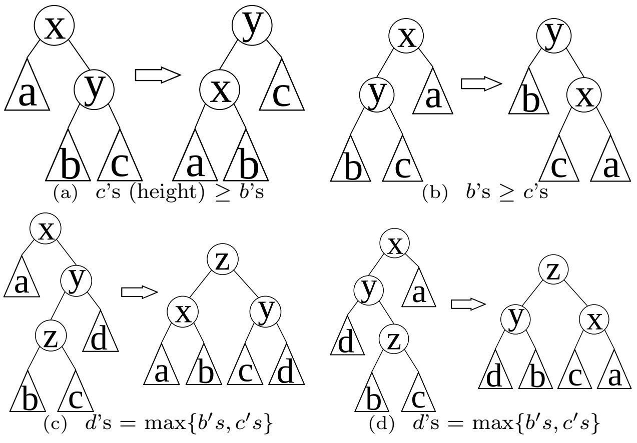
Figure 8: Updates of a dAs-tree: Case 2

Step 2: Removal Element. Removing an element from eT_i for each i can be done in $O(\log |eT_i|)$. Below we mainly focus on removing an element from each eV_i.

An element needs to be removed if either Case A: its hashed value is the same as that of the new element, or Case B: its total dominance count reaches k after the new element comes.

We can immediately verify that in both cases, we do not need to discount dominance counts at other elements in $SK(s_i)$ due to the removal of a new element. This is because for Case A, when we update the the dominance counts of other elements for the new element, we already discount the dominance counts caused by e. For Case B, any element dominated by e has been already removed in earlier rounds; this is because the total dominance count of e must be $k-1$ (for its to reach k) before a new element arrives.

Nevertheless, once an element e removes in both cases we may need to fix the β value of at its parent if the dominance count of e was larger than its sibling; consequently, it may be propagated to the root. Clearly, this takes $O(\log |SK(s_i)|)$ time. Moreover, once e removes from eV_i, we also remove its parent and connect its sibling to the grand parent to enforce the constraint that every internal node has two children nodes (noting all elements are kept at leaf level); meanwhile we also pass the α value at the original parent of e to e's sibling. Next, we may need to re-balance eV_i from the leaf-level. The AVL-tree balancing technique is to iteratively re-balance the tree from the leaf-part where the node is deleted; it is based on the following 4 cases as depicted in Figure 9 where in each case a, b, c, d are the subtrees. In each case, while re-balancing the tree, we also need to recalculate α and β. Specifically, we pass the α values of x, y, and z to their decedents a, b, c, and d respectively. Then after rebalancing the tree, we recalculate β values at x, y, z from a, b, and c in a bottom-up fashion, while α value at x, y, and z are assigned to zero. Clearly, this takes constant time.



(a) c's (height) $\geq b$'s (b) b's $\geq c$'s (c) d's $= \max\{b's, c's\}$ (d) d's $= \max\{b's, c's\}$

Figure 9: Rebalance

Example 6 *Regarding Case (d) in Figure 9, we give the α value at x to the roots of trees a, b, c, and d, respectively. Pass the α value at y to the roots of b, c, and d, and pass the α to the roots of b and c.*

After rebalancing, we calculate the β value at x, the β value at y, and the β at z iteratively by using (4).

Regarding the example in Figures 5 and 6, after removing the elements e_4, e_5, and e_7, the dAs-tree has been updated to the one as depicted in Figure 10 by the rebalancing technique.

In our implementation, we combine the update of dominating counts due to the removal of e with the rebalancing by one-pass bottom-up. The total time-complexity is $O(\log |SK(s_i)|)$. Finally, it should be clear that if the total dominance count at an element (leaf) is k, then at the

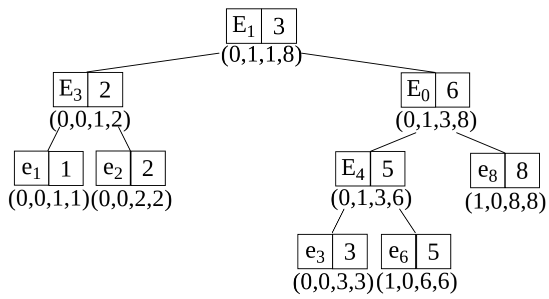


Figure 10: after removing e_4, e_5, and e_7

root $\alpha + \beta = k$ according to Max-Count property in the last section. Then, iteratively using the Max-Count property from the root we can reach an a leaf with the total dominance count k; this also runs in time $O(\log |SK(s_i)|)$.

Step 3: Insert an element. Clearly, inserting a new element into each eT_i can be done in $O(\log |SK(s_i)|)$. We focus on the corresponding update of each eV_i.

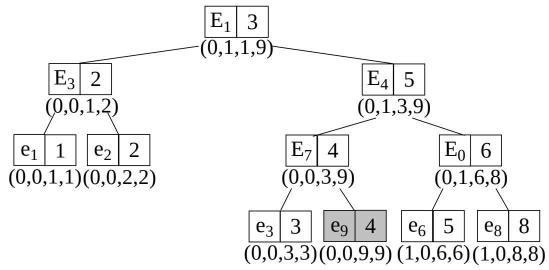


Figure 11: after inserting e_9

Recall that a new element e is not dominated by any existing elements. Therefore, after inserting e into eV_i, the α values at the ancestors of e should be 0. On the other hand, the original α values contributed to computing the maximum dominance count of the elements rooted at those ancestors (i.e. Max-Count property in Section 4.4.2); thus we iteratively update the β values to enforce the Max-Count property on the path p from the root to e while changing α to 0. We also need to pass the α of each node in p to its child which is not on p. Then, we use the above rebalancing technique to balancing eV_i, if necessarily. Clearly, the complexity per element is $O(\log |SK(s_i)|)$.

Regarding the example in Figure 10, after inserting 9 the dAs-tree becomes the one in Figure 11.

4.4.3 Remarks

To save computation costs, when we remove an element from eV_i with the same value as that of the new element, we just replace it by the new element to save possible rebalancing for insertion and deletion. Finally, according to the above time complexity analysis, the time complexity to maintain each $SK(s_i)$ is $O(\log |SK(s_i)| + TR_i)$ $(1 \leq i \leq l)$ per element where TR_i is the spanning tree size to cover the nodes visited in Case 2 of Step 1.

Note that to achieve the ϵ-approximation with confidence, $l = O(\log \frac{1}{\delta})$ and $|SK(s_i)|$ has the (expected) size $O(\frac{1}{\epsilon^2} \log n)$.

In [10, 17], efficient techniques have been developed to continuously maintain skyline over sliding windows. Nevertheless, these techniques are not applicable to continuously maintain $SK(s_i)$. This is because that in continuously maintaining skyline, an element is immediately removed if it is dominated by another element. The main challenge in continuously maintaining k-skyband is to efficiently computing the dominance counts. In [12], it proposes to use k-skyband to answer top-k queries over sliding windows. The technique is to simply re-compute the dominance count for each element once a new element arrives. It is efficient when k is small (a typical situation in top-k queries); nevertheless, it is inefficient when k is large - a typical situation in our problem to guarantee ϵ-approximation for a small ϵ. Thus, the technique in [12] are not applicable to our problem setting where we need to process data streams in real time.

5 Performance evaluation

We present the evaluation results of a comprehensive performance study. As mentioned earlier, the techniques in [8, 16] are the only 2 existing techniques which may be immediately applied to counting distinct elements against sliding windows. Nevertheless, the sketch technique in [8] requires a pre-fixed sample space $\Omega(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\sqrt{m})$, and the technique requires space $O(N\frac{1}{\epsilon^2}\log\frac{1}{\delta}\log m)$ in the worst case, to achieve ϵ-approximation with $(1-\delta)$ confidence. In fact, our experiment shows that for the datasets used below in our performance study, an application of the algorithm in [16] into our problem always requires tens of times more than the dataset size, while the technique in [8] requires at least tens of times more space than our technique. Therefore, we only present the performance evaluation of our techniques presented in this paper. We summarize them below.

SE-FM	Algorithm 1: the space-efficient sketch construction algorithm in Section 3.1.
SE-PCSA	The sketch construction technique in Section 3.3.
k-SKB	Algorithm k-Skyband in Section 4: sketch construction techniques.

We evaluate their space and time efficiency, as well as accuracy in terms of the relative errors; that is, $\frac{|A_{s,t}-n_{s,t}|}{n_{s,t}}$. The corresponding query algorithms are also implemented.

In our experiments, two synthetic datasets are generated, *Random* and *Zipf*. In a Random dataset, time-stamps of data elements are randomly generated following a uniform distribution, while time-stamps in a Zipf dataset follow a Zipf distribution. We assume that data elements arrive according to their time-stamps. We use *duplication ratio*, $\alpha = \frac{N-n}{N}$, to control the total number of duplicated objects, where n is the number of distinct objects and N is the total number of data elements. For each synthetic dataset, we first generate n distinct objects, then each object pairs one of the N time-stamps randomly according to a uniform model. Each of the remaining $(N-n)$ time-stamps randomly pair one of the n distinct objects according to a uniform model.

The following real dataset *WCH* (World Cup 98's HTTP request data) is used in our performance study. It is downloaded from the Internet Traffic Achieve [9] and consists of 20 million records of requests made to the 1998 World Cup Web site on June 10, 1998. Each record contains time-stamp, clientID, URLID, serverID, and package size (PSIZE). In the dataset, we treat ⟨clientID, URLID, serverID, PSIZE⟩ as an object. In the dataset, we found there are totally more than 1.97M duplicated data objects and the maximum duplication number of an object is 566.

All algorithms are implemented by C++ and the experiments have been carried out on a PC with Intel P4 2.8GHz CPU and 1G memory under the operation system - Debian Linux. Table 1 below lists the parameters that potentially have an impact on our performance study. In our experiments, all parameters use default values unless otherwise specified.

Notation	Definition (Default Values)
d	Dataset Model (Random)
N (syn. data)	Dataset Size ($10M$)
α (syn. data)	Duplication Ratio (0.2)
ζ (SE-PCSA)	Number of times to hash an item (100)
ϵ	Guaranteed Precision (0.02)
$1-\delta$	Confidence (0.95)

Table 1: System Parameters

To "discount" O notation in space requirements of SE-FM, SE-PCSA, and k-SKB, respectively, we adopt the same constant factor 2. That is, $l = \frac{2}{\epsilon^2}\log\delta^{-1}$ in SE-FM and SE-PCSA, and $k = \frac{2}{\epsilon^2}$ in k-SKB. In SE-FM and SE-PCSA, we choose $k = 32$ because we use the public code from Massive Data Analysis Lab to generate hash functions [11] and 2^{32} is large enough to accommodate massive number of distinct data elements; we also choose $L = \frac{1}{\epsilon}$. In k-SKB, we choose $L = k$, $l = \log\frac{1}{\delta}$. We also modify the code in [11] to generate hash functions in k-SKB.

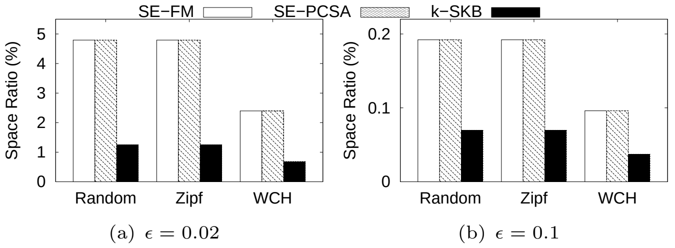


Figure 12: Space Efficiency against Different Dataset

5.1 Space Efficiency

We record the maximal size (i.e. the maximal number of elements) of sketch, by each algorithm, during the continuous processing of a dataset. The ratio of such sketch size to the total number of elements processed is called *space ratio*. We study possible impacts from dataset models, dataset sizes, ϵ and $(1-\delta)$. Note that the space requirement in SE-FM and SE-PCSA are the same and fixed for given m, ϵ, and δ, while the space ratio changes when data size changes. The space required in k-SKB is "opportunistic".

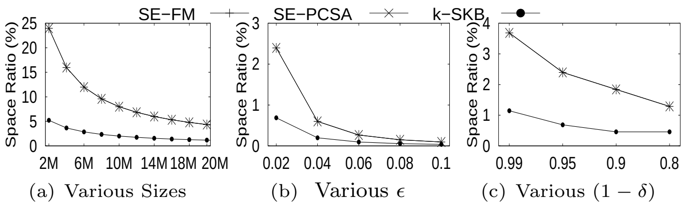


Figure 13: Impact of Sizes, ϵ, and δ

The first experiment results are presented in Figure 12. They demonstrate that k-SKB requires the smallest space. The second experiment evaluate the possible impacts from data sizes, ϵ, and δ. The evaluation results against the real dataset (WCH) are presented in Figure 13 where the experiments regarding Figures 13(b) and 13(c) are against the whole dataset. Again, they demonstrate that k-SKB requires the smallest space.

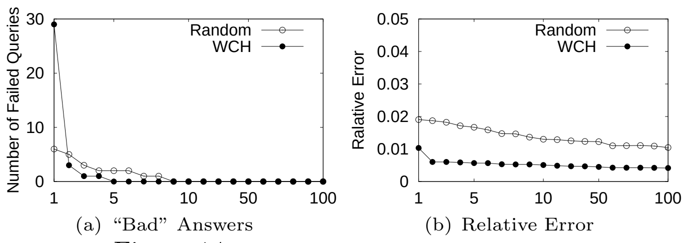


Figure 14: Accuracy of SE-PCSA Variants

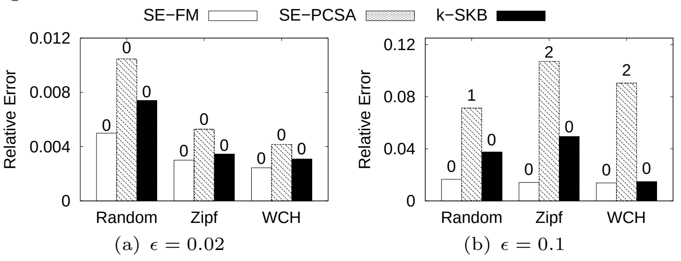
5.2 Evaluating Accuracy

The results of the first experiment, against the dataset Random and WCH, are reported in Figure 14. We study an impact of different values of ζ (i.e., the number of subsketches an element will be hashed in SE-PCSA).[5] As demonstrated by Figure 14(a), when $\zeta = 100$ the number of query results exceeding the relative error guarantee is 0, and an improvements of relative errors becomes less significant after $\zeta \geq 100$.

The second experiment is conducted against the 3 dif-

[5]In SE-PCSA, different values of ζ will not make any difference in space requirement if the other parameters are the same.

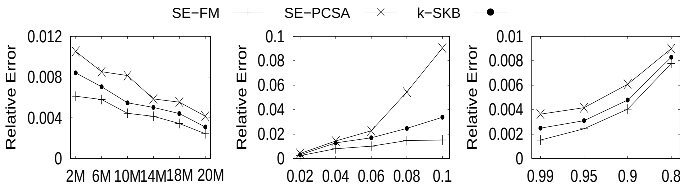
ferent datasets and is reported in Figure 15. It shows that SE-FM provides the highest accuracy, while k-SKB is the second. The numbers (0 or 1) above those "bar figures" are the numbers of answers exceeding their corresponding probabilistic (with confidence at least 0.95) relative error guarantees.



(a) $\epsilon = 0.02$ (b) $\epsilon = 0.1$

Figure 15: Accuracy against Different Dataset

The third experiment evaluates possible impacts from data sizes, ϵ, and $(1-\delta)$. The experiment is conducted against real dataset WCH and is reported in Figure 16. It shows that SE-FM always provides the highest accuracy and k-SKB is the second accurate. We also report that all answers obtained against the sketches by SE-FM or k-SKB satisfy the corresponding probabilistic error guarantees while SE-PCSA leads to 8 answers exceeding a designated relative error guarantee ϵ for the setting - $\epsilon = 0.02$ and $(1-\delta) = 0.8$

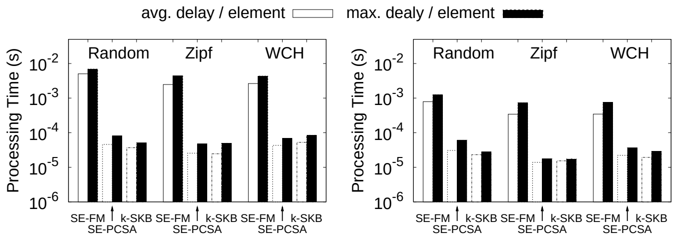


(a) WCH Dataset (b) Various ϵ (c) Various $(1-\delta)$

Figure 16: Effect of Sizes, ϵ, and $(1-\delta)$

5.3 Time Efficiency

The cost of processing one data element may be too small to be recorded accurately (especially for SE-PCSA and k-SKB), we record the average time for processing every batch of 1K elements as *the delay of one element.* In addition, we also record the maximum value of such delay per data element time as *the maximal delay of each element.*



(a) $\epsilon = 0.02$ (b) $\epsilon = 0.05$

Figure 17: Time Evaluation over Different Datasets

The first experiment is conducted against the 3 datasets Random, Zipf, WCH. The experiment results are reported in Figure 17. They indicate that SE-FM can only process a medium speed data stream online - 200-400 elements per second when $\epsilon = 0.02$ and about 2500 elements per second when $\epsilon = 0.05$. However, both k-SKB and SE-PCSA can process high speed data streams. They can process at least 20,000 data elements per second even with $\epsilon = 0.02$. We also tested the naive technique to maintain each $SK(s_i)$ regarding BJKST-based sketches; that is, we update the total dominance count of each data element once a new element comes (we first find the left-most element with hashed value smaller than that of the new element and then do a linear scan from the element). Our experiment shows that the naive algorithm can only process about 100 elements per seconds; thus it can support very slow data streams only.

The second experiment set evaluates possible impact of

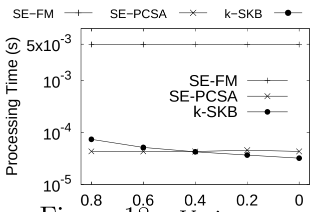


Figure 18: Various α

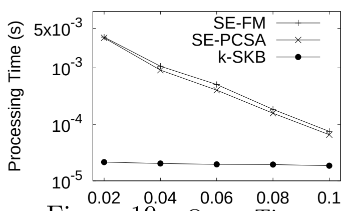


Figure 19: Query Time

duplication ratios. As we cannot change duplication ratios in real dataset, the dataset Random is used for this purpose. We record the average delay of an element. Figure 18 shows the experiment results. They demonstrate that our techniques are not very sensitive to different duplication ratios.

Finally, we evaluate the 3 query processing algorithms against the real dataset WCH. $1K$ queries are randomly generated as before. We vary ϵ from 0.02 to 0.1 and other parameters adopt default values. The average response time of the $1K$ queries for each algorithm, is reported in Figure 19, respectively. As expected, SE-FM and SE-PCSA have the similar performance as they use the same sketch structure for query; both of them require much more time than k-SKB does, especially when ϵ is small.

6 Conclusions

In this paper, we investigated the problem of approximately counting distinct elements against sliding windows (DCSW). Novel space and time efficient techniques are developed for continuously maintaining sketches so that a DCSW can be processed with the guarantee of ϵ-approximation. This is the first work providing the space and time efficient data stream techniques to approximately counting distinct objects over sliding windows. The space required by our techniques is near optimal. Besides proven accuracy and space guarantees, our algorithms are also efficient enough to support on-line computation of very high speed data streams with an element arrival rate up to 20K/second. Moreover, we showed that our techniques may be immediately extended to cover other problems, such as heavy hitters, fault-tolerant distributed computation, etc.

References

[1] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM'02.*

[2] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *ICDE'04.*

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* 2001.

[4] G. Cormode, S. Muthukrishnan, and W. Zhuang. What's different: Distributed, continuous monitoring of duplicate-resilient aggregates on data streams. In *ICDE'06.*

[5] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *SODA03.*

[6] W. Feller. *An Introduction to Probability Theory and Its Applications.* John Wiley & Sons, Inc., 1966.

[7] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[8] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, 2001.

[9] Internet Traffic Archive. `http://ita.ee.lbl.gov`.

[10] X. Lin, H. Lu, J. Xu, and J. X. Yu. Continuously maintaining quantile summaries of the most recent n elements over a data stream. In *ICDE'04.*

[11] Massive Data Analysis Lab. `http://www.cs.rutgers.edu/~muthu/massdal.html`.

[12] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD 2006.*

[13] D. Papadias, Y. Tao, R. Kalnis, and J. Zhang. Indexing spatiotemporal data warehouses. In *ICDE*, 2002.

[14] D. Papadias, Y. Tao, and F. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1):41–82, 2005.

[15] R. Ramakrishnan and J. Gehrke. *Database Management Systems, Second Edition.* 2000.

[16] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatiotemporal aggregation using sketches. In *ICDE 2004.*

[17] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *TKDE*, 18(2):377–391, 2006.