Top-k Frequent Items and Item Frequency Tracking over Sliding Windows of Any Size

Chunyao Song[1], Xuanming Liu[2], Tingjian Ge[2] and Yao Ge[1]

[1] *College of Computer and Control Engineering, Nankai University*
[2] *University of Massachusetts Lowell*

Abstract

Many big data applications today require querying highly dynamic and large-scale data streams for top-k frequent items in the most recent window of any specified size at any time. This is a challenging problem. We propose a novel approach called *Floating Top-k*. Our algorithm does not need to explicitly maintain any item counts over time or to deal with count updates upon item entry and expiration. Succinctly we use only a small-size data structure to retrieve the top-k items dynamically in a window of any size within an upper bound. We prove that the space and time costs of Floating Top-k grow only logarithmically with the window size rather than linearly as in previous work. Our comprehensive experiments over three real-world datasets show that Floating Top-k not only provides accuracy guarantees, but it also has two to three orders of magnitude smaller memory footprint, and is one to two orders of magnitude faster than previous approaches. Hence, Floating Top-k is both effective and scalable, significantly outperforming competing approaches. In addition, we devise a concise and efficient solution called *Progressive Trend Model* to a related problem of tracking the frequency of selected items, improving upon previous work by twenty to thirty times in model conciseness when having the same accuracy and efficiency.

Keywords: data stream, top-k frequent items, item frequency tracking, sliding window.

1. Introduction

In this big data era, data are generated at unprecedented rates. Social network data, web data, business/server logs, and data from the Internet of Things are just some examples. For such real-time data streams, it is often useful to dynamically query the top-k most frequent ("hot") items in a past window of any size chosen by users. The stream server thus needs to efficiently maintain a succinct data structure, ready to answer top-k most frequent item queries for a past window of any size at any time. Let us look at some examples.

Example 1. *Twitter has been a surging online social network [4]. Hashtags are a reasonable indicator of the contents and topics of the discussions in the Twitter platform. One may want to query, in real time, the top-10 hottest (most frequent) hashtags in the past hour, or in the past day, week, or month, and so on.*

Example 2. *As another example, for Amazon, or an Internet search engine, or an online news web site, one may be interested in finding the top-k purchased products, or rented movies, or searched terms, or clicked links, or browsed news, in a past window of any duration chosen by the user.*

We call this problem the *Windowed Top-k Frequent Items*. It is challenging because there is a large number of distinct data items and it is generally too expensive or even infeasible to maintain a counter for each distinct data item. For instance, in Example 1, with a high-rate Twitter stream, it is too costly to keep a counter for each hashtag string. Moreover, we need to get the top-k frequent items for a window of any size (within an upper bound W). A window slides as time goes, and items expire from windows at different times for windows of different sizes.

In addition to this Windowed Top-k Frequent Items problem, there are also application scenarios where a user would like to succinctly track the frequencies of selected items that he/she is interested in. For example, in 2016, one may be interested in tracking the frequency of Twitter hashtag "TownHallDebate" (for

U.S. presidential debates), which can be useful for political campaigns. Likewise, in Example 2, tracking the sales of a product over a long period of time may be beneficial for business intelligence. Tracing the popularity of a particular contagious disease through social networks can also help to understand the spread of the disease and to come up with preventive measures accordingly. Tracking item frequency is widely used in the information cascade study [22], for dynamic spreading processes, such as rumour propagation and marketing campaigns. In the field of security, monitoring the frequency of body contacts and mass events provides a possible way to avoid crowd disasters [18]. Moreover, tracking frequency has been successfully applied to capturing the styles and trends of scientific development [36]. Scientific memes can be abstracted into a pattern by tracking the frequency of occurrence and finding the relationship between the frequency and the degree to which the memes propagate along the ciatation graph [23]. Furthermore, in interactive data exploration, after querying the top-k frequent items, the user may want to individually track the frequency evolvement of some of these items. As time goes by, some of them may not be "hot" anymore, and the user would start another data exploration cycle—querying the top-k followed by tracking individual items.

We call this second problem an *Item Frequency Tracking Problem*. The goal here is to use a *succinct* model to *accurately* summarize the frequency history of the items in question. Thus, given that such a model satisfies some accuracy constraint, it is desired to be as concise as possible. Furthermore, obtaining the model through real-time stream data needs to be very efficient.

1.1. Related Work

Efficiently (in time and space) answering frequent item (i.e., heavy-hitter) queries over the *entire data stream* or over a *static window* has been extensively studied. Cormode and Hadjieleftheriou [13] give an excellent survey with experimental comparisons of various approaches. For instance, the work by Misra and Gries [31] is one of the algorithms. In our experiments, we implement the FREQUENT algorithm (i.e., Algorithm 1) in [13] to find out the *ground truth* top-k

items in any *static* fixed window for comparing various competing approaches. However, there are fundamental differences between [13] and our work. We study the top-k most frequent items problem *in a past window of any size*; we can dynamically answer the query *at any time*.

Liu et al. [27] give another excellent survey that summarizes different methods for heavy hitters. The existing algorithms are divided into sampling-based, counting-based and hashing-based categories. As stated in [27], although sampling might be the most straightforward way to deal with big data, there are some shortcomings with sampling for data streams. Specifically, it is difficult to decide the sampling rate when the size of the stream is unknown. In addition, as the answer quality usually improves as the sample size increases, achieving a required accuracy may need samples that exceed the storage capacity. In contrast to counting-based and hashing-based methods, sampling-based algorithms only update the synopsis at the samples. As such, [27] only focuses on counting-based and hashing-based methods. We will also discuss counting- and hashing-based related work in detail. The surveyed data stream models include time-sensitive models, distributed models, hierarchical and multi-dimensional models, and skewed data models. Sliding window falls into the time-sensitive models. Over the years there is a series of work on heavy-hitter queries in a sliding window, with a subsequent one improving upon a former one: Golab et al.'s [20], Arasu and Manku's [6], and Lee and Ting's [26]. As experimentally demonstrated by Homem and Carvalho [21] (and verified in our experiments), these approaches do not perform as well as the algorithms in [21] for top-k frequent items over a sliding window. In addition, Chen et al. [12] present an algorithm λ-$HCount$ based on a time fading model for computing the frequency counts of stream data. It uses r hash funcions to estimate the density values of data items and has a similar problem as the aforementioned three approaches. More importantly, heavy-hitter queries rely on a frequency threshold Φ to determine which items to pick. However, in big data applications and large-scale dynamic streams such as Twitter, even the top-k items in a large window may be only a tiny fraction out of the total. Furthermore, as the window slides and

stream changes, and for different window sizes that users may request, Φ is highly dynamic and is never fixed for top-k items. Thus, it would be impossible to determine a good Φ value to use. A conservative very-low Φ value would return too many items. To make matters worse, most of these heavy-hitter approaches do not provide frequency estimates for each item [21], making it impossible to pick the top-k among many that are returned. Hence, among this line of work, we only show the comparisons with Homem and Carvalho's work [21] in our experiment section (in addition to the comparisons with Persistent Data Sketching [41] and Piecewise Linear Approximation [34]).

Homem and Carvalho's algorithms [21] extend the Space-Saving algorithm originally proposed by Metwally et al. [29] to handle sliding window queries; the new algorithm is called *Filtered Space-saving with Sliding Windows*. However, like other sliding window work discussed above, a major limitation is that it is only designed for *one fixed window size*, even though the window slides over time, while our method simultaneously handles *all window sizes* within an upper bound W. Note that this is not the same as *one fixed window size*. W is the farthest data unit that users care about, e.g., a year, a month or a day ago. We can answer any query of window size $w \in [1, W]$ dynamically by only maintaining one small-size data structure. Of course, with Filtered Space-saving with Sliding Windows, one can independently handle *multiple* fixed window sizes, but then the memory footprint and processing cost have to add up linearly as well. Even with a *single window size*, the algorithm has a high space complexity $O(kW)$ and maintenance time complexity $O(k \log k + kW)$ per time unit. With *all window sizes*, this space and time complexities become $O(kW^2)$ and $O(kW \log k + kW^2)$ respectively. It is worth noting that Gibbons et al. [19] improve upon [16] for the basic counting problem. The worst case per-data-element processing time is improved from $O(\log W)$ to $O(1)$ and only $O(\frac{1}{\epsilon} \log^2 W)$ memory bits are needed for each data item. Suppose the number of different items in window W is N, it is possible to handle top-k frequent items query in $O(\frac{k}{\epsilon} \log^2 W)(\epsilon \in (0, 1))$ memory bits by using the method of [19] and scanning the stream N times. Although the space complexity is low enough, its time complexity is $O(NW + kWlogk)$

for all window sizes. By contrast, our method only has a space complexity $O(k \log W)$ and a time complexity $O(k \log k \log W)$ per time unit.

Another line of work extends data stream sketches to handle heavy-hitter queries in a sliding window. Papapetrou et al. [35] extend Count-Min sketch [15] with exponential histograms [16] to handle sliding windows, and call the new sketch Exponential Count-Min which can handle various kinds of queries including point queries, inner product and self-join queries, and heavy-hitter queries (while we only focus on top-k frequent item queries). More recently, Wei et al. [41] propose *Persistent Data Sketching*, which handles, among other types of queries, heavy-hitter queries over sliding windows. Like [35], Persistent Data Sketching also extends the Count-Min sketch [15], but using the more succinct piecewise functions [34] (instead of exponential histograms as in [35]). Thus, it improves upon the version of [35] (as its analysis shows). We compare with Persistent Data Sketching in our experiments. Moreover, some novel sketches based on the time fading model or distributed streams are proposed. Cafaro et al. [10] present a new sketch based algorithm where the key ideas are borrowed from forward decay, the Count-Min and the Space Saving algorithms. It works in the time fading model. [11] implements a parallel version of [10]. Shah et al. [38] focus on query estimation over sliding window distributed data streams. They propose an efficient Exponential Space Saving sketch approach, whose overall memory growth is sub-linear with respect to the data size and length of sliding window. Exponential Space Saving sketch provides the same average estimation errors and outperforms Exponential Count-Min sketch in terms of the communication cost of distributed queries.

For heavy-hitter queries, this line of sketch-based work assumes that items have integer id's in the range of $[1, n]$ and uses the dyadic range sum technique in [14] to decompose $[1, n]$ into $\log n + 1$ levels, where level l has $\frac{n}{2^l}$ dyadic ranges ($0 \leq l \leq \log n$). Then a persistent Count-Min sketch is built for each level to track the total frequencies of elements in every dyadic range. Thus, a heavy-hitter query with frequency threshold Φ fraction can be answered by recursively querying the persistent sketch from level $\log n$ down to level 0. The idea is that

there cannot be more than $\frac{1}{\Phi}$ dyadic ranges with frequency more than fraction Φ. Once we locate these frequent dyadic ranges, in their next lower level, we only query those $\frac{2}{\Phi}$ sub-divided dyadic ranges, and again there cannot be more than $\frac{1}{\Phi}$ of them being frequent. Finally in level 0, we get all individual frequent items. With dynamic data items, we may not know a tight upper bound of the number of distinct items n *a priori*. Suppose we are able to assume 32-bit integer item id's, then there are 32 levels, i.e., 32 persistent Count-Min sketches, which are very heavy in memory and processing costs. Additionally, we may have to maintain and look up a large 1-to-1 mapping table between integer id's and items if items do not have integer id's to begin with (e.g., hashtag strings). For top-k queries, again this suffers from the problem of choosing Φ, as discussed above. Furthermore, since retrieval cost is linear to $\frac{1}{\Phi}$, a very small Φ entails a large number of lookups of the persistent sketches (in addition to repeated trial and error).

Ben-Basat et al. [8] also address heavy-hitter queries in sliding windows. Their approach divides the stream into W-sized frames and further partitions each frame into k equal-sized blocks, and the window of interest is also of size W. The whole algorithm design is based on such k equal-sized blocks. However, in practice, data flow fluctuates and we could not know the "block size" in advance. The algorithm in [8] will not work for equal-sized blocks based on time. Moreover, one cannot query arbitrary window sizes. Besides what has been discussed, there is some more remotely related work in the literature. For example, Mouratidis et al. study continuous monitoring of top-k queries in data stream sliding windows [33]. There are two major differences from ours. First, it is based on a *preference function* to compute a score for top-k, but not on *frequency*. Second, its window size has to be fixed in advance and users cannot query arbitrary window sizes. Lam and Calders [24] study top-k items in a data stream with the highest max-frequency, which is defined as the maximum of the item frequency *over all window lengths*. Mirylenka et al. [30] introduce the notion of *Conditional Heavy Hitters*. This concept is distinct from prior notions of heavy hitters and frequent itemsets. It cares about items that are condition-

ally frequent: a particluar item is frequent within the context of its parent item. Then they develop several streaming algorithms for retrieving conditional heavy hitters. It solves a different problem and is not for a query with a particular window size. Shah et al. [37] compute hierarchical heavy hitters by modifing Misra Gries algorithm. In [39], a new concept called *Hierarchically Correlated Heavy Hitters* is described to capture the sequential nature of the relationship between pairs of hierarchical items at multiple concept levels and local contextual patterns within the context of the global patterns. The algorithm of [39] finds the correlation between items corresponding to hierarchically discounted frequency counts. Like [30], the algorithms of [37, 39] are not designed for top-k frequent items queries of a paricular window size. Le et al. [25] study top-k erasable pattern mining problem as a variant of frequent pattern mining and propose two efficient methods using pruning strategies and the subsume concept, which is not suitable for data streams and window queries. Erra et al. [17] present a revised *TF-IDF* measure and a parallel implementation of the calculation of the approximate *TF-IDF* based on GPUs to process continuous data streams. It returns frequent top-k items by calculating the approximate *TF-IDF* value that only works well for a fixed window size. Babcock et al. [7] propose priority-sample to sample from a timestamp-based moving window. Upon each element's arrival, it is assigned a randomly chosen priority between 0 and 1. The purpose of this work is to maintain a uniform random sample of size k. To achieve this goal, priority-sample generates k priorities $p_1, ..., p_k$ for each element and chooses the element with the highest p_i for each i. The goal of this work is different from ours and is not able to handle frequent items query. Also, we show in Section 3.2 why a simple random level assignment upon each element's arrival cannot serve our needs. The detailed analysis is shown in Section 3.3.

Datar et al. [16] address the problem of maintain aggregates and statistics over data streams. Similar to us, it works for sliding windows of any sizes chosen at query time. It solves a basic counting problem—counting the number of 1's in the last w bits of the stream (at any time), where $w \leq W$, and W

is an upper bound. It works by maintaining exponential histograms. While bearing some similarity with our Item Frequency Tracking problem, our target is fundamentally different. As shown in Definition 2, our problem is to *succinctly* track the item counts at each time unit within a very large window (imagine this can be tracked for many items at the same time). Hence we resort to very concise piecewise polynomial functions (i.e., a form of dynamic data compression). Note that this problem is also different from the work of prediction and forecasting models in statistics [28].

For this problem, the Piecewise Linear Approximation [34] also provides a solution, which is very efficient and provides guarantees on model accuracy. The popularity and wide use of [34] are attributed to its high efficiency—amortized $O(1)$ time per point [34], as well as its adaptivity and scalability to any long period of time, especially suitable for data stream settings. Our method, called Progressive Trend Model, uses different techniques, namely five-point stencil [9] to estimate a few low-order derivatives using data points with progressive step sizes (granularities), as well as Taylor series, to explore the trend, resulting in more powerful function models. We prove that its cost is also amortized $O(1)$ time per point, and our experiments show that it is 20-30 times more concise than Piecewise Linear Approximation, under the same accuracy constraint.

We have briefly discussed how to deal with the windowed top-k frequent items and the item frequency tracking problems in the ICDE 2017 poster version [40], where we show the algorithms and a few preliminary evaluation results. By contrast, in this paper, our contributions include more complete algorithms and the added theoretical analyses (which are not in [40]), along with more intuitions, explanations, and examples. In addition, in this paper, we have conducted much more experiments to systematically evaluate our work.

1.2. Our Contributions

We first formally state the Windowed Top-k Frequent Items problem, as well as the Item Frequency Tracking problem. For the first problem, we devise an algorithm called *Floating Top-k*. The basic idea of it is to let each item

in the stream perform an independent action—choosing a random "level" from a distribution, such that some aggregate value grouped by each distinct item is probabilistically proportional to the frequency of that item. Thus, we can estimate the top-k frequent items in a window based on the k items/groups with top aggregate values. The "aggregate" value here is the maximum *level* of the item, where an appearance of the item in the stream gets a random level, as stated above. An interesting property with the above procedure is that we do *not* need to explicitly maintain any item counts over time or deal with the count updates upon item entry and expiration from windows of any sizes. This is because each item in the stream simply makes its own choices (of levels) independent of any other items. Our Floating Top-k algorithms maintain a small-size *floating tuple pool*, such that the top-k items with highest maximum levels can be retrieved for any-size windows, at any time. We prove that the expected space complexity of our method is $O(k \log W)$, where k is the number of items parameter in top "k", and W is the maximum window size (in number of time units). We also prove its time complexity. The fact that the space and time costs of Floating Top-k grow only logarithmically with W is significant; all previous approaches to this problem (Filtered Space-saving with Sliding Windows and Persistent Data Sketching) grow at least linearly with W. Our experiments using three real-world datasets show that Floating Top-k retrieves very accurate top-k most frequent items. The memory footprints of Filtered Space-saving with Sliding Windows and Persistent Data Sketching are 2 to 3 orders of magnitude larger than that of our algorithms even for relatively small W, and grow linearly with W. Floating Top-k is also at least 1 to 2 orders of magnitude faster than Filtered Space-saving with Sliding Windows and Persistent Data Sketching. Thus, our method is highly scalable for high-rate data streams with dynamic items and arbitrary-size windows.

For the Item Frequency Tracking problem, we propose an algorithm called *Progressive Trend Model*. We prove that it has an amortized cost of $O(1)$ per time unit. Experiments show that it is much more concise (20 to 30 times) and slightly faster than Piecewise Linear Approximation (previous work) under the

same accuracy guarantees. Our contributions can be summarized as follows:

- We state the Windowed Top-k Frequent Items and the Item Frequency Tracking problems (Section 2).
- We propose an algorithm to effectively solve the Windowed Top-k Frequent Items problem, and prove its complexity and parameter choices (Section 3).
- We devise a Progressive Trend Model algorithm for the Item Frequency Tracking problem, and prove its amortized cost of $O(1)$ per time unit (Section 4).
- We perform a comprehensive experimental evaluation, using three real-world datasets and comparing with three methods in previous work (Section 5).

2. Problem Statement

We are given as input a data stream $s = a_1 a_2 \ldots$. Each item a_i is drawn from a highly dynamic, potentially very large domain $\mathcal{D}$. We partition the time domain into *time units*, such as seconds or minutes, depending on the window granularity in queries. For example, if the finest window granularity a user cares about is at *minute* boundaries, then a time unit is a minute. A user could ask for top-5 hot news in the past 30 minutes, but has no need in asking for the top-5 hot news in the past 30 minutes and 2 seconds. In general there can be any number of items arriving in each time unit. We first formally define the Windowed Top-k Frequent Items problem.

Definition 1. *(Windowed Top-k Frequent Items). Let the current time unit be t. The windowed top-k frequent item problem is to return the top-k most frequent items for any given time window from time unit $t - w$ to time unit t, denoted as $[t - w, t]$, where $w \leq W$ (W is an upper bound of w). This query may be asked at any time unit t.*

In general, W can be very large. For example, the time window can be seconds, minutes, hours, days, weeks, or even months. The system should maintain a data structure as concise as possible, in real time as data streams in. Then whenever a query is asked at any time unit t, the top-k most frequent items in the query window need to be returned. Let us now define the second problem.

Definition 2. *(Item Frequency Tracking Problem). For a selected item i, we maintain a concise data structure M such that for any time unit t' chosen uniformly at random from the window $[t - W, t]$, with probability at least $1 - \epsilon$, the count of item i in t' returned by looking up M has an error no more than δ fraction from the true count.*

Our goals in a solution to the Item Frequency Tracking problem are threefold: (1) it provides accuracy guarantees; (2) building M is incremental and highly-efficient for real-time operations; (3) the resulting data structure/model M should be succinct.

3. Windowed Top-$k$requent Items

3.1. Intuition

A major issue with previous methods (Filtered Space-saving with Sliding Windows and Persistent Data Sketching), as also found in the experiments (Section 5), is that they are too heavy, particularly in memory footprint, and in turn processing overhead. They typically require memory space growing linearly with the window size W (and with a large constant). Moreover, it is hard to succinctly maintain item counts over time due to continuous item's entry into and expiration from all the windows within the upper bound W.

The basic idea of our method is that each arriving item in the stream makes an independent random choice of its "levels", following a particular distribution. Then for any window, some *aggregate value* grouped by items is probabilistically proportional to the total count of that item in that window. This aggregate value is the *maximum level* assigned to that item in the window. For example,

suppose the items $(i1, i2, i3, i2, i3, i1, i2, i4)$ arrive in increasing time order. Let us say that they are given random levels (2, 3, 1, 0, 3, 1, 4, 2) respectively. $(i1, i2, i3, i4)$'s aggregate values (i.e., maximum levels) will be (2, 4, 3, 2) for a window of size 9. Then we use (2, 4, 3, 2) to give evidence of the frequency of the items approximately and obtain top-k frequent items. Hence, we may just somehow maintain a succinct data structure, called a *Floating Top-k tuple pool*, that incrementally keeps the items with top-k maximum levels for each window.

By doing this, an interesting and nice effect is that we do not need to maintain continuous counter states, and hence it is extremely simple to deal with item's entry into and exit from all windows. In other words, there is no need to explicitly handle item expiration as in previous methods. In other methods, for example, if we maintain frequencies for each item i as in the algorithm of [21], when an item expires, we must deduct the expired item counts from the total counts, resulting in a high maintenance overhead. While in our method, each tuple in the top-k tuple pool has a timestamp in the interval $[t - W, t]$ where t is the current time and W is the maximum window size. Thus, simply collecting all tuples in the pool that fall in the query window will be sufficient (note that we allow query window to be less than W). This advantage is attributed to the independent level choices of arriving items.

3.2. Main Algorithms

With the intuitions in Section 3.1, we next present the main algorithms for the Windowed Top-k Frequent Items problem. We call our method *Floating Top-k*. The first algorithm, MaintainFloatingTopK, as shown below, is for the stream system to maintain a *Floating Top-k tuple pool* Γ. As discussed in Section 3.1, this pool Γ is to store a number of items that have the highest maximum levels. Then at any time, when the user's query comes in, the system can retrieve the top-k most frequent items from the current snapshot of Γ.

The key point of the pool Γ is that it needs to be sublinear to the maximum window size W. In fact, we will show shortly in Theorem 1 that the expected size of Γ is $O(k \log W)$. To achieve this, we organize the tuples in Γ (where each

Algorithm 1: MaintainFloatingTopK (s, W, k)

Maintain a pool Γ to store a number of items that have the highest maximum levels.

Input: s: data Stream,

W: upper bound of window size,

k: upper bound of number of top results needed

Output: a continuously evolving data structure Γ

```
1  for j ← 0 to W − 1 do
2      Γ[j] ←empty list
3  foreach current time unit t do
4      for j ← W − 1 down to 1 do
5          Γ[j] ← Γ[j − 1]
6      Γ[0] ← empty list
7      foreach item i that arrives in t do
8          l ← RandomLevel(p)
9          Γ[0] maintains top-k tuples(i, l) with distinct i and highest l (i.e.,
               ranked on l in descending order)
10     ctop ← Γ[0]                          //maintain a current top-k vector
11     for j ← 1, ..., W − 1 do
12         for (i, l) ∈ Γ[j] do
13             if l ≤ ctop[k].level then
14                 remove (i, l) from Γ[j]
15                 continue
16             if ∃l′, s.t.(i, l′) ∈ ctop then
17                 if l > l′ then
18                     replace (i, l′) by (i, l) in ctop
19                 else
20                     remove (i, l) from Γ[j]
21             else
22                 replace ctop[k] by (i, l) in ctop
```

tuple contains level information l of an item i) in their *timestamp* order. Thus, if we scan the tuples in Γ in *reverse* chronological order, and maintain a vector of k tuples *ctop* that have the highest levels thus far, we can remove a tuple (i, l) from Γ if it does not have a higher level than any tuple in *ctop*. This is because (i, l) will expire from a window earlier than all tuples in *ctop* anyway; hence there is no chance for it to be in the top-k of any window from now on. We will show an example of this point after explaining the algorithm.

The input parameter k to MaintainFloatingTopK is also considered as an upper bound, since the algorithm can easily handle top-k' most frequent items for any $k' \leq k$. In lines 1-2, we initialize the Floating Top-k tuple pool Γ to be all empty. For simplicity and clarity, we present Γ as an array, where the index is time unit in *reverse* chronological order, i.e., $\Gamma[0]$ holds some *tuples* at the current time unit ("now"), and $\Gamma[j]$ holds some tuples at time unit j before "now" ($0 \leq j \leq W - 1$). Here, a tuple has the form (i, l) indicating item i with level l. It may seem that this by itself is already $O(W)$ space and time complexity. However, as we prove in Theorem 1, most entries in Γ are expected to be empty; we actually implement Γ with a linked list ordered by time, where each tuple also has a "time unit" attribute.

Lines 4-22 are the actions we perform for each current time unit t (i.e., "now"). Lines 4-5 are due to the time unit shift. Lines 6-9 first assign a random level for each item in the current time unit (t), and then get the top-k highest-level tuples from them and assign them to $\Gamma[0]$ (the RandomLevel called in line 8 will be explained in detail below). In case there are fewer than k items/tuples at time t, we just keep all of them. Note that we need to keep all these tuples (at most k) in $\Gamma[0]$, because they are the "newest" (i.e., latest). If one were to query the top-k now with window size 1, they will be returned. Intuitively, "newer" tuples in the pool are "better" as they expire later. Hence, in lines 10-22, we iterate through the tuples in reverse chronological order.

Line 10 initializes the "current" top-k vector *ctop*. The loop from line 11 considers what will be in top-k if the query window is of size j. Line 12 iterates through all tuples (with timestamp $t - j$) that are currently in the pool. In line

13, if such a tuple (i, l) has level l not better than the lowest level in *ctop* (tuples in *ctop* are in descending order of levels), we just remove this tuple from the pool (lines 14-15). Otherwise, the condition in line 16 indicates that there exists a tuple in *ctop* that is from the same item (i.e., i) as this tuple (i, l). Then we must only keep one of them in *ctop* (lines 17-20)—the one with higher level (in case it is a tie, we keep the newer one). This can be easily implemented by searching for the item i from $ctop[0]$ to $ctop[k]$ and by doing a simple insertion sort. In this way, we update *ctop* and maintain its descending order. The reason we only record the highest level for each item is because the resulting top-k items are ordered by their maximum levels. We will show the relationship between the expected maximum level of an item and its true counts in Section 3.3. In a nutshell, an item with a higher count is expected to have a greater maximum level. If (i, l) does not make it into *ctop*, we remove it from Γ (as it will never be from now on). In lines 21-22, the tuple (i, l) will replace the one in *ctop* with the lowest level (i.e. the last element in *ctop*). The resulting *ctop* then maintains the top-k items for the largest window size W. Finally, the continuously evolving tuple pool Γ is exposed to top-k queries at any time (i.e., will be an input to the RetrieveTopK algorithm discussed below).

Recall that the algorithm RandomLevel is called in line 8 MaintainFloatingTopK; we present this simple algorithm. The choice of its input parameter p will be discussed in Section 3.4. Line 1 of RandomLevel picks a real value m from the interval $[0, 1)$ uniformly at random. Then lines 2-9 implement an iterative procedure: starting from level $l = 0$ (line 2), we stop and return l as the level value with probability $1 - p$. That is, with probability p, we advance to the next level $l = 1$ (line 9), and continue this recursive probabilistic level promotion again, until it stops and a level l is returned (line 6). A greater m implies a greater level l. This helps us to get an aggregated maximum level of an item which is probabilistically proportional to the total count of that item (i.e., calling this algorithm more times tends to result in a higher m and hence higher level in at least one call). In addition, our overall method can conveniently re-construct the top-k tuples in a window of any size and for sliding

Algorithm 2: RANDOMLEVEL(p)

Get a random level of an item.

Input: p : probability of level promotion

Output: a random level

```
1 m ← random(0, 1)
2 l ← 0
3 q ← 1
4 while true do
5     if m < q(1 − p) then
6         return l
7     m ← m − q(1 − p)
8     q ← qp
9     l ← l + 1
```

windows. The detailed analysis is shown in Section 3.3. By contrast, a simple random level assignment upon each item's arrival cannot serve this purpose.

Next we show the algorithm RETRIEVETOPK, which returns the top-k most frequent items for any window of length $w \leq W$, i.e., in the window $[t-w+1, t]$, based on the tuple pool structure Γ that is built and continuously maintained. As discussed earlier, k is an upper bound (used for building Γ); we can answer top-k' queries ($k' \leq k$) too, through replacing k by k' in RETRIEVETOPK.

Recall that, in the Floating Top-k tuple pool Γ, the list of tuples at a particular time unit, $\Gamma[u]$ ($0 \leq u \leq w-1$), is sorted in level-descending order. Thus, RETRIEVETOPK is essentially a multi-way merge sort, until we get k tuples with highest levels. This can be done by using a max-heap structure (priority queue) in line 1, starting the heap with only the first tuples at each non-empty list $\Gamma[u]$. Then we keep popping the top-one from the heap (and add the next one from its $\Gamma[u]$ list into heap), until we have k of them.

Example 3. *We use a simple example to illustrate how* MAINTAINFLOATING-TOPK *works. At the top of Figure 1 we show the data stream in increasing time order (time units* $t0, t1, \ldots, t8$*), together with the items* $(i1, i2), (i2), (i1), (i1)$*,*

Algorithm 3: RETRIEVETOPK (Γ, w, k)

Obtain top-k frequent items within the most recent window of the given size based on the tuple pool that is built and continuously maintained.

Input: Γ : Floating Top-k structure,

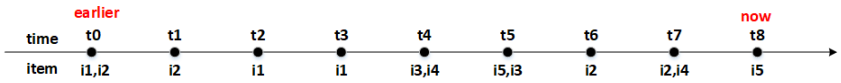
w : query window size,

k : number of top results needed

Output: top-k frequent items in the window $[t - w + 1, t]$

```
1  build a max-heap H (sorted on level) using each first tuple in Γ[0], ..., Γ[w − 1]
2  R ← empty list
3  for j ← 1...k do
4      while true do
5          pop max from H; let it be Γ[u][v] = (i, l)
6          if i is not in R then
7              break
8      append Γ[u][v] to R
9      if Γ[u][v] is not the last element in Γ[u] then
10         push Γ[u][v + 1] into H
11 return R
```

$(i3, i4), (i5, i3), (i2), (i2, i4), (i5)$ *(only one or two items per time unit for simplicity). Time* $t8$ *is the current time unit ("now"). Suppose* $W = 9$ *and* $k = 2$*. Each row in the table in Figure 1 shows the tuples in the Floating Top-k tuple pool* Γ *at the corresponding time unit. In a tuple at each column, the second value is the random level that item is assigned by* RANDOMLEVEL*. Let us pick two time units to see how the algorithm works. First consider* $t = t3$ *and see how* Γ *changes from* $t = t2$ *to* $t = t3$*. At* $t3$*, item* $i1$ *arrives with a random level* 4*. Lines 6-9 of* MAINTAINFLOATINGTOPK *assign* $\Gamma[0]$ *to be* $(i1, 4)$ *(there are fewer than* $k = 2$ *tuples; but that's all we have). Then lines 10-22 iterate through the existing tuples in* Γ *(inherited from row* $t = t2$*) in reverse chronological order. First, for tuple* $(i1, 0)$*, the same item* $i1$ *already exists in the current*



Γ

	t0	t1	t2	t3	t4	t5	t6	t7	t8
t=t0	{i1, 2} {i2, 0}								
t=t1	{i1, 2}	{i2, 3}							
t=t2	{i1, 2}	{i2, 3}	{i1, 0}						
t=t3		{i2, 3}		{i1, 4}					
t=t4		{i2, 3}		{i1, 4}	{i3, 2} {i4, 5}				
t=t5		{i2, 3}		{i1, 4}	{i3, 2} {i4, 5}	{i5, 3} {i3, 0}			
t=t6				{i1, 4}	{i3, 2} {i4, 5}	{i5, 3} {i3, 0}	{i2, 4}		
t=t7					{i4, 5}	{i5, 3}	{i2, 4}	{i2, 3} {i5, 2}	
t=t8					{i4, 5}	{i5, 3}	{i2, 4}	{i2, 3} {i5, 2}	{i5, 1}

Figure 1: A simple example to illustrate the process of maintaining the floating pool, where $W = 9$ and $k = 2$.

top-kctop in $(i1, 4)$*; hence* $(i1, 0)$ *is removed from* Γ *(line 20). The same is true for tuple* $(i1, 2)$*, while* $(i2, 3)$ *joins ctop and stays in* Γ*. Now consider the row* $t = t7$*. Item* $i2$ *arrives with level* 3 *and* $i5$ *arrives with level* 2*. ctop includes* $(i2, 3), (i5, 2)$*. For tuple* $(i2, 4)$*, item* $i2$ *is in the current ctop; hence* $(i2, 3)$ *is replaced by* $(i2, 4)$ *(line 18). Similarly,* $(i5, 3)$ *replaces* $(i5, 2)$ *in ctop. Tuple* $(i3, 0)$ *is removed from* Γ *because its level is less than the current top-2 in ctop (line 20). The same is true for tuple* $(i3, 2)$*. Then the next tuple,* $(i4, 5)$*, however, replaces* $(i5, 3)$ *in ctop (line 22). Finally, tuple* $(i1, 4)$*, whose level is equal to* $ctop[k]$*, is removed from* Γ*. Suppose at this moment a top-2 query is asked with* $w = 9$*, then* RetrieveTopK *will return* $i2$ *and* $i4$*. If the query is asked with* $w = 3$*, then* RetrieveTopK *will return* $i2$ *and* $i5$*.*

We next perform a detailed analysis of our Floating Top-k algorithm framework in Section 3.3.

3.3. Analysis

In this subsection, we first analyze the time and space complexity of our main algorithms. Then we analyze the accuracy of the finally retrieved top-k items. For complexity, we first show that the number of (i, l) tuples maintained in the Floating Top-k structure Γ has a nice bound. We first show the following result.

Theorem 1. *The expected number of tuples maintained in the Floating Top-k tuple pool is* $O(k \log W)$.

Proof 1. *For the current time unit* t, $\Gamma[0]$ *stores* k *tuples.* $\Gamma[1]$ *stores the additional* (i, l) *tuples that go into the top-k in the window* $[t-1, t]$ *(in addition to a few top ones in* $\Gamma[0]$*). Since the choices of levels are random and independent for each item in the stream, we have* $\boldsymbol{E}[|\Gamma[1]|] = \frac{k}{2}$*, i.e., in expectation, half of the* k *highest levels come from* $\Gamma[1]$*, and the other half from* $\Gamma[0]$*. Similarly,* $\boldsymbol{E}[|\Gamma[2]|] = \frac{k}{3}, ..., \boldsymbol{E}[|\Gamma[W-1]|] = \frac{k}{W}$*. Hence, the expected size of the Floating Top-k tuple pool is:*

$$\begin{aligned}\boldsymbol{E}[|\Gamma|] &= \boldsymbol{E}[\sum\nolimits_{i=0}^{W-1} \Gamma[i]] = \sum\nolimits_{i=0}^{W-1} \boldsymbol{E}[|\Gamma[i]|] \\ &= k \cdot \sum\nolimits_{i=0}^{W-1} \frac{1}{i+1} = O(k \log W)\end{aligned}$$

Note that in case the stream is so slow that there are fewer than k *tuples per time unit, we can just merge multiple time units into one "large unit" that has sufficient tuples. There will be fewer than* W *large units in the pool and* $\boldsymbol{E}[|\Gamma|]$ *can only be smaller.*

We now have the complexity results.

Theorem 2. *The time complexity is* $O(k \log k \log W)$ *per time unit for* MaintainFloatingTopK*, and the total space complexity is* $O(k \log W)$ *in expectation. The expected time and space complexity of* RetrieveTopK *is* $O(k \log w)$.

Proof 2. *Firstly, the call* RandomLevel(p) *in line 8 takes* $O(\frac{1}{1-p})$ *time, since the loop in lines 4-9 of the algorithm* RandomLevel *can be considered as a geometric distribution with success (i.e., stop of the loop) probability* $1-p$ *in each loop. Thus, the expected cost is* $O(\frac{1}{1-p})$*, which is a constant (we will discuss the choice of* p *shortly). Also note that the random levels returned by* RandomLevel *can be easily precomputed offline using a pseudorandom generator.*

Secondly, from Theorem 1, the loops in lines 11-12 of MaintainFloatingTopK *examines an expected number of* $O(k \log W)$ *tuples* (i, l) *in the pool* Γ.

Within the loop (lines 13-22), each line takes $O(1)$ time, except lines 18 and 22 where we "replace" a tuple in ctop by (i, l), which takes $O(\log k)$ time, as we need to search to maintain the sorted order of ctop. Note that line 16 takes $O(1)$ time too, as the match of item i can be done using a hash table. Thus, the expected time complexity per time unit is $O(k \log k \log W)$. It is also clear that the expected space complexity is $O(k \log W)$ for storing the top-k tuple pool.

Finally, the heap building in line 1 of RetrieveTopK *takes an expected $O(k \log w)$ cost in both time and space (again from Theorem 1 for the size of the floating tuple pool). Note that we use "w" for the actual window size in the query. Thereafter, the heap operations in lines 3-10 take $O(k \log(k \log w))$ time. Hence the overall expected cost in time and space of* RetrieveTopK *is $O(k \log w)$.*

Now let us analyze the relationship between an item's frequency and its maximum level.

Theorem 3. *For any query window $[t - w, t]$, the expected maximum level of an item i returned by* RandomLevel *is $\boldsymbol{E}[L] = \sum_{r=1}^{\infty}[1 - (1 - p^r)^n]$, where n is the number of times i appears in that window, and r is an integer.*

Proof 3. *$L = \max_{j=1,\ldots,n} L_j$, where L_j is the level of item i at its j's appearance. Since the level L of the item i is a random variable that takes on only non-negative integers, it follows that*

$$\begin{aligned}
\boldsymbol{E}[L] &= \sum_{r=1}^{\infty} Pr(L \geq r) \\
&= \sum_{r=1}^{\infty} Pr(\max_{j=1,\ldots,n} L_j \geq r) \\
&= \sum_{r=1}^{\infty} [1 - \prod_{j=1}^{n} Pr(L_j < r)] \\
&= \sum_{r=1}^{\infty} [1 - \prod_{j=1}^{n} [1 - Pr(L_j \geq r)]] \\
&= \sum_{r=1}^{\infty} [1 - (1 - p^r)^n]
\end{aligned}$$

Note that the first equality is true since L is a non-negative integer [32]. Intuitively, $Pr(L \geq r)$ is the probability that we can add 1 to L, iteratively for increasing r.

From Theorem 3 we can see that, the greater n is (i.e., the more frequent item i appears in the query window), the greater $\mathbf{E}[L]$ is. Theorem 3 does not give a closed form solution. We have the following result.

Theorem 4. *An unbiased estimate of the number of appearances of an item i in a query window, given its maximum observed level l returned by* RandomLevel*, is $\frac{p+1}{2p}(\frac{1}{p})^l - \frac{1}{2}$.*

Proof 4. *We define a random variable X as the number of times item i appears in a time window when its maximum level first reaches value l. Then we define a random variable Y as the number of times item i appears in the time window when its very next appearance increases its maximum level to $l+1$. The event that item i's level reaches value l has a probability p^l. Thus, X follows a geometric distribution with parameter p^l. We then have $\boldsymbol{E}[X] = \frac{1}{p^l}$. Similarly, $Y+1$ follows a geometric distribution with parameter p^{l+1}. Thus $\boldsymbol{E}[Y+1] = \frac{1}{p^{l+1}}$ and $\boldsymbol{E}[Y] = \frac{1}{p^{l+1}} - 1$. As we observe that the maximum level is l when the query is issued, the actual number of times N that item i appears must satisfy $X \leq N \leq Y$. Since N is equally likely to be any value in that interval, we have an unbiased estimate $N = \frac{X+Y}{2}$, and hence $\boldsymbol{E}[N] = \frac{\boldsymbol{E}[X]+\boldsymbol{E}[Y]}{2} = \frac{1}{2}(\frac{1}{p^l} + \frac{1}{p^{l+1}} - 1) = \frac{p+1}{2p}(\frac{1}{p})^l - \frac{1}{2}$.*

We now further show the order-preserving property in Theorem 5.

Theorem 5. *For a given query window, the probability that a frequent item i is not reported by* RetrieveTopK *is no more than $e^{-\mu/2}$, where $\mu = \frac{n_i}{n_k}$, n_i is the count of item i, k is the upper bound parameter used in* MaintainFloatingTopK*, and n_k is the count of the item ranked the k'th in the ground truth.*

Proof 5. *Let the maximum level of the item ranked the k'th in the query window*

be τ. Then for the j'th appearance of item i, define a random variable

$$X_j = \begin{cases} 1 & \textit{if its level} \geq \tau \\ 0 & \textit{otherwise} \end{cases} \quad (1 \leq j \leq n_i),$$

giving n_i random variables. Thus, $Pr(X_j = 1) = p^\tau$, and $\boldsymbol{E}[X_j] = p^\tau$. Further define $X = \sum_{j=1}^{n_i} X_j$. From linearity of expectation,

$$\boldsymbol{E}[X] = \sum\nolimits_{j=1}^{n_i} \boldsymbol{E}[X_j] = n_i p^\tau \tag{1}$$

On the other hand, $n_k p^\tau \approx 1$, as τ is the maximum level of the rank-k item (There are one or more occurrences of the rank-k item at level τ by definition. Thus, one can show a probabilistic lower bound of $n_k p^\tau$ close to 1. Note that a greater $n_k p^\tau$ only helps our Chernoff bound in the proof below in the right direction). Combining this with Equation (1) gives

$$\boldsymbol{E}[X] = \frac{n_i}{n_k} = \mu \ (\mu \textit{ is as defined in the theorem})$$

Then from Chernoff bound [32],

$$Pr(X < 1) = Pr(X < [1 - (1 - \frac{1}{\mu})]\mu) \leq e^{-\mu(1-\frac{1}{\mu})^2/2} \approx e^{-\mu/2}$$

as we consider frequent item i and hence large μ; the above probability is small. Note that μ grows exponentially as we slightly increase k (the upper bound that determines the cost of the top-k tuple pool). Finally, from the definitions of X_j and X, $Pr(X < 1)$ is the probability that item i has never had level at least τ in its n_i appearances in the window, and hence is not reported by RetrieveTopK*, finishing the proof.*

Theorem 5 shows that as the frequency ratio $\mu = \frac{n_i}{n_k}$ increases (i.e., item i is much more frequent than the item ranked at the k'th), the probability that item i is not reported in top-k decreases exponentially fast.

We may further enhance the accuracy of the top-k results by reducing the variance through independently repeating MaintainFloatingTopK and RetrieveTopK a constant number of times. Suppose it is repeated c times,

which we call c rounds, and we get $c \cdot k$ item-level pairs (i, l). For each distinct item i that appears in these results, we first use Theorem 4 to calculate the estimated counts in each of the c rounds. It is possible that i only appears in some rounds but not others. For a round that misses i, we use the *minimum level of that round* $- 1$ as the level l in Theorem 4 to estimate the count. Then, removing the maximum and the minimum (to drop possible outliers), we get the *mean* of the remaining $c-2$ counts as the final count of item i. In the end, we select top-k items based on these final counts. Although repeating MaintainFloatingTopK and RetrieveTopK c times increases accuracy, it consumes more computation resources. We show the effect of varying c in Section 5.

3.4. Choice of Parameters

We now discuss the choice of parameter p, i.e., the promotion probability in choosing random levels for each item. The intuition is that we should ensure that the maximum levels of top-k items should be spread out in a sufficiently large range (e.g., $2k$), so as to be robust against variations of item levels. We have the following result.

Theorem 6. *Let the total number of items in a time window be n, and the number of the item with rank k be n_k (which can be estimated at runtime as discussed below). Then we should choose the parameter p as $p = (\frac{n_k}{n})^{\frac{1}{k'}}$, where k' is the desired number of levels that top-k items span across.*

Proof 6. *First consider all items together, and let the maximum level in the time window be l. Then from Theorem 4, we have*

$$n \cong \frac{p+1}{2p}(\frac{1}{p})^l - \frac{1}{2} \tag{2}$$

Since k' is the desired number of levels that top-k items span across, the item with rank k should have the maximum level $l - k'$. Again from Theorem 4, we have

$$n_k \cong \frac{p+1}{2p}(\frac{1}{p})^{l-k'} - \frac{1}{2} \tag{3}$$

Dividing Equation (2) by Equation (3), and simplifying, we have $p = (\frac{n_k}{n})^{\frac{1}{k'}}$ *as in the theorem.*

For example, if we first collect the statistics that n=100,000 and n_k=3,000 (k=5). n_k can be estimated by using any p value first (e.g., p=0.5). Suppose the desired $k' = 2k = 10$. Then Theorem 6 advises us to choose $p = 0.03^{0.1} = 0.7$. Note that we cannot indefinitely increase k', as Theorem 6 shows that a greater k' implies a greater p value; yet this would increase the cost of the RandomLevel algorithm invoked for each item, which has an expected cost of $O(\frac{1}{1-p})$.

Finally, we note that the choice of p may be dynamic. Based on the estimated n_k and n at runtime for window size W, we can dynamically adjust p.

4. Concise Real-Time Tracking of Item Frequency

In the previous section, we have studied how to locate top-k most frequent items in any time window. As discussed in Section 1, there are also many usage scenarios where a user knows which item(s) he/she is interested in. For example, in interactive data exploration, after querying top-k frequent items, the user may want to individually track the frequency evolvement of some of these items. As time goes by, some of them may not be "hot" any more, and the user would start another data exploration cycle—querying top-k followed by tracking individual items. This process may go on repeatedly. Thus, our proposed techniques in this section complement those in Section 3, and are needed in a comprehensive data exploration system.

Imagine that one may want to track a great number of individual items at the same time. Thus, our goal in solving the Item Frequency Tracking problem is *concise* and *efficient* (real-time) tracking of the historical frequency counts of a selected item over a time window, subject to the accuracy requirement. We first give some intuition and background. To track the frequency of an item over a large window, it is intuitive and broadly applicable to use *piecewise* functions, since most likely no single function can fit the whole large window. The question

595 is what function we use for each "piece", so that it is not only concise, but also efficient, while guaranteeing accuracy.

Background. We devise a novel solution by using *five-point stencil* [9] to estimate some low-order derivatives from data, combined with Taylor series models. In numerical analysis, the five-point stencil of a point x in one di-
600 mension consists of the point itself together with its four "neighbors", i.e., $\{x-2h, x-h, x, x+h, x+2h\}$, where we call h the *step size* between the points. Knowing the function values $f(\cdot)$ at these five points, one can approximate a number of low-order derivatives of $f(\cdot)$ at point x. For example, $f'(x) \approx \frac{-f(x+2h)+8f(x+h)-8f(x-h)+f(x-2h)}{12h}$. Likewise, there are formulas [9] to
605 approximate $f''(x)$, $f^{(3)}(x)$, and $f^{(4)}(x)$, etc. Once we have these derivatives, we use the Taylor series at x as a model function.

One novel aspect of our solution is to progressively increase the step size h, to explore a suitable one so that the item frequency and change patterns (i.e., speed $f'(t)$, acceleration $f''(t)$, etc.) are well represented, and the Taylor series
610 model can cover as many data points (i.e., time units) as possible. That is, **we maximize the coverage of a "piece" in the piecewise model**. We use two auxiliary algorithmic constructs "chains" and "rounds" to achieve this. The rationale of this point is illustrated in Figure 2(a), where time "zero" is the center point of five-point stencil. The five point stencil inside the dashed oval
615 corresponds to the initial step size $h = 1$. Due to its limited scope, the function that it produces can only be close to a horizontal line, which would not cover a big range in time. In our method, we iteratively (but efficiently) explore other step size h values. For example, the center (zero) point plus the four points outside the dashed oval is another five-point stencil that possibly results in a
620 better function that covers more points. We now present the PROGRESSIVETRENDMODEL algorithm.

Figure 2(b) illustrates some high-level notions in the algorithm. The time line is partitioned into *epochs*. Each epoch in the end has one function (Taylor series) as part of the whole model. Each epoch has a "zero" point in time as its
625 first data point. We use the terms data "point" and "time unit" interchangeably.

Algorithm 4: ProgressiveTrendModel (s, i)

Return the built model that tracks the frequency of a given item in a data stream.

Input: s : data stream,

i : item being tracked

Output: a model

```
1  F ← φ; zero, front, front_p ← s.now(); chain ← 0; h ← 1/2; f, v ← null;
2  while s.now() is in the window where i is tracked do
3      while f = null do
4          chain ← chain + 1; h ← 2h
5          if chain > α then
6              chain ← 1
7              if front − zero ≥ 2(front_p − zero) then
8                  front_p ← front                          //advance to next round
9              else
10                 h ← 1                                    //start a new epoch
11                 zero, front, front_p ← s.now()
12                 F ← F ∪ f_e
13         f ← BuildModel(zero, h)
14         if Verify(f, zero) = false then
15             f ← null
16     if v = null then
17         v ← s.next(i)
18     if Match(f, v) then
19         v ← null
20     else
21         front ← s.now() − 1
22         f_e ← f                    //must be the best so far in this epoch
23         f ← null
24 return F
```

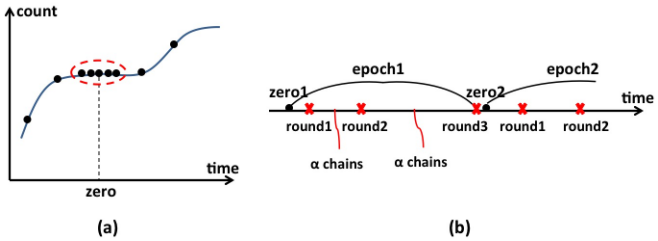


Figure 2: (a) Five-point stencil at different step sizes (granularities). (b) Some high-level notions in the ProgressiveTrendModel and the progress of the entire algorithm.

As discussed earlier, the so-called *zero* point is just the center of a five-point stencil and where the Taylor series is at. An epoch comprises a number of *rounds*, and the change of rounds keeps track of how our models are progressively extended. For example, in Fig. 2 (b), the end of round 1 implies that there is a function f_1 that could cover all data points in round 1 but not another point outside round 1. The end of round 2 implies another function f_2 that could cover all data points from the zero of this epoch to the end of round 2. Thus f_2 is better than f_1 for covering more data points. Then each round comprises α *chains*. Here α is a small integer parameter that provides a tradeoff between model conciseness and performance (a greater α may result in a more concise model, but with higher computation cost). Within one round, we try different step sizes α times to build a model. In this work, we use $\alpha = 3$ as the default; in the experiment section we explore the effect of different α values. Within an epoch, from one round to the next, the progress (i.e., distance to the epoch's zero point) at least doubles.

The input parameter s is the data stream in a natural chronological order and i is the item being tracked. We scan s in time unit order and get the counts of i. Line 1 of the algorithm initializes the function set F to be an empty set. At the end of the algorithm, the constructed F will be returned as the whole model M that is referred to in Definition 2. In line 2, $now()$ function is called on the stream s, which returns the timestamp (time unit number) of the current time unit in the stream. Initially, we assume this is positioned at the first time unit in the window where item i is tracked. Variable *zero* indicates the center of a

five-point stencil and beginning of an epoch, as discussed earlier. Variable $front$ is the frontier data point of the current *round* (recall Figure 2b), while $front_p$ is that of the previous round in the same epoch (if any). Lines 3-4 initialize variables chain as the *chain* number in the current round, h as the step size of five-point stencil (always a power of 2), f as the current Taylor series function, and v as the count of item i in the next time unit, used to explore the frontier of the current round.

The loop at lines 5-26 handles all time units in the window where i is tracked. Line 8 checks the condition when all α chains in the current round have been tried (each chain corresponds to one setting of the step size h; from one chain to the next in the epoch, h doubles). Line 10 checks the condition whether the progress of this round at least doubles, compared to the previous round. If so, we advance to the next round in the same epoch (line 11). Or else we start a new epoch (lines 12-15), and add f_e to set F as the function of the previous epoch, where f_e stores the best function of this epoch (i.e., matching all data points there).

The function BUILDMODEL in line 16 uses the values (i.e., counts of item i) at time units $zero - 2h$, $zero - h$, $zero$, $zero + h$, $zero + 2h$ and five-point stencil to estimate the derivatives, and then returns the Taylor series at $zero$ as the function. By default, we use up to the third derivative in this work. Note that in case $zero + h$ or $zero + 2h$ are in the future (i.e., after $s.now()$), $s.next(i)$ (which returns the count of item i in the next time unit) will be called to advance the stream until $zero + 2h$ is reached.

Next, the VERIFY function in line 17 verifies if the model f can accurately approximate the time units between $zero$ and $s.now() - 1$. Suppose there are t time units in this interval. That is, **for at least $(1 - \epsilon)t$ time units, the error is no more than δ fraction**. If so, VERIFY returns $true$; otherwise $false$. This error budget is similarly true for the MATCH function in line 21. For the interval between $zero$ and $s.now()$, if the error budget has not been used up, MATCH returns $true$. Parameters ϵ and δ are from the accuracy requirement of the user. Intuitively, a smaller ϵ and a lower δ give higher accuracy, which

will, in turn, result in a larger model size. We also show this in Section 5.

If verification fails (i.e., f is not as good as the best one in the epoch), we set f to null (line 18) and advance to the next chain. Once we get a function f that can advance the stream, lines 19-20 ensure that v contains the next data value to be covered by the model. Once v is matched (by the current function), it is set to null (lines 21-22). Otherwise, the frontier of the round is the previous time unit, and we store the best function in the epoch so far into f_e (lines 23-26). In the end, the set of functions F is returned. The following example illustrates how this algorithm works.

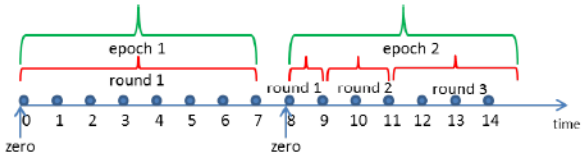


Figure 3: An example to illustrate the levels of evolution (i.e., chains, rounds, and epochs) and how ProgressiveTrendModel works.

Example 4. *In Figure 3, each tick on the time axis indicates a data point as we track item i. Let the current time window start from time 0—initially zero, $front$, $front_p$ and $s.now()$ are all at time 0. In the first try, the chain number is 1, and the step size h is 1. As zero is the start of the current window, the points used for the five-point stencil are at times $(-2, -1, 0, 1, 2)$. Denote the resulting function as $f_{(2)}$, for $s.now()$ is time 2 when this is done. We verify if $f_{(2)}$ can accurately model the item counts (of i) between times zero and $s.now() - 1$ (which is trivially true right now). Then we sequentially check the value (v in line 20) of the subsequent data points until adding one more would exceed the error budget of using $f_{(2)}$; suppose this happens at time $s.now() = 5$, and $front$ is set to $s.now() - 1 = 4$. The recorded current best model is $f_{(2)}$. Next h is doubled to 2 for the second try in the current round. The points used for five-point stencil are at times $(-4, -2, 0, 2, 4)$, giving a function $f_{(4)}$. Again the verification is done for the time points from 0 to 4. If this passes, one data point a time is retrieved from the stream and checked against $f_{(4)}$. Suppose the error*

bound is exceeded at time 8. *Then* $front = 7$ *and* $f_{(4)}$ *is the current best model of epoch 1. Likewise, in the third try,* $h = 4$, *and data points at* $(-8, -4, 0, 4, 8)$ *are used to get* $f_{(8)}$. *Suppose* $f_{(8)}$ *fails the verification. As the number of chains exceeds* α *(default 3), we advance to the next round with chain reset to 1 and* $front_p = 7$. *In the second round,* h *is doubled to* 8, *and we get function* $f_{(16)}$. *After another three tries, suppose the progress of this round does not double that of the previous round (lines 12-15); then a new epoch is started, for which zero is set to time* 8, *and* h *is reset to* 1. *Epoch 2 in Figure 3 illustrates a possible progress with three rounds. This process continues. In all, the algorithm finds a single best model for each epoch which satisfies the accuracy guarantee, while chains and rounds are used to progressively extend the epoch size, covering as many data points as possible.*

We next prove that this algorithm is very efficient.

Theorem 7. *The amortized cost of* ProgressiveTrendModel *is* $O(1)$ *per time unit.*

Proof 7. *We use an accounting argument. Recall that Figure 2(b) shows the progress of the* ProgressiveTrendModel *algorithm. Let us consider each epoch alone. Whenever a data point in the stream is first encountered (within some chain of some round), we pay an amount of* 3α *in advance. We show that this payment scheme is enough for covering all costs in the algorithm. In general, suppose a data point (time unit* t*) is first encountered in round* r *of an epoch. In Figure 2(b), time unit* t *must be outside round* $r - 1$ *for it to be first encountered by round* r. *The* 3α *that we pay for this data point is allocated as follows:*

(1) α *is for the cost either to* Verify *(line 17) or to* Match *(line 21) this time unit* t *by the* α *chains in round* r.

(2) Another α *is for the cost to* Verify *(line 17) the "buddy" time unit* t' *of time unit* t *(if any) by the* α *chains in round* r. *The buddy* t' *is an "old" time unit that has been encountered in a previous round; it is of the same distance to*

the "zero" point of the epoch as the distance from t to the end of round $r-1$. From line 10, it follows that, except for the case that r is the last round of an epoch, any "old" time unit (t') within round $r-1$ has a buddy (t) which is newly encountered in round r. Hence, the actual cost of verifying the "old" time units in round r is all accounted for.

(3) The last α in our payment for time unit t is to take care of the last round of an epoch. In the worst case, all α chains of the last round merely verify all "old" data points D encountered in previous rounds, but are not able to match any new data points. Hence, no extra payment is made by our scheme, but we need to cover the cost C of verifying all data points D α times. C is taken care of by the leftover payment α (i.e., the last α as said in this paragraph) from each data point in D when it was first encountered.

As discussed earlier, α is a (small) constant; thus the amortized cost is $O(1)$ per time unit.

5. Experiments

We perform a comprehensive empirical evaluation, using three real-world datasets and comparing with three previous methods that are most relevant. Through experiments, we answer the following questions:

- How does our Floating Top-K method compare with competing methods Persistent Data Sketching (PDS) [41] and Filtered Space-saving with Sliding Windows (FSW) [21] in terms of the quality of the top-k items found?
- How do they compare in memory usage, and processing overhead under the three real datasets and various parameter settings such as window sizes?
- What are the observed Floating Top-k tuple pool sizes?
- For tracing individual items, how does our algorithm Progressive Trend Model (PTM) compare with the previous approach of *piecewise regression*

(i.e., Piecewise Linear Approximation (PLA) [34]) in the size (conciseness) of the produced models under the same accuracy constraint?

- How do PTM and PLA compare in processing overhead?

5.1. Datasets and Experiment Setup

We use the following three real-world datasets:

- **Kosarak data.** This dataset contains the anonymized click stream of a Hungarian online news portal [2]. Each click *session* (in chronological order) contains a number of (news) items, each of which is identified by an integer. There are 8,019,015 items in total and 41,269 distinct items. This dataset was also used in previous work [24].

- **Twitter data.** We use the Twitter Stream API [1] implemented by twitter4j [5] to retrieve real-time Twitter streams from Wednesday October 28, 2015 to Thursday December 10, 2015. On average, we get 10's of hashtags per second. Each item of the stream contains a timestamp (Unix time) and a hashtag (many hashtags may have the same timestamp). Clearly the string hashtags are highly dynamic and we do not know the number of distinct hashtags in advance.

- **World-Cup data.** It contains all the requests made to the 1998 World Cup web site between April 30, 1998 and July 26, 1998 [3]. There are 1,352,804,107 requests in total. We are mainly concerned with two attributes: *timestamp* of a request (Unix time) and the *objectID*, a unique integer identifier for the requested URL. The mappings between the integers and URLs are 1-to-1 and preserved across the entire dataset.

We implement all the algorithms in the paper in Java. In addition, we implement three most relevant competing methods: (1) Persistent Data Sketch (PDS) [41], (2) Filtered Space-saving with sliding Windows (FSW) [21], and (3) piecewise regression (Piecewise Linear Approximation, PLA) [34]. Moreover, we implement (4) the FREQUENT algorithm (i.e., Algorithm 1) in [13] to find

out the ground truth top-k items in any static fixed window after knowing that window is used in comparison, in order to compare the quality of top-k items found by each competing method. All experiments are performed on a machine with an Intel Core i7 2.50 GHz processor and an 8GB memory.

5.2. Experiment Results

In the first set of experiments, we compare our Floating Top-k method with the most relevant two competing methods, PDS [41] and FSW [21] in solving the Windowed Top-k Frequent Items problem as defined in Section 2. In this comparison, we mainly study three aspects: (a) the quality of the top-k items found, (b) the processing overhead in time, and (c) the memory footprint.

For (a), the quality of top-k items found in a window, a reasonable metric is *top-k strength*, which is the ground-truth total count (i.e., strength) of the k items selected by a method for a window. The true counts of the k selected items are easy to get in the experiments, since whichever window is used in the query, after knowing it, we can simply use k counters and do another pass over the dataset to get the true counts. Thus, the top-k strength metric indicates the *quality* of the selected top-k items in a window, and can be used to compare various competing methods.

In the same vein, in the experiments, after knowing which window is selected by a query, we can simply re-play the dataset and use the FREQUENT algorithm (i.e., Algorithm 1 in [13]) to efficiently find out the *ground-truth top-k items* in that window. The aforementioned top-k strength of a method can be compared against this ground-truth top-k items' strength. In Figure 4, we show the top-k strengths of our method (Floating Top-k), together with those of the ground-truth top-k items and of previous methods PDS and FSW, using the Kosarak data. In Figures 5 and 6, we also show the stream system throughput (indicating the overhead of maintaining and processing data for top-k queries), and memory footprint for these methods over various maximum window sizes W (ranging from the past 3.6K sessions to the past 360K sessions). For Floating Top-k, each session is a time unit; we use parameter $p = 0.79$ as determined by Theorem 6

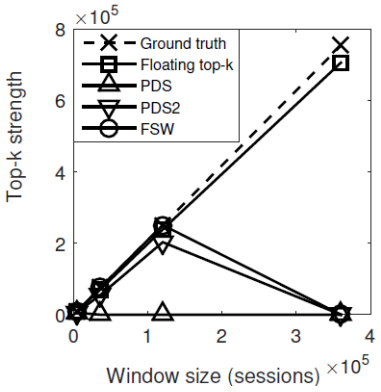


Fig. 4 Top-k strengths of Floating Top-k, PDS, FSW, and ground-truth top-k items for different window sizes on Kosarak

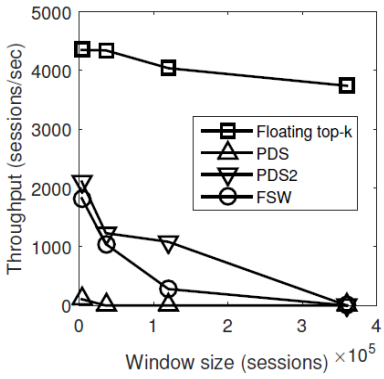


Fig. 5 Stream system throughputs of Floating Top-k, PDS, and FSW for different window sizes on Kosarak

(from a run over an initial segment of the stream), and use 20 rounds ($c = 20$) by default. In a later experiment, we also study the effect of varying c values.

In PDS, we consider the real-time dynamic stream setting where we do not know the exact number of distinct news items in advance, but assume that the ids are integers and hence an upper bound is 2^{31}. In PDS2 of the figures, we assume the static scenario in which we know the exact number of distinct news items in advance, which is 41,269 as we know from the dataset. Recall that answering top-k queries with PDS uses *historical window heavy hitter queries* (Section III-B in [41]), which resort to the *dyadic range sum technique* in [14] to recursively sub-divide the id range $[0, n]$ over $\log n + 1$ levels, and in each level, each interval is divided into two. Thus, PDS must know n first. In PDS of Figure 4, $n = 2^{31}$ and there are 32 levels (i.e., 32 *persistent* Count-Min sketches), while in PDS2, $n = 41,269$ and there are 16 levels.

Furthermore, in order to use heavy-hitter queries to answer top-k queries, we have to repeatedly run multiple heavy-hitter queries in a trial-and-error fashion to figure out the right Φ threshold. For $k = 10$, we find $\Phi = 0.005$ for Kosarak data. In addition, like in [41], we fix the sizes of the two dimensions of the Count-Min sketches in PDS/PDS2 to be $width = 20,000$ and $d = 7$. When the

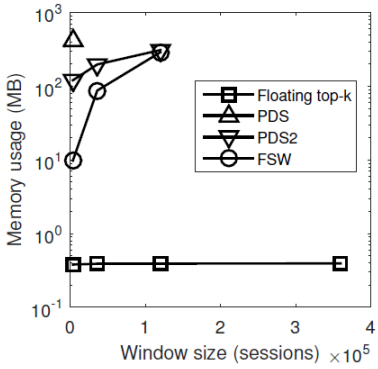


Fig. 6 Memory usages of Floating Top-k, PDS, and FSW for different window sizes on Kosarak

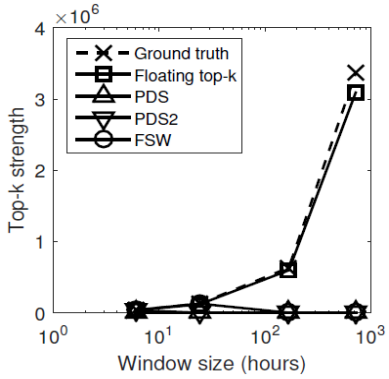


Fig. 7 Top-k strengths of Floating Top-k, PDS, FSW, and ground-truth top-k items for different window sizes on Twitter

number of distinct items is much greater than 20,000, the Count-Min sketches would have many collisions and would be very inaccurate. This leads to its low top-k strength in Figure 4 when window size is large.

Figure 4 shows that the Floating Top-k algorithm retrieves top-10 items with strengths very close to those of the ground-truth top-10 items. PDS takes the most memory (Figure 6), and is only feasible when window size $W = 3,600$ sessions among the ones we test (we constrain the maximum memory consumption of this program to be 512 MB). For window size $W = 36,000$ sessions PDS' memory footprint is already beyond the constraint. PDS2 (with a much smaller id range) is feasible for window size up to 120,000, but not 360,000 sessions. PDS has a very large memory footprint because it uses piecewise linear approximation (PLA [34]) to keep track of the history (within the window) of each cell of each Count-Min sketch. PDS has much more levels (32) than PDS2 (16), and hence is even worse. However, in a dynamic setting when we do not know the exact number of distinct items, we can only use PDS. We note that the PDS work [41] is not specifically targeted at the Windowed Top-k Frequent Items queries; it also addresses window point queries, heavy-hitter queries, and window join size estimation.

In Figure 6, we can see that the memory footprint of our method Floating

Top-k is very small (about 0.39 MB) and is almost constant (in fact, only increases slightly as window size increases). This is consistent with our analysis in Theorem 1 that the expected number of Floating Top-k tuples maintained in the pool is only $O(k \log W)$. By contrast, PDS and PDS2 have about three orders of magnitude larger memory footprints. We measure the number of Floating Top-k tuples for various W values, and show the result in Table 1 (for both Kosarak and Twitter datasets; the result for World-Cup data is similar and not shown). We can see that the numbers are relatively small compared to W and increase only logarithmically.

Table 1: Floating Top-k tuple pool sizes $|\Gamma|$ for various window sizes.

	Kosarak				Twitter			
Window	3.6k	36k	120k	360k	6h	1d	7d	30d
$\|\Gamma\|$	63	85	109	115	67	75	89	141

Accordingly, Figure 4 shows that PDS and PDS2 have reasonable top-k strength results for the first one and three window sizes, respectively, but even for those window sizes, they are not as good as the top-k strengths of Floating Top-k. Figure 5 shows the stream system throughput as it continuously maintains the corresponding data structures for each method respectively; thus it indicates the processing overhead in time for each method. Floating Top-k is 2 to 4 times faster than PDS2 and 40 times faster than PDS for the window sizes where they are feasible.

We also compare with FSW [21]. As discussed in Section 1.1, there is a limitation of FSW: it is only designed for one fixed window size, even though the window slides over time, while with Floating Top-k and PDS users can query any window sizes within W. Of course, with FSW, one can independently handle *multiple* fixed window sizes, but then the memory footprint and processing cost have to add up linearly as well. Nonetheless, for this comparison experiment, we

run FSW for each window size separately, and show the results of top-k strength, throughput, and memory footprint (for a single window size) in Figures 4, 5, and 6 respectively.

However, if we set the *time unit size* to be *one session*, as the case for Floating Top-k and PDS, FSW is *infeasible* (out of memory) even for window size 3,600 sessions (i.e., 3,600 time units). This is because FSW has a much greater memory footprint than Floating Top-k — FSW takes memory $O(kW)$, as it contains a histogram bucket for each of the W time units in each of the histograms that it maintains. Thus, for FSW only, we set the time unit size to be 60 sessions, and hence we have W that is 60 times smaller (in time units), even though the window contains the same number of sessions. Of course, by doing this, what we lose is that the FSW window slides with coarser granularity over time (i.e., updated more slowly). Even with this change, we can see from Figure 6 that FSW takes from 30 times larger memory footprint than Floating Top-k when window size is 3,600 sessions, to almost 1,000 times larger when the window size is 120,000 sessions. FSW is still out of memory when the window has 360,000 sessions. Its memory footprint increases linearly with W.

Figure 4 shows that, for the three window sizes where FSW is feasible, it is very accurate—its top-k strengths are about the same as those of the ground-truth top-k items. Figure 5 shows that Floating Top-k is much faster than FSW even for the window sizes where FSW is feasible, ranging from 2.4 times (when window size is 3,600 sessions) to 14 times (when window size is 120,000 sessions). FSW has a much greater processing overhead.

We next experiment on the Twitter dataset. Unlike the Kosarak dataset, we do not have integer item id's, and the stream items are strings (hashtags). This works the same for Floating Top-k and FSW; but for PDS, we have to dynamically create a dictionary that records the 1-to-1 mappings between string hashtags and integer id's, since PDS [41] uses the *dyadic range sum technique* [14]. Thus, we use MD5 hash function over a string hashtag to convert it into a random integer, and keep an in-memory dictionary for the mappings between integer id's and hashtags (so that we can also look up the dictionary to obtain

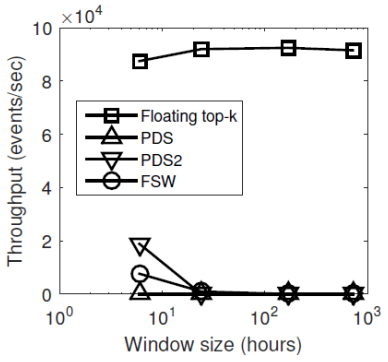


Fig. 8 Stream system throughputs of Floating Top-k, PDS, and FSW for different window sizes on Twitter

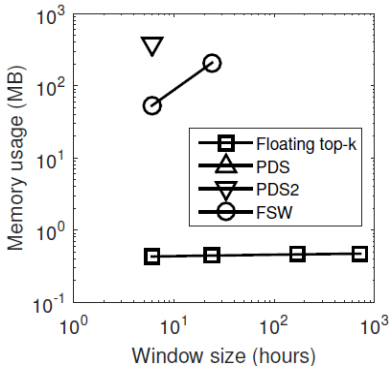


Fig. 9 Memory usages of Floating Top-k, PDS, and FSW for different window sizes on Twitter

the hashtags once we get top-k using PDS). In addition, Twitter hashtags are case insensitive.

We see some interesting results with Twitter data. For example, when we query the top-10 frequent hashtags in the one week period between November 14, 2015 and November 21, 2015, the results include "PrayForParis", "ParisAttacks", and "Paris", due to the terrorist attacks in Paris on November 13, 2015. The top ones also include "MTVStars" and "MadeintheAM" (a music album released on Nov 13, 2015). "GOPDebate" is also among the top ones during the presidential debate period in US.

In Figure 7, we show the top-k strengths of various methods, Floating Top-k, PDS, and FSM, together with those of the ground-truth top-10 hashtags for various window sizes ranging from 6 hours to 30 days, where a time unit is one minute. In Figures 8 and 9, we show the system throughputs and memory footprints. PDS's memory footprint (using integer id's) exceeds the constraint for all window sizes tested; hence it is not shown in Figure 9. Thus, we add a modified version of PDS, shown as PDS2 in the figures, where the integer id's are limited within 17 bits, i.e., after using MD5 hash on a hashtag string, we do "mod 2^{17}". With this version, Figure 9 shows that, for window size 6 hours, PDS2's memory footprint is about 3 orders of magnitude more than that of

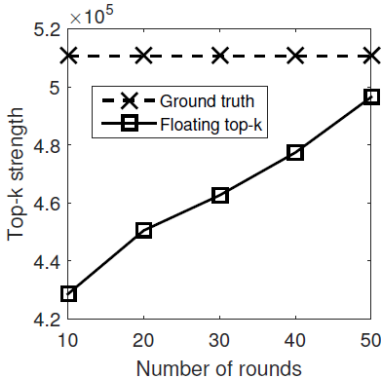


Fig. 10 Top-k strengths of Floating Top-k, and ground-truth top-k items for different numbers of rounds on WorldCup

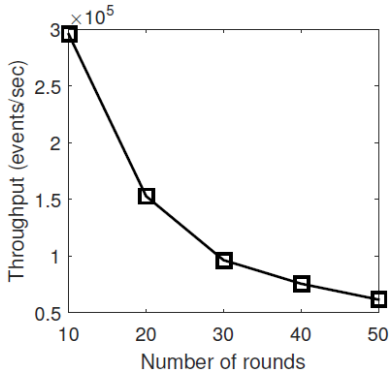


Fig. 11 Stream system throughputs of Floating Top-k for different numbers of rounds on WorldCup

the Floating Top-k, and becomes infeasible with greater window sizes. Figure 9 also shows that FSW's memory footprints are 123 and 470 times of the Floating Top-k's when window sizes are 6 and 24 hours, respectively. FSW's memory consumption exceeds the limit when window size is 7 days, as it grows linearly with W. We also show the Floating Top-k tuple pool sizes in Table 1.

Figure 7 shows that the top-k strengths from Floating Top-k are very close to those of the ground-truth top-k items, while PDS2 and FSW are close for window sizes where they are feasible. Figure 8 further shows that, Floating Top-k is about one to two orders of magnitude faster than PDS2 and FSW when they are feasible.

We also do the same experiments with the World-Cup dataset and obtain similar results. Here we show the effect of varying the number of rounds parameter c. We have discussed how to enhance the accuracy of the top-k results by repeating MaintainFloatingTopK and RetrieveTopK c times in Section 3.3. By default, we use $c = 20$, i.e., combining results from 20 runs of the Floating Top-k. We fix the time unit size at one hour, window size at 7 days, and vary the number of rounds between 10 and 50. Figure 10 shows the top-k strength results of Floating Top-k compared to the ground-truth top-k items when $k = 5$, while Figures 11 and 12 show the throughputs and memory

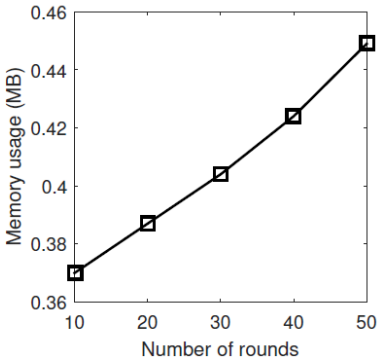


Fig. 12 Memory usages of Floating Top-k for different numbers of rounds on WorldCup

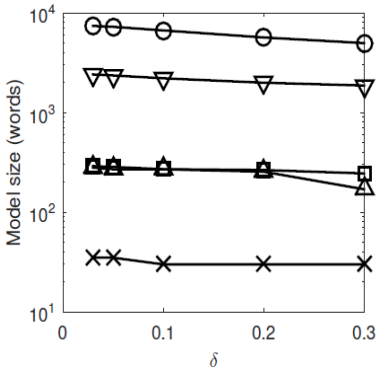


Fig. 13 Model sizes of PTM, and PLA for different δ values on Twitter

consumptions, respectively. These figures indicate a tradeoff between accuracy and performance. When the number of rounds increases, the top-k strength is closer to the ground-truth top-k items, but the throughput decreases and memory footprint increases (yet still very low).

In the next set of experiments, we exam our algorithm ProgressiveTrendModel (PTM) for tracking specific items. We compare with the most relevant and common previous approach of piecewise regression called *piecewise linear approximation* (PLA) [34]. We study the conciseness of the resulting models from PTM and PLA when they have the same accuracy guarantees, as well as their processing overheads reflected in system throughputs.

We first show the results under the Twitter dataset. We trace the frequency of hashtag "ParisAttacks" from November 14, 2015 to November 21, 2015, where we set time unit to 5 minutes. Recall that PTM has two accuracy parameters (ϵ, δ) (i.e., with probability at least $1-\epsilon$, the error is no more than δ fraction), while PLA only has one accuracy parameter δ (i.e., error is no more than δ fraction). Thus, to fairly compare the two, we set $\epsilon = 0$ for PTM, then PTM and PLA have the same accuracy guarantee for the same δ value. For various δ values, we show the conciseness (sizes) of the resulting models in Figure 13, and the system throughputs in Figure 14. Figures 13 and 14 share the same legend,

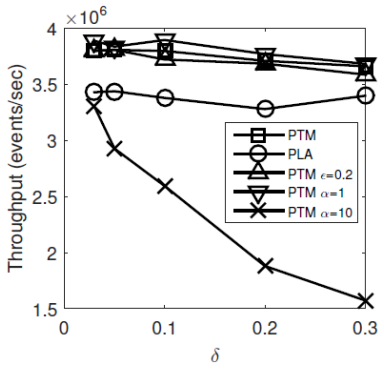


Fig. 14 Stream system throughputs of PTM, and PLA for different δ values on WorldCup

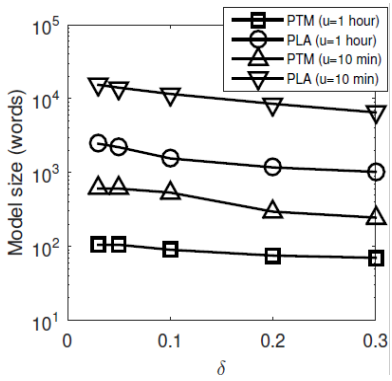


Fig. 15 Model sizes of PTM, and PLA for different δ values on WorldCup

as shown in Figure 14. The y-axis of Figure 13 is in "words", where a word is of 32 bits.

For PTM, we use the default parameter value $\alpha = 3$ (i.e., 3 tries before advancing to either next round or next epoch). We also experiment with a few variants of PTM, where we change only one parameter but keep others unchanged. The first variant is to set $\epsilon = 0.2$ (instead of 0), while the other two variants are to set $\alpha = 1$ and $\alpha = 10$, respectively. Figure 13 shows that the sizes of PTM are between 20 to 30 times smaller than those of the PLA under the same accuracy guarantees, while the throughput of PTM is slightly higher than PLA's (both are very high). PLA is slightly slower since it starts a new function more frequently than PTM. PTM's models are smaller because the exploration of progressive trend with different h values and the use of Taylor series result in more powerful functions than PLA, and hence each function covers more data points (time units).

The variant of PTM with $\epsilon = 0.2$ slightly reduces the model sizes, since being more permissive on errors can only reduce the number of functions. The variant of PTM with $\alpha = 1$ has a significantly greater model size, since there is only one try of h value before deciding to advance to next round or next epoch, resulting in many more functions. On the other hand, setting $\alpha = 10$ produces

significantly smaller model sizes, as there are more tries before giving up on an epoch and starting a new epoch and function. However, Figure 14 shows that, especially for higher δ values, PTM with $\alpha = 10$ is much slower. This is because allowing more errors implies verifying more data points for each function (before giving up), and higher α values signify that this verification needs to be done more times. Thus, the α parameter provides a tradeoff between model conciseness and performance.

Finally, we show the results of PTM and PLA over the World-Cup dataset, where we trace and model the frequency of the item with id 57 (object /images/space.gif) over a one-month period of time. The throughput result is similar to that of the Twitter dataset, and we only show the model conciseness result in Figure 15. We compare PTM and PLA over two different settings of time unit size $\mu = 1$ *hour* and $\mu = 10$ *minutes*, respectively. For each μ value, PTM is about 20 times smaller than PLA's. In addition, a greater time unit size (μ) implies fewer time units in the fixed time period, and hence a smaller model.

5.3. Summary of Experiment Results

Our comprehensive experimental results in this section over three real-world datasets show that Floating Top-k is the only feasible approach so far for the Windowed Top-k Frequent Items problem in top-k strength, memory footprint, and throughput. The memory footprints of PDS and FSW are 2 to 3 orders of magnitude larger than that of Floating Top-k for small W, and grow linearly with W. Floating Top-k is also 1 to 2 orders of magnitude faster than PDS and FSW when they are feasible. The $O(k \log W)$ space complexity of Floating Top-k makes it highly scalable for high-rate data streams with dynamic items and arbitrary-size windows. Furthermore, for the Item Tracking Problem, the PTM is much more concise (20 to 30 times) and slightly faster than PLA under the same accuracy guarantees.

6. Conclusions

In high velocity, volume, and diversity modern data stream applications, the ability to query the top-k most frequent ("hottest") items is particularly important. We propose a novel solution called Floating Top-k. Our comprehensive analysis and experiments show that Floating Top-k is the only feasible and scalable solution to this problem thus far. In addition, we devise a solution to the related problem of concisely tracking selected items in data streams, which also significantly improves upon previous work.

Acknowledgements

Chunyao Song was supported in part by the NSFC under the grants 61702285 and 61772289, the NSF of Tianjin under the grants 17JCQNJC00200, and the Fundamental Research Funds for the Central Universities under the grants 63171111. Tingjian Ge was supported in part by the National Science Foundation under the grants IIS-1149417 (CAREER award) and IIS-1633271.

References

[1] https://dev.twitter.com/streaming/overview.

[2] http://fimi.ua.ac.be/data/ kosarak.dat.gz.

[3] http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[4] https://twitter.com/?lang=en.

[5] http://twitter4j.org/en/index.html.

[6] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 286–296, 2004.

[7] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 633–634, 2002.

[8] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *Proceedings of the 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[9] Richard L. Burden and J. Douglas Faires. *Numerical analysis*. Brooks/Cole Pacific Grove, CA, 2001.

[10] Massimo Cafaro, Marco Pulimeno, Italo Epicoco, and Giovanni Aloisio. Mining frequent items in the time fading model. *Information Sciences*, 370-371:221–238, 2016.

[11] Massimo Cafaro, Marco Pulimeno, and Italo Epicoco. Parallel mining of time-faded heavy hitters. *Expert Systems with Applications*, 96:115–128, 2018.

[12] Ling Chen and Qingling Mei. Mining frequent items in data stream using time fading model. *Information Sciences*, 257(2):54–69, 2014.

[13] Graham Cormode and Marios Hadjieleftheriou. Methods for finding frequent items in data streams. *VLDB Journal*, 19(1):3–20, 2010.

[14] Graham Cormode and Shan Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, volume 30, pages 296–306, 2003.

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

[16] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002.

[17] Ugo Erra, Sabrina Senatore, Fernando Minnella, and Giuseppe Caggianese. Approximate TF-IDF based on topic extraction from massive message stream using the GPU. *Information Sciences*, 292(C):143–161, 2015.

[18] Dirk Helbing et al. Saving human lives: what complexity science and information systems can contribute. *Journal of Statistical Physics*, 158(3): 735–781, 2015.

[19] Phillip B. Gibbons and Srikanta Tirthapura. Distributed streams algorithms for sliding windows. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 63–72, 2002.

[20] Lukasz Golab, David DeHaan, Erik D. Demaine, Alejandro López-Ortiz, and James Ian Munro. Identifying frequent items in sliding windows over on-line packet streams. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, pages 173–178, 2003.

[21] Nuno Homem and Joao Paulo Carvalho. Finding top-k elements in a time-sliding window. *Evolving Systems*, 2(1):51–70, 2011.

[22] Mahdi Jalili and Matjaž Perc. Information cascades in complex networks. *Journal of Complex Networks*, 5:665–693, 2017.

[23] Tobias Kuhn, Matjaž Perc, and Dirk Helbing. Inheritance patterns in citation networks reveal scientific memes. *Physical Review X*, 4(4):041036, 2014.

[24] Hoang Thanh Lam and Toon Calder. Mining top-k frequent items in a data stream with flexible sliding windows. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 283–292, 2010.

[25] Tuong Le, Bay Vo, and Sung Wook Baik. Efficient algorithms for mining top-rank-k erasable patterns using pruning strategies and the subsume concept. *Engineering Applications of Artificial Intelligence*, 68:1–9, 2018.

[26] Lapkei Lee and Hingfung Ting. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 290–297, 2006.

[27] Hongyan Liu, Yuan Lin, and Jiawei Han. Methods for mining frequent items in data streams: an overview. *Knowledge and Information Systems*, 26(1):1–30, 2011.

[28] Spyros G. Makridakis, Steven C. Wheelwright, and Victor E. McGee. *Forecasting: methods and applications*, volume 753. Wiley, 1983.

[29] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory*, pages 398–412, 2005.

[30] Katsiaryna Mirylenka, Graham Cormode, Themis Palpanas, and Divesh Srivastava. Conditional heavy hitters: detecting interesting correlations in data streams. *VLDB Journal*, 24(3):395–414, 2015.

[31] Jayadev Misra and David Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, 1982.

[32] Michael Mitzenmacher and Eli Upfal. *Probability and computing: randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[33] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 635–646, 2006.

[34] Joseph O'Rourke. An on-line algorithm for fitting straight lines between data ranges. *Communications of the ACM*, 24(9):574–578, 1981.

[35] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. Sketch-based querying of distributed sliding-window data streams. *VLDB Journal*, 5(10):992–1003, 2012.

[36] Matjaž Perc. Self-organization of progress across the century of physics. *Scientific Reports*, 3(1720), 2013.

[37] Zubair Shah, Abdun Naser Mahmood, and Michael Barlow. Computing discounted multidimensional hierarchical aggregates using modified Misra Gries algorithm. *Performance Evaluation*, 91:170–186, 2015.

[38] Zubair Shah, Abdun Naser Mahmood, Zahir Tari, and Albert Y. Zomaya. A technique for efficient query estimation over distributed data streams. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2770–2783, 2017.

[39] Zubair Shah, Abdun Naser Mahmood, Michael Barlow, Zahir Tari, Xun Yi, and Albert Y. Zomaya. Computing hierarchical summary from two-dimensional big data streams. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):803–818, 2018.

[40] Chunyao Song, Xuanming Liu, and Tingjian Ge. Top-k frequent items and item frequency tracking over sliding windows of any sizes. In *Proceedings of the 33rd IEEE International Conference on Data Engineering*, pages 199–202, 2017.

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