6-2017

SAP: Improving continuous top-K queries over streaming data

Rui ZHU
Northeastern University

Bin WANG
Northeastern University

Xiaochun YANG
Northeastern University

Baihua ZHENG
Singapore Management University, bhzheng@smu.edu.sg

Guoren WANG
Northeastern University

SAP: Improving Continuous Top-K Queries over Streaming Data

Rui Zhu, Bin Wang, Xiaochun Yang, Baihua Zheng, Guoren Wang

Abstract—Continuous top-k query over streaming data is a fundamental problem in database. In this paper, we focus on the sliding window scenario, where a continuous top-k query returns the top-k objects within each query window on the data stream. Existing algorithms support this type of queries via incrementally maintaining a subset of objects in the window and try to retrieve the answer from this subset as much as possible whenever the window slides. However, since all the existing algorithms are sensitive to query parameters and data distribution, they all suffer from expensive incremental maintenance cost. In this paper, we propose a self-adaptive partition framework to support continuous top-k query. It partitions the window into sub-windows and only maintains a small number of candidates with highest scores in each sub-window. Based on this framework, we have developed several partition algorithms to cater for different object distributions and query parameters. To our best knowledge, it is the first algorithm that achieves logarithmic complexity w.r.t. k for incrementally maintaining the candidate set even in the *worst* case scenarios.

Index Terms—Continuous top-k query, sliding window, streaming data, dynamic partition

1 INTRODUCTION

This paper studies the problem of continuous top-k query over sliding window [6], [17], [25], a classic problem in the data stream environment. It has various applications, ranging from sensor data analysis to economic decision making, as well as wireless sensor and market surveillance.

For example, in stock market, a continuous top-k query (top-k query for short) can be used to monitor real-time transactions and hence retrieve the 10 most significant transactions within the last 30 minutes. The query results could help investors to track market hotspots and make sensible decisions. In fire monitoring systems, a top-k query can be used to monitor real-time data (e.g., temperatures, humidity, and UV indexes) from sensors and hence detect the ten regions in which conflagrations are most likely to happen. In traffic systems, it can be used to monitor real-time data (e.g., vehicle speed, vehicle density) from RFID readers and thus detect the top-10 congested regions. These are only a small part of the immense applications of top-k query over streaming data.

Without loss of generality, this query window can be either time- or count-based. In both cases, the query window has a fixed window size and a fixed slide (either a time interval or an object count). Formally, in a *count-based window*, it returns the k objects with highest scores in the query window containing n objects whenever the window slides; in a *time-based window*, it returns the k objects with highest scores in the last n time units whenever the window slides [11], [22]. In other words, a continuous top-k query could be expressed by the tuple $\langle n, k, s, F\rangle$, where parameter s represents the number of objects that arrive whenever the window slides or the duration in terms of time units between two adjacent window sliding, and F is a preference function. The continuous top-k query returns k objects $o_1, o_2, \ldots, o_k$ from the query window that have the highest $F(o_i)$ scores. For simplicity, in the rest of the paper, we only consider the count-based windows. However, our techniques can also be applied to answer top-k queries over the time-based sliding windows (see Appendix A).

Based on whether re-scanning of the window is required, the state-of-the-art efforts on continuous top-k query over streaming data can be categorized into two groups, namely *multi-pass* based approaches and *one-pass* based approaches. Many *multi-pass* based approaches maintain top-k' [17], [26] objects in current window W as candidates with $k \leq k' \leq k_{max}$, where parameter k_{max} (e.g., $k_{max} = 2k$) controls the maximum size of candidate set C. The main limitation of these approaches is that if many candidates expire from the window but only a few of newly arrived objects are able to contribute to the candidate set, they have to perform expensive re-scanning to construct a new candidate set.

One representative *one-pass* based approach utilizes k-skyband technique to avoid the re-scanning problem [19]. Here, an object o is a k-skyband object if there are less than k objects o' who come later than o and have scores higher than $F(o)$. As proved in [19], top-k results can only appear in those skyband objects and hence re-scanning can be successfully avoided. However, as will be reviewed in Section 2, one-pass based approaches suffer from extremely high computation cost incurred by maintaining domination relationships among candidates. In the worst case, the incremental maintenance complexity is linear to the window size.

MinTopK algorithm [25] is another representative one-pass based approach. It improves k-skyband based algorithm via considering the parameter s. Under sliding window model, since s objects actually arrive in (or expire from) the window at the same time, only the top-k objects among these s objects have the chance to become k-skyband. By maintaining all the top-k objects from each s objects in a candidate set, MinTopK only performs one scanning and meanwhile guarantees the result accuracy. It also proposes some techniques to further identify and filter out *removable* candidates that have *zero* chance to become a result object. Obviously, MinTopK performs well if $s \gg k$. However, when the condition $s \gg k$ does not hold, MinTopK might not perform well. The reason is, the smaller the s, the more the candidates MinTopK

- *R. Zhu, B. Wang, X.Yang and G. Wang are with the College of Information Science and Engineering, Northeastern University, China*
- *B. Zheng is with the School of Information Systems, Singapore Management University, Singapore.*

needs to maintain, and the higher the incremental maintenance cost is. As shown in Section 2, the incremental maintenance cost of MinTopK is $O(\frac{n}{s}+\log(\frac{nk}{\max(s,k)})+cost_F)$, which heavily relies on the parameter s.

In summary, the *multi-pass* based approaches may frequently re-scan the window, while the *one-pass* based approaches have to maintain many candidates. When the window slides, the *multi-pass* based approaches might need to re-scan the window if the size of its candidate set falls below k; while the *one-pass* based approaches might need to evaluate and maintain new candidates even with low scores as the chance that an object becomes a candidate not only depends on its score, but also depends on its arrival order. In addition, both types of approaches are sensitive to both the query parameters(i.e., n, s, and k) and the distribution of objects' scores.

Solution Overview. Motivated by the disadvantages of existing approaches on continuous top-k searches on streaming data, we dedicate this paper to the development of a new approach that is less sensitive to the query parameters, and is able to achieve a stable and efficient performance under various distributions of objects' scores. We propose a novel *self-adaptive partition based framework*, named SAP, as a solution. SAP employs a popular technique named *partition* [1], [7], [10], [8], [9] to organize objects in the query window. In brief, SAP partitions the window into several disjoint sub-windows. In each partition (a.k.a. sub-window) P_i, we maintain the k objects with highest scores as candidates, denoted as P_i^k, and objects in $P_i - P_i^k$ that may contribute to the result set, namely *meaningful objects*, via a meaningful object set M_i. Intuitively, for each partition P_i, objects in $P_i - P_i^k - M_i$ are guaranteed *non-result objects*, and those in M_i are guaranteed *non-result objects* until P_i moves to the front of the window. Consequently, the partition scheme effectively limits the incremental maintenance scope to only the objects in $\cup P_i^k$ and the meaningful objects in M_i of the front partition (i.e., the partition currently located at the front of the window)(see Section 3.1).

One simple approach to construct M_i is to re-scan the partition P_i when objects in P_i^k move out of the window. However, this requires re-scanning of P_i whose cost might be high, especially when the size of P_i is big. We maintain the k-skyband objects in $(P_i - P_i^k)$ as meaningful objects. We propose a novel structure namely S-AVL (see Section 5.1) that is able to maintain a set of potential meaningful objects for each partition P_i with very low cost. In addition, we also strategically defer the formation of M_i to the moment when P_i is about to become the first partition in the current window. If we select a proper partition size (e.g., $\sqrt{n\max(s,k)}$), S-AVL structure is able to reduce the incremental maintenance cost of SAP to $O(\log k)$. To our best knowledge, it is the *first* algorithm that achieves logarithmic complexity w.r.t. k for incrementally maintaining the candidate set even *in the worst case scenarios* (see Section 4.1).

Our SAP framework is general, and it can support different partitioning algorithms. In this paper, we first present a simple equal partition in order to quantify the main benefit of partitioning in a simplified manner; we then present a dynamic partition which can be used in real applications to select a proper partition size for each partition by considering the score distribution of underlying data stream and query parameters. The dynamic partition algorithm enables SAP framework to adjust the partition size to cater for queries with different parameters and data streams with different distributions. Note, this task is very challenging as SAP does not assume any prior knowledge on objects' distribution, and it is almost impossible to learn the distribution of streaming data in the high-speed streaming environment.

The rest of the paper is organized as follows. First, we review existing works in Section 2. Next, we introduce the self-adaptive partition based framework (SAP) in Section 3. We also propose a few partitioning algorithms to cater for different types of data streams in Section 4. We then propose a novel data structure named S-AVL to reduce the formation cost of meaningful object set M_i in Section 5. We report our comprehensive experimental study in Section 6 to evaluate the performance of SAP, as compared with state-of-the-art techniques. Finally, we conclude our paper in Section 7.

2 PRELIMINARY

In the next section, we first review some important existing results about continuous queries over sliding window. Then, Section 2.2 describes the *mann-whitney-ranks test*, which is an effective method for hypothesis testing, and is fundamental to the subsequent discussion.

2.1 Related Work

A large number of continuous queries over streaming data have been studied, including join [21], continuous top-k multi-query [27], continuous top-k query [17], [25], outlier detection [12], [3], [4], kNN query [18], [2] and skyline query [23], [14]. In this paper, we focus on the problem of continuous top-k query over streaming data. Based on whether re-scanning is necessary, existing algorithms can be clustered into two groups, namely *multi-pass* based approaches and *one-pass* based approaches.

Multi-pass based approaches. Many multi-pass based approaches [17], [26] maintain top-k' ($k \leq k' \leq k_{max}$) objects in current window W as candidates, where k_{max} is a parameter that determines the maximum capacity of the candidate set C (e.g., $k_{max} = 2k$). Whenever a query result expires from the window, if $|C| \geq k$, the new result could be retrieved from C; otherwise, a re-scanning of current window W is triggered to re-fill the candidate set C with top-k_{max} objects.

Since the re-scanning cost is high, several algorithms have been proposed to reduce the re-scanning cost, including the most representative one SMA [17]. SMA reduces the re-scanning cost via two facets. First, it uses a grid structure to index streaming data. When re-scanning of the window happens, the grid structure enables it to access *only* a few cells according to the coefficients of the preference function F. Second, it introduces the concept of *dominance* [13], [20] to remove *unqualified* candidates that cannot become query results in any time slot. Formally, given objects o and o', o is dominated by o' if $F(o) < F(o')$ and $o.t \leq o'.t$, denoted as $o' \prec o$. Here, $o.t$ refers to the arrival order of object o.

Given a sliding window W and a set of objects O_W in W, o's dominant number, denoted as $D(o, O_W, W)$, refers to the number of objects in W that dominate o, i.e., $D(o, O_W, W) = |\{o' \in O_W | o' \prec o\}|$. As shown in Figure 1(b), object a is dominated by three objects and hence $D(a, O_W, W)$ is set to three. Objects with $D(o, O_W, W) < k$ are named as *k-skybands*, and non-k-skybands (i.e., objects with $D(o, O_W, W) \geq k$) can be safely pruned away as they will not be one of the top-k objects. In this way, those filtered-out objects will not be evaluated when the re-scanning happens and hence the re-scanning cost is reduced.

However, SMA only maintains a small portion of k-skybands, so the re-scanning still happens. For example, when objects' scores keep decreasing (as shown in Figure 1(a)), the re-scanning happens frequently and the total re-scanning cost could be high.

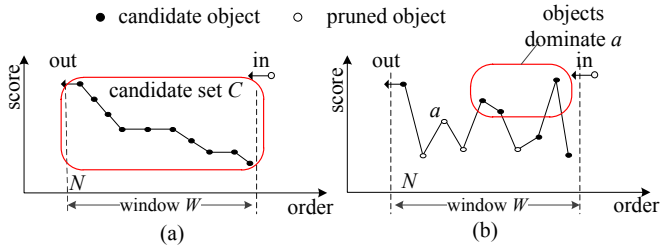


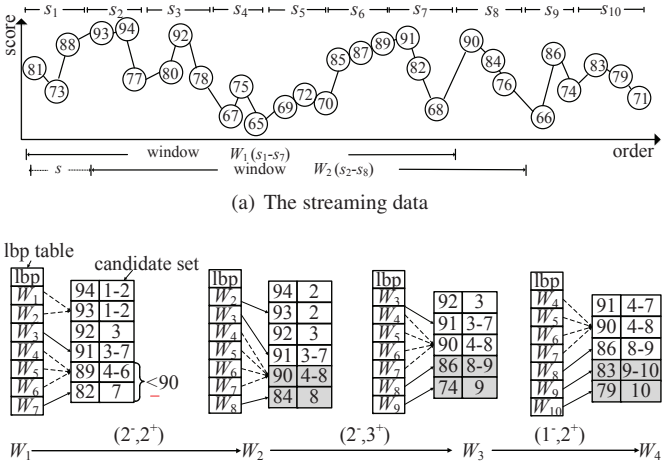
Fig. 1. Problems with k-skyband based approaches ($N = 10$, $k = 2$)

One-pass based approaches. In order to avoid the re-scanning, a simple approach is to maintain all the k-skybands in the window [19]. Whenever a new object o_{in} slides into the window W, it inserts o_{in} into the candidate set C. Meanwhile, $D(o, O_W, W)$ values of all the candidate objects $o \in C$ dominated by o_{in} are increased by one and only objects o with $D(o, O_W, W) < k$ remain in C [19]. Obviously, o_{in} remains within the candidate set C until another k objects with scores higher than $F(o_{in})$ flow into the window. In other words, all the objects, even with very low scores, are present within candidate set C for certain period of time. Although all those false candidates will be removed from C eventually, it takes extra space to preserve them in C and extra time to maintain their dominant numbers. As pointed out in [19], when the objects' scores are not correlated with their arrival orders (e.g., as shown in Figure 1(b)), k-skyband based approaches incur $O(k \ln \frac{n}{k})$ space cost and $O(\log k + \log \ln n + n_d)$ incremental maintenance cost. Here, n_d refers to the average number of candidates in current candidate set C that can be dominated by the newly arrived object o_{in}.

In an extreme case where scores of objects in the window are anti-correlated with their arrival orders (as shown in Figure 1(a)), all the objects are k-skyband objects and they all have to be preserved in C. In this case, the space cost is $O(n)$. If a new object o_{in} with $F(o_{in})$ larger than that of any candidate object flows into the window, it actually dominates all the candidate objects and hence n_d is n, and the total incremental maintenance cost is increased to $O(n + cost_F)$.

A novel one-pass algorithm named MinTopK [25] improves the performance of k-skyband based algorithms via considering parameter s. Recall that in a count-based window, parameter s refers to the number of newly arrival objects whenever the window slides, and MinTopK maintains every s objects that flow into the window at the same order i in a set s_i. To simplify our discussion, we assume s is a factor of n and $m = \frac{n}{s}$ is an integer. Accordingly, a query window W could be represented by a set of s_is, e.g., $\{s_1, \cdots, s_m\}$.

MinTopK consists of *initialization* phase and *incremental maintenance* phase. In the initialization phase, it constructs a list of *predicting result sets* for each W_i with $1 \leq i \leq m$, assuming the current window contains $\{s_1, s_2, \cdots, s_m\}$. Here, W_i is a *future* window formed by $\{s_i, s_{i+1}, \cdots, s_{m+i-1}\}$ where some of its s_js might not be available yet. For example, when W_1 is the current window, s_8 of W_2 is not available, and s_8 and s_9 of W_3 are not available yet. A *predicting result set* R_i^m contains k objects in the current window that might contribute to the result set corresponding to a future window W_i, i.e., R_i^m maintains the top-k objects in sets $\{s_i, s_{i+1}, \cdots, s_m\}$. In other words, $\cup_{j=1}^{m} R_j^m$ contains all the objects in the current window W_1 that might become query results in the future and hence the rest of objects in W_1 can be safely discarded, as proved by [25]. An example is plotted in Figure 2 with $s = 3$, $n = 21$, $k = 2$, and $m = 7$. R_1^7 contains the top-2 objects in $\cup_{j=1}^{7} s_j$ that are 94 and 93. Similarly, $R_2^7 = \{94, 93\}$, $R_3^7 = \{92, 91\}$, $R_4^7 = \{91, 89\}$, $R_5^7 = \{91, 89\}$, $R_6^7 = \{91, 89\}$, and $R_7^7 = \{91, 82\}$. The candidate set is their union, i.e., $\cup_{i=1}^{7} R_i^7$ (e.g., $\{94, 93, 92, 91, 89, 82\}$ in above example).



(a) The streaming data

(b) v^- and u^+denote the numbers of deletions and insertions respectively when the window slides from W_i to W_{i+1}.

Fig. 2. The MinTopK Algorithm ($n = 21$, $k = 2$, $s = 3$)

In addition to the candidate set, MinTopK also maintains a pointer lbp for each future window W_i. Here, lbp of W_i points to the smallest candidate o in R_i^m. In the near future, when a new object o_{in} arrives, if $F(o_{in})$ is higher than the score of the object pointed by $W_i.lbp$, o_{in} will become a new candidate for W_i. In brief, lbps serve two purposes, i.e., locating all the future windows W_i such that the new object o_{in} is predicted to be one of the top-k objects; and identifying false candidates that are disqualified from being one of the top-k objects in any future window W_i because of o_{in}. In the event that o_{in} cannot be one of the top-k objects for any of the future windows, o_{in} can be safely discarded. The second phase of MinTopK is to process newly arrived objects and update lbps if necessary. For example, when the window slides from W_1 to W_2, it processes objects in s_8 one by one. MinTopK first locates 90 to the right position in the current candidate set, that is right after 91 but before candidates 89 and 82. 89 is pointed by the lbp of W_4, W_5 and W_6, while 82 is pointed by the lbp of W_7. Given the fact that $90 > 89$ and $90 > 82$, 90 is inserted to the candidate set as a new candidate, as it is currently one of the top-k objects for future window W_js with $4 \leq j \leq 8$. Accordingly, lbps of W_is with $4 \leq i \leq 7$ will be pointing to 90, and candidates 89 and 82 will be removed from the candidate set. MinTopK then checks the second object 84. It inserts 84 into the candidate set and updates the lbp of W_8 to point to 84. It also checks the third object 78 and discards it immediately as it cannot contribute to any predicted result set. Figure 2(b) shows the refined candidate set w.r.t. W_1, W_2, W_3, and W_4.

Now, we discuss the incremental cost of minTopK. Let $\langle n, k, s, F \rangle$ be a top-k query. Given a newly arrival object o_{in}, minTopK firstly inserts o_{in} into the candidate set C if o_{in} cannot be filtered out, with the cost of $O(\log(|C|))$. Note $|C|$, the size of the candidate set, is bounded by $\frac{nk}{\max(s,k)}$. Next, minTopK accesses the lbp table to update the lower-bound of each predicting result

set with score lower than $F(o_{in})$, with the cost bounded by the total number of *lbp*s (i.e., $\frac{n}{s}$ at most). In total, the incremental maintenance cost of minTopK is $O(\frac{n}{s} + \log(\frac{nk}{\max(s,k)}))$ *in the worst case*.

Other queries. Besides the continuous top-k query, other types of continuous queries (e.g., continuous skyline query and kNN query) have also been well studied. The first approach for skyline queries over sliding window is proposed by Lin et al. in [14]. They use an interval tree to maintain points in a sliding window. Tao et al. propose methods for skyline monitoring over sliding windows in [23], aiming at improving the space/time efficiency by discarding records that cannot participate in the skyline until their expiration.

Bohm et al. study the problem of continuous NN query over sliding window in [2]. They use a skyline data structure to maintain objects that mingt be able to become the nearest neighbor of one or more continuous queries. In order to enhance the query processing, they propose a delaying strategy to process those objects which are not immediately nearest neighbors of any query. Mouratidis et al. study the continuous NN queries over sliding window in [18] . Their key idea is to reduce the problem to skyline maintenance in the distance-time space and to pre-compute the future changes in the NN set. All these solutions can be easily adopted to answer kNN query.

The common idea of these efforts is to maintain a subset of objects to answer the query via maintaining the domination relationship among candidates. However, like top-k algorithms, they do not fully utilize the query parameters and distribution of streaming data to enhance the algorithm performance.

Discussion. In summary, existing approaches are sensitive to query parameters, and none of them has considered the challenges brought by different data distributions. For example, when objects' scores keep decreasing, *one-pass* based approaches suffer from expensive candidate maintenance cost; while *multi-pass* based approaches suffer from frequent re-scanning. Motivated by these observations, we devote this paper to the design and development of new algorithms that can support top-k monitoring over streaming data in real-time, and is able to achieve an efficient and stable performance, independent of the distribution and properties of data stream and the settings of the queries.

2.2 The Mann-Whitney-Ranks Test

Given two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$ with unknown distributions, WRT (short for *mann-whitney-ranks test*) can be applied for testing whether $\mathcal{D}_1$ tends to have larger values than $\mathcal{D}_2$[1]. Specially, let SD_1 and SD_2 be the sample sets of $\mathcal{D}_1$ and $\mathcal{D}_2$ respectively. We re-arrange these objects of $SD_1 \cup SD_2$ in a single sequence based on ascending order, after applying a score function. For example, in our scenario (i.e., top-k search), we can order objects o in $SD_1 \cup SD_2$ based on their $F(o)$ values, from the smallest to the largest. Each object $o \in SD_1 \cup SD_2$ will be assigned a rank, denoted as $o.r \in [1, |SD_1| + |SD_2|]$. Function $R_1(SD_1, SD_2)$/$R_2(SD_1, SD_2)$ (in short R_1/R_2) is to return the summation of ranks for all the objects of SD_1/SD_2 in the sample set $SD_1 \cup SD_2$ under this re-arrangement, i.e., $R_1(SD_1, SD_2) = \sum_{o \in SD_1} o.r$ and $R_2(SD_1, SD_2) = \sum_{o \in SD_2} o.r$.

Given the hypothesis $\mathcal{H}$ that if $\mathcal{D}_1$ and $\mathcal{D}_2$ are actually drawn from the same distribution, the samples in SD_1 shall tend to be dispersed throughout the ordering of all $|SD_1 \cup SD_2|$ samples. In other words, the value of R_1 and that of R_2 are able to tell certain clues. To be more specific, if $T_{low} \leq \mathsf{min}(R_1, R_2) \leq T_{up}$, $\mathcal{D}_1$ and $\mathcal{D}_2$ might be from the same distribution, and the hypothesis $\mathcal{H}$ is accepted accordingly; otherwise, the hypothesis is rejected. Here, T_{low} and T_{up} refer to the lower-bound and upper-bound of the acceptance region respectively, which could be obtained from the table of rank-sum test [15]. If $R_1 > T_{up}$, we conclude that the objects in $\mathcal{D}_1$ tend to be larger than the objects in $\mathcal{D}_2$.

Note, when both SD_1 and SD_2 contain at least 10 samples, WRT has approximately the normal distribution. In this case, we could construct the similar function for evaluation. To be more specific, we can treat the distribution of R_1 as if it is Normal(μ_1, σ_1), where $\mu_1 = \frac{R_1(R_1+R_2+1)}{2}$, and $\sigma_1 = \sqrt{\frac{R_1 R_2 (R_1+R_2+1)}{12}}$. Here, $u_{1-\frac{\alpha}{2}}$ is the the upper quartile in normal distribution, and α is the probability of Type I error, i.e., with the default value 0.05 [16].

3 The Partition Based Framework

In this section, we propose a novel *self-adaptive partition based framework*, named SAP. We first present the basic idea of our SAP framework, and explain how to support continuous top-k queries under SAP framework. To simplify our discussion, we assume $s = 1$ in this section. However, our techniques are independent on s and definitely applicable when $s > 1$, and we will discuss the impact of s on the partitions in Sections 4 and 6.

3.1 Basic Idea

Formally, given a query window W that contains n objects, a partition $\mathcal{P}(W, m) = \{P_0, P_1, \cdots P_{m-1}\}$ is to partition the objects in W into m sub-windows P_is such that i) $W = \cup_{i=0}^{m-1} P_i$; ii) $\forall 0 \leq i < j < m$, $P_i \cap P_j = \emptyset$; and iii) $\forall 0 \leq i < j < m$, all the objects in P_i arrive earlier than any object in P_j. Via partitioning the objects in a window into smaller sub-windows, we effectively limit the update caused by window sliding to *at most* one sub-window. In the rest of the paper, we may use sub-windows and partitions interchangeably if the context is clear.

In each partition P_i, we maintain the k objects with highest scores as candidates, denoted as P_i^k. An example partition is depicted in Figure 3. Objects in a window W are partitioned into m (5 in our example) equal partitions, denoted as P_0, P_1, $\cdots$, P_{m-1}. Note partitions could have different sizes, although we assume equal partition for simplicity in this example. To simplify our discussion, each circle refers to an object o with the number in the middle representing $F(o)$ and we simply name an object as $F(o)$ (e.g., object 95 refers to the first object in P_0 with $F(o) = 95$). P_i^k keeps the top-k objects in P_i, e.g., $P_0^k = \{95, 92\}$ and $P_1^k = \{87, 79\}$ ($k = 2$ in our example).

All the objects in $P_i - P_i^k$ have their scores lower than the scores of the top-k objects in P_i^k and hence will not be able to contribute to the result set, until top-k objects in P_i^k start sliding out of the window. In order to guarantee the accuracy of the result, re-scanning of $P_i - P_i^k$ is needed. In order to avoid re-scanning partition P_i when objects in P_i^k move out of the window, we maintain a meaningful object set M_i that contains all the

1. Note, WRT works when the samples in $SD_1 \cup SD_2$ are independent of each other. If the samples in $SD_1 \cup SD_2$ are dependent of each other, we could use *paired sample T-Test* [15] for evaluation. Because of space limitation, we skip the details.

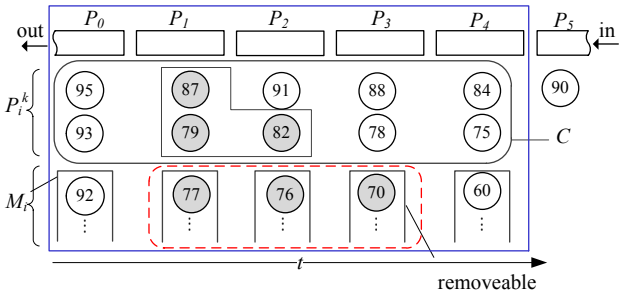


Fig. 3. An Example Equal Partition ($m = 5$, $k = 2$, $s = 1$)

meaningful objects in $(P_i - P_i^k)$ with the potential to become the result objects when objects in P_i^k slide out of the window. Take the partition P_0 in Figure 3 as an example. The corresponding $M_0 = \{92\}$. The detailed formation and maintenance of M_i will be explained later in Section 5.1.

We also observe that set M_i for some partition P_i could be empty as none of the objects in $(P_i - P_i^k)$ is qualified. For example, in Figure 3, the smallest candidate object (i.e., 79) in P_1 is dominated by 91, 82 from P_2, 88 from P_3, and 84 from P_4, and hence M_1 is guaranteed to be empty. Is it possible to derive that M_i is empty without checking objects in $(P_i - P_i^k)$? The answer is yes, via a novel concept named *group dominance number* as formally defined in Definition 1.

Definition 1. **Group Dominance Number.** Given a partition $\mathcal{P}(W, m)$, let P_i be a partition in W and o_i^k be the object with k-th highest score in P_i. The *group dominance number* of partition P_i, denoted as $P_i.\rho$, refers to the number of candidate objects in $C - P_i^k$ that dominate o_i^k without considering candidates contributed by P_i, i.e., $P_i.\rho = |\{o \in C - P_i | o \prec o_i^k\}|$.

To be more specific, given a partition P_i with $P_i.\rho \geq k$, it is guaranteed that the object o_k with smallest value in P_i^k is dominated by $P_i.\rho$ objects in the following partitions. As all the objects in $P_i - P_i^k$ have their values smaller than that of o_k, they are all dominated by at least those $P_i.\rho$ ($\geq k$) objects in the following partitions and hence could not be meaningful objects. In other words, M_i corresponding to the partition P_i with $P_i.\rho \geq k$ is guaranteed to be empty, and its formation could be safely skipped.

Lemma 1. Given a partition $\mathcal{P}(W, m)$, let R be the result set and M_0 be the meaningful object set corresponding to the partition P_0 that is currently located at the front of the window. If $P_0.\rho \geq k$, it is guaranteed that $R \subseteq C$; otherwise, it is guaranteed that $R \subseteq C \cup M_0$.

As stated in Lemma 1, if $P_0.\rho \geq k$, C (where $C = \cup P_i^k$) forms a superset for the result set R. Otherwise, $C \cup M_0$ (where $C = \cup P_i^k$) forms a superset for the result set R. As Lemma 1 is straightforward, we skip the proof for space saving.

Based on Lemma 1 and the group dominance number introduced above, we propose Top-k algorithm to support the top-k search. The pseudo code is listed in Algorithm 1. It only considers a partition $\mathcal{P}(W, m)$. For sliding-out object o_{out}, it only affects the first partition P_0 (line 1). If it is a candidate object, we perform additional update, guided by Lemma 1 (lines 2-6). For the sliding-in object o_{in}, we insert it into P_m^k (line 1), and perform result set update if necessary (line 7-11). Here, P_m^k uses a AVL-Tree to maintain the k objects with highest scores in P_m. Especially, when P_m reaches its full capacity (e.g., $\frac{n}{m}$ in the case of equal

Algorithm 1: Top-k

Input: a partition on query window $\mathcal{P}(W, m)$, objects o_{in} and o_{out}, current result set R and candidate set C
Output: updated partition $\mathcal{P}(W, m)$, updated candidate Set C, and updated result Set R

1 Window Update: $P_0 \leftarrow P_0 - \{o_{out}\}$, $P_m \leftarrow P_m \cup \{o_{in}\}$;
2 **if** $o_{out} \in C$ **then**
3 $C \leftarrow C - \{o_{out}\}$;
4 **if** $o_{out} \in R$ **then**
5 $R \leftarrow R - \{o_{out}\}$;
6 $R \leftarrow R \cup max_k(C \cup P_m^k \cup M_0)$;
7 **if** $min(P_m^k) \leq F(o_{in})$ *or* $|P_m^k| < k$ **then**
8 $P_m^k \leftarrow P_m^k \cup \{o_{in}\}$;
9 **if** $|P_m^k| > k$ **then**
10 $P_m^k \leftarrow P_m^k - \min(P_m^k)$;
11 $R \leftarrow$ update$(C \cup P_m^k \cup M_0)$;
12 **if** P_m *is full* **then**
13 remove partition P_0, and create a new partition;
14 $C \leftarrow$ Refine(C, P_m^k);
15 **if** $P_1.\rho < k$ **then**
16 $M_1 \leftarrow$ form(P_1);

partition), we insert P_m^k into C, and perform the necessary checking to remove false candidates from C via function Refine(C). In addition, we construct M_1 for P_1 if necessary, i.e., if $P_1.\rho < k$.

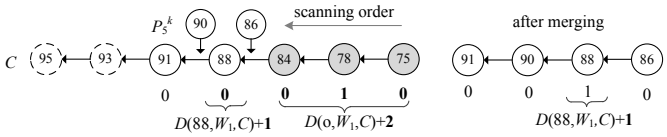


Fig. 4. Example of removable candidates disqualified by newly entered objects, where grey cycles refer to removable candidates and dotted cycles refer to outdated candidates.

Note that, we insert P_m^k into C via a merge operation, since the objects in C and those in P_m^k are preserved in two sorted lists, i.e., based on ascending order of their scores. During the insertion, we sequentially scan these two lists. As shown in Figure 4, objects in P_5^k are scanned based on the order of 86 first and then 90, and objects in C are scanned based on the order of 75 first, 78 second, and so on. We assume each candidate object o in C in the sliding window W_j is associated with a counter $D(o, C, W_j)$, initially set to 0. Its value refers to the number of candidate objects who can dominate o. When an i^{th} object $o_i \in P_m^k$ is inserted into C, all the objects behind it but before o_{i+1} (i.e., having scores lower than o_i but higher than the $(i+1)^{th}$ object $o_{i+1} \in P_m^k$) will have their associated counters increased by i. Once $D(o, C, W_j)$ reaches k, o is removed. Via integrating merging and refining, we need to scan C and P_m^k *only once* with the cost bounded by $O(|C|)$.

3.2 Discussion

Compared with other approaches, one immediate benefit of partition is that we could utilize, as superior as possible, a partition to catch "high-quality" candidates distributed at any part of the window, and to form a candidate set that covers all the query results as complete as possible. Even if not, we still could refine the candidate set efficiently using other nice properties of partition. Based on these properties, only a small number of insertions/deletions are required to maintain the candidates.

Take the-state-of-the-art algorithm MinTopK as an example. When window slides from W_1 to W_2, MinTopK requires 2 insertions (e.g., 90, 84) and 2 deletions (e.g., 89, 82) for incremental maintenance. Totally, when the window slides from W_1 to W_4, it incurs 7 insertions and 5 deletions[2]. Consider the partition in Figure 5. The newly arrival objects 90 and 84 consume 1 insertion respectively, while the following newly arrival objects, except 86, are all filtered out. Under this partition, it only requires 3 insertions (e.g,. 90, 84, 86) and 2 deletions (e.g,. 89, 84) from W_1 to W_4.

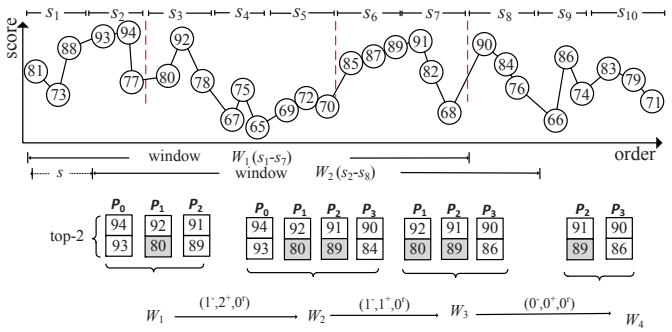


Fig. 5. An un-even partition, where grey rectangles refer to removable candidates, and u^-, v^+, w^r denotes the numbers of insertions, deletions, and re-scanning when the window slides respectively

However, a natural question is how to find such a superior partition. The smaller the partition size, the more the operations (e.g,. insertions and deletions), accompanied with a lower M_0 construction frequency, and vice versa. As will be presented in Section 4, we firstly attack this dilemma via utilizing the parameters k, n, and s. In this way, we successfully reduce the *incremental maintenance* cost to $O(\log k)$ and reduce the size $|C \cup M_0|$ to $O(k\sqrt{\frac{n}{max(k,s)}})$ even *in the worst case scenario* (see Section 4.1). Furthermore, we exploit a more superior algorithm, where it could self-adaptively adjust the partition size to cater for different streaming data distributions, e.g., the partition in Figure 5. We want to highlight that, this task is very challenging as we usually have no prior knowledge on objects' score distribution and it is impossible to learn the distribution of streaming data in the timely changing and high-speed streaming environment.

4 Partition Algorithms

After presenting the basic idea and main advantages of framework SAP, we need to address the issue of how to partition the window. In the following, we firstly present a simple equal partition with guaranteed incremental maintenance cost which serves as the baseline partition approach. Next, we present a dynamic partition that adjusts the size of each partition based on properties of underlying data streams.

Note, no matter how we partition the objects, s objects sliding into the window at the same time must be accommodated by one single partition, in order to guarantee the accuracy of group dominance number and to fulfill the requirement that all the objects in partition P_j arrive earlier than any object in partition P_{j+1}. In addition, each partition P_j shall contain at least k objects to ensure the existence of P_j^k. In other words, $\forall P_j \in \mathcal{P}(W, m)$, i) $|P_j| = i \cdot s$ with i an integer, and ii) $|P_j| \geq k$.

2. Note that we use the number of insertions and deletions to illustrate the basic ideas behind the algorithms without being distracted by the extra details.

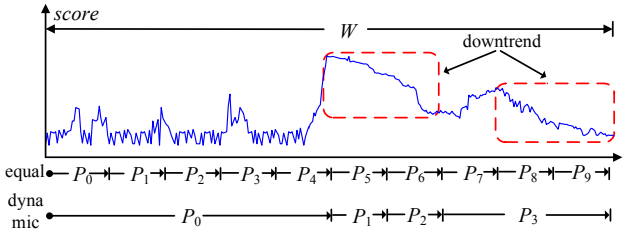


Fig. 6. Equal vs. Dynamic Partition ($N = 10^6, s = 10^4, k = 10$)

4.1 The Equal Partition Algorithm

In order to demonstrate the advantage of partition, we first present an equal partition algorithm. In the following, we develop a simple cost model to analyze the incremental maintenance cost under equal partition. As mentioned previously, we need to monitor $C \cup M_0$ in order to support the top-k search. Among every s objects that slide into the window simultaneously, there are maximum $\mathsf{min}(s, k)$ k-skyband objects and hence the size of M_0 is bounded by $\frac{|P_0|}{s} \cdot \mathsf{min}(s, k) - k = \frac{|P_0|k}{\mathsf{max}(s,k)} - k$. Accordingly, Equation (1) presents the upper bound of $|C \cup M_0|$ under equal partition.

$$|C \cup M_0| \leq \begin{cases} (m-1)k + \frac{N}{m} & k \geq s \\ (m-1)k + \frac{kN}{sm} & k < s \end{cases} \quad (1)$$

Note that the upper bound of $|C \cup M_0|$ could be minimized when $m = \lceil \sqrt{\frac{n}{\mathsf{max}(s,k)}} \rceil$. In the following, notation m^* is used to represent the value of $\lceil \sqrt{\frac{n}{\mathsf{max}(s,k)}} \rceil$. We want to highlight that, each partition has its minimum size of $\mathsf{max}(s, k)$ and its size must be a multiple of s. For example, if $\frac{n}{m} \leq 2s$, each partition is set to size s and m is then changed accordingly. In this case, equal partition under SAP framework is equivalent to MinTopK. In other words, MinTopK is a very special case of equal partition under our SAP framework. Instead of fixing the partition size to a constant number based on s like MinTopK does, our SAP framework is far more general and flexible.

Take Figure 6 as an example. Since $n = 10^6$ and $s = 10^4$, we partition the window into $\sqrt{\frac{10^6}{10^4}} = 10$ sub-windows. One benefit of equal partition is that we could avoid some extreme cases. For example, the scores of objects in P_5, P_6, P_8 and P_9 demonstrate a downtrend, and there are many meaningful objects in these four partitions. If we maintain all the meaningful objects in the window as *one-pass* algorithm does, the candidate set size could be huge. Because of partition, we only maintain the meaningful objects in $|C \cup M_0|$ which is bounded by $2 \times 10\sqrt{\frac{10^6}{10^4}}$ (i.e., 200).

Now, let's analyze the incremental maintenance cost. Besides reporting the results when they are changed, the incremental maintenance cost is mainly contributed by the merge cost of inserting P_m^k to C, the manipulation cost to manage meaningful objects in M_0, and the cost of maintaining P_m^k.

As for P_m^k, it takes $O(\log k)$ time to insert o_{in} into P_m^k. As for M_0, as will be reviewed in Section 5.1, we propose a novel structure named S-AVL to maintain the objects in M_0. The time complexity of constructing this structure is $O(\frac{n}{m} \times cost_F + \frac{n(k-P_0.\rho)}{ms} \log(k - P_0.\rho))$. In addition, we utilize some nice properties of partition for amortizing this part of cost within a period of time when $\frac{N}{m}$ objects slide out of the window. Therefore, when the window slides, the manipulation cost for M_0 is bounded by $O(\log k + cost_F)$ per object. The merge cost has the time complexity of $O(mk)$. After we amortize the merge cost within a period of time when $\frac{n}{m}$ objects slide out of

the window, the merge cost can be reduced to $O(\frac{m^2 k}{n})$. If m is set to $\lceil\sqrt{\frac{n}{\mathsf{max}(s,k)}}\rceil$ to minimize the upper bound of $|C \cup M_0|$, the merge cost is $O(1)$. After we consolidate these costs, the complexity of incremental maintenance cost is $O(\log k + cost_F)$ (when $m = \lceil\sqrt{\frac{n}{\mathsf{max}(k,s)}}\rceil$). We notice that under this setting, the complexity of incremental maintenance cost is *irrelevant* to the partition m and achieves a logarithmic complexity. Here, $cost_F$ refers to the cost of computing object's score.

In brief, through the above analysis, we understand that a simple equal partition with $m = \lceil\sqrt{\frac{N}{\mathsf{max}(k,s)}}\rceil$ can achieve an incremental maintenance cost bounded by $O(\log k + cost_F)$. To the best of our knowledge, among all the algorithms that can support continuous top-k query over sliding windows, this is the *first* algorithm that achieves logarithmic complexity w.r.t. k for incrementally maintaining the candidate set even *in the worst case scenarios*.

4.2 Dynamic Partition Algorithm

Our top-k algorithm requires to maintain the top-k objects of each partition P_i in P_i^k, and maintain all the k-skyband objects in $P_i - P_i^k$ using M_i. As explained previously, the incremental maintenance cost is caused by set $C \cup M_0$. If somehow we can predict that objects in P_i^k have a high chance to be non-k-skyband, we should enlarge the size of P_i to increase the possibility of including some real k-skyband objects in P_i^k. On the other hand, if we know that M_i is very likely to be non-empty, it is beneficial to reduce the size of P_i to decrease the likelihood of having a non-empty M_i. Obviously, the number of high-score objects in a partition has a direct impact on the proper size of a partition, which has not yet been considered by any existing structure designed for top-k searches over stream data, including the equal partition proposed in Section 4.1.

Take Figure 6 as an example. If we can predict that objects in P_0^k, P_1^k, P_2^k, P_3^k, and P_4^k have a high chance to be non-k-skyband, we could merge them together, and regard their union as one partition to effectively cut down the candidate maintenance cost. By contrast, if we can predict that M_5 corresponding to P_5 is very likely to be non-empty, we do not enlarge its size to keep the size of candidate set as low as possible. Clearly, this finding offers some room to further improve the partition, and the *dynamic partition algorithm* is the solution we propose. It is able to address the challenging issue of selecting a proper partition size, such that the chance of including k-skyband objects in P_i^k is high but that of including k-skyband objects in M_i is low.

Before we present the details of this new algorithm, we first introduce parameter l_{min} that refers to the *minimum* size of any partition P_i, i.e., $|P_i| \geq l_{min}$. Recall that, as analyzed in Section 4.1, the upper bound of $|C \cup M_0|$ under equal partition is $O(k\sqrt{\frac{n}{max(s,k)}})$ when $m = m^* = \lceil\sqrt{\frac{n}{\mathsf{max}(s,k)}}\rceil$. We set $l_{min} = \frac{n}{m^*} = \sqrt{n\mathsf{max}(s,k)}$ as any further reduction of the partition size will increase the upper bound of $|C \cup M_0|$. To simplify our discussion, we call a sub-window that contains l_{min} objects as *minimal partition unit* (in short *unit*).

Different from equal partition, the dynamic algorithm needs to decide a proper size for each new partition P_m. In this paper, we employ the *Mann-Whitney-Ranks Test* [15] (in short WRT) method to achieve this goal. The reason we adopt WRT is to construct an evaluation function $\mathcal{F}$ for evaluating whether objects in P_m^k (i.e., the top-k objects in partition P_m) have larger values, as compared with high-score objects in the search window within a time interval $\mathcal{I}$ that contains certain partitions prior to P_m. If the answer is yes, P_m might not be a proper partition as its probability of constructing M_i for P_m will be high in the future. Accordingly, we introduce Theorem 1, and parameter η which is the solution to $\frac{(\eta k - k)}{\sqrt{\eta k}} = 3$.

Theorem 1. Given a dataset $\mathcal{D}$ and two independent subsets SD_1 and SD_2, if $|SD_1| = \eta \cdot |SD_2|$, $\Pr(\theta_1^k > \theta_2^k) \approx 1$, where θ_1^k and θ_2^k refer to the k^{th} largest value of objects in SD_1 and SD_2 respectively, and η is the solution to $\frac{(\eta k - k)}{\sqrt{\eta k}} = 3$.

Proof. Please refer to Appendix B.

Based on Theorem 1 and WRT, we present our evaluation function $\mathcal{F}$ in Equation (2). It is based on top-k objects in P_m and top-ηk objects in the interval $\mathcal{I}$. Here, $\mathcal{I}^{\eta k}$ returns top-ηk objects in an interval $\mathcal{I}$ of the window W. If $\mathcal{F} > 0$, we conclude that candidates in P_m^k tend to have larger values than objects in $\mathcal{I}$ and the probability of constructing S-AVL for P_m might be high in the future. In other words, P_m is an improper partition and it most likely contains too many objects. Otherwise ($\mathcal{F} \leq 0$), partition P_m is proper and the probability of constructing S-AVL for P_m is very likely to be low in the future.

$$\mathcal{F}(P_m^k, \mathcal{I}^{\eta k}) \leq \begin{cases} R_1(P_m^k, \mathcal{I}^{\eta k}) - T_{up}(k, \eta k) & k \leq 10 \\ \frac{R_1(P_m^k, \mathcal{I}^{\eta k}) - \frac{k(k+\eta k+1)}{2}}{\sqrt{\frac{\eta k^2 (k+\eta k+1)}{12}}} - u_{1-\frac{\alpha}{2}} & k \geq 10 \end{cases} \tag{2}$$

The reason we compare top-k objects in P_m and top-ηk objects in $\mathcal{I}$ is to approximate the probability of $P_m.\rho \geq k$. Assume the current time stamp is t_0, we set $\mathcal{I}$ to $[t_0 - n + |P_m|, t_0)$. Note, we need to decide $P_m.\rho \geq k$ at the moment $t_0 + n - |P_m|$. That is to say the value of $P_m.\rho$ depends on the high score objects whose arrival times fall within the duration of $(t_0, t_0 + n - |P_m|]$. However, objects arrived within $(t_0, t_0 + n - |P_m|]$ are not available at time stamp t_0. As an alternative, we visit the top-ηk candidates whose arrival times are within $[t_0 - n + |P_m|, t_0)$ in the current window, i.e., $\mathcal{I} = [t_0 - n + |P_m|, t_0)$.

As a summary, our dynamic partition works as follows. It scans objects unit by unit. Every time when a new unit U of objects arrives, it merges U with current P_m to construct a new partition P'_m (i.e., $P'_m = P_m \cup U$), and then performs the evaluation $\mathcal{F}(P'^k_m, \mathcal{I}^{\eta k})$. If $\mathcal{F} \leq 0$ and $|P'_m| \leq l_{max}$, P'_m is a valid partition and it replaces P_m to become the current partition; otherwise ($\mathcal{F} > 0$ or $|P'_m| > l_{max}$), partition P'_m is invalid. We then finalize P_m as it is and a new partition P_{m+1} is initialized to accommodate the new unit of objects (i.e., $P_{m+1} = U$), which also becomes the current partition. Here, l_{max} refers to the *maximum* size of any partition P_i, i.e., $|P_i| \leq l_{max}$. It is set to the solution of $\frac{n - l_{max}}{l_{max}} = \eta$ to make sure function $\mathcal{I}^{\eta k}$ contains sufficient objects for the evaluation $\mathcal{F}$.

4.3 Enhanced Dynamic Partition Algorithm

Our dynamic partition algorithm is able to adjust the size of a partition as underlying data objects change. As compared with the equal partition that suggests $\frac{n}{m^*}$ as the partition size, the partitions under dynamic partition tend to have larger sizes as the size of $|P_i|$ could be several times larger than $l_{min} = \frac{n}{m^*}$. In this case, the upper bound of $|C \cup M_0|$ is no longer bounded by $O(k\sqrt{\frac{n}{\mathsf{max}(s,k)}})$. For example, a partition shown in Figure 7 actually contains 10 units (denoted as $U_0, \cdots, U_9$ in Figure 7). When

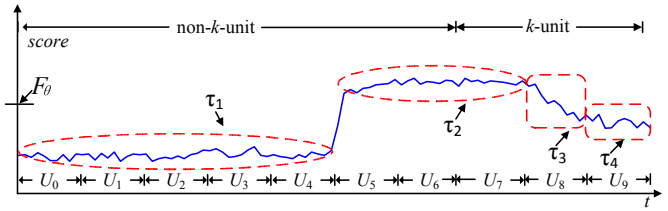


Fig. 7. The k-unit Identification

it is scanned, the k-skybands contained in these ten units all should be maintained, and $|M_0|$ cannot be bounded by $O(k\sqrt{\frac{n}{\max(s,k)}})$. The other observation is that, in order to construct M_0 for P_0, all ten units contained in P_0 need to be scanned. However, some units (e.g., U_0, U_1, U_2, U_3, U_4 in Figure 7) actually do not contain any k-skyband object. If we can somehow identify those units, the construction cost of S-AVL could be significantly reduced.

In order to address above two main issues, we further improve the dynamic partition algorithm with the new algorithm named as *enhanced dynamic partition algorithm* via a concept of k-unit. On one hand, enhanced dynamic partition algorithm wants to bound the size of M_0 to be $O(k\sqrt{\frac{n}{\max(s,k)}})$ even when P_0 is big. In other words, enhanced dynamic partition algorithm remains the self-adjustment capability of the dynamic partition algorithm but guarantees the size of M_0 is bounded by $O(k\sqrt{\frac{n}{\max(s,k)}})$. On the other hand, enhanced dynamic partition algorithm will also try to avoid accessing the units that contain no k-skyband. In the following, we first introduce the concept of k-unit, next present an approach that can tell whether a unit is a k-unit with a very low cost. In Section 5.2, we will explain how the enhanced dynamic partition algorithm can achieve above two objectives via k-unit.

k-unit. Given a unit U of size l_{min} (i.e., a partition under equal partition), let $U.u$ refer to the maximum number of k-skyband objects in U. A unit is labelled as k-unit if $U.u > O(k)$; or a non-k-unit otherwise (i.e., $U.u \leq O(k)$). Based on our observations, we understand that it is very likely for k-units to have non-empty S-AVL; while the chance for non-k-units to have non-empty S-AVL is much lower. Consequently, we group objects (in units) into non-k-units or k-units. In other words, we expect that the number of potential candidates from k-units that we need to maintain is very different from that under non-k-units. In enhanced dynamic partition algorithm, we maintain k objects with largest scores for each k-unit U_v, denoted as U_v^k; while we maintain the object with the highest score for each non-k-unit U_n, denoted as U_n^1.

The TBUI algorithm. After we present the concept of k-units, we need to address the issue of how to identify k-units. In order to find k-unit efficiently, we prefer to locate all k-units without calculating the values of $U_v.u$. First, we present an interesting finding in Theorem 2, which confirms a unit U_v cannot be a k-unit if the unit U_{v+1} right after U_v satisfies certain conditions. With the help of this finding, we propose the *threshold-based k-units identification algorithm* (in short TBUI) to locate k-units via maintaining a self-adaptive threshold τ.

Theorem 2. Given two adjacent units U_v and U_{v+1} in P_i and a threshold τ, if both units have $\omega(k)$ objects with scores higher than τ, U_v must be a non-k-unit.

Proof. Please refer to Appendix B.

According to Theorem 2, it is not hard to observe that the key difficulty of locating k-units lies in the selection of the threshold value τ. If τ is too small, many units will be labeled as k-units. On the other hand, if τ is too big, we might not be able to construct any non-k-units. In this paper, we select τ according to Theorem 3, i.e., the ζ^*-th highest score in a unit U_h. It indicates if the distribution of objects' scores in U_i is not very different from that of U_h $(i > h)$ (also U_{h+1}, U_{h+2},..., U_{i-1}), we can find $\omega(k)$ objects in U_i whose scores are higher than the ζ^*-th highest score in U_{i-1}, and hence U_{i-1} must be a non-k-unit. Otherwise, U_{i-1} may be a k-unit. Back to the example shown in Figure 7. Because the score distributions in U_0, U_1, U_2, U_3, U_4 are roughly the same, we could use the same value as the threshold. However, when we handle the objects in U_5, the score distribution in U_5 is very different from that in U_0. Therefore, the threshold τ has to be updated.

Theorem 3. Given two units of objects U_1 and U_2 with similar objects score distribution, $\Pr(k \leq |X| < \zeta_{max}) \approx 1$, where X refers to the set of objects in U_2 with scores higher than $F(\theta_1^{\zeta^*})$, $F(\theta_1^{\zeta^*})$ refers to the $(\zeta^*)^{th}$ highest score in U_1, ζ^* and ζ_{max} refer to the solution to $\frac{\zeta-k}{\sqrt{\zeta}} = 3$, and $\frac{\zeta_{max}-\zeta^*}{\sqrt{\zeta^*}} = 3$ respectively.

Proof. Please refer to Appendix B.

In the following, we will explain TBUI algorithm in detail, including *the threshold initialization*, *the threshold adjustment* and *k-unit maintenance*, with its pseudo code listed in Algorithm 2. Note the auxiliary structure L_i is a list of length $L_i.Length$, with each element $L_i[j]$ corresponding to the j^{th} unit of partition P_i. To be more specific, if U_j is a k-unit, $L_i[j]$ maintains U_j^τ; otherwise, $L_i[j]$ maintains the object in U_j with the highest score. Parameter $flag_i$ is to indicate whether the initialization of threshold τ is in operation.

The initialization of τ starts when objects of the first unit U_0 start sliding into the search window W. We set τ to be 0 at the beginning (Lines 1-2 of Algorithm 2). As objects of U_0 arrive, a subset U_0^τ of U_0 maintains all the objects in U_0 with scores higher than τ (Lines 3-4). When $|U_0^\tau|$ reaches $2\zeta^*$, we update τ to the median score of $2\zeta^*$ objects in U_0^τ using median-search algorithm [5], and meanwhile update U_0^τ to only include those objects with scores higher than new τ (Lines 5-6). Note ζ^* refers to the solution to $\frac{\zeta-k}{\sqrt{\zeta}} = 3$. As new objects arrive, U_0^τ is expanded and τ is updated again when U_0^τ reaches $2\zeta^*$. Once all the objects in the first unit U_0 arrive, we set τ to the ζ^*-th highest score in U_0^τ. In addition, we regard U_0 as a k-unit and construct U_0^k which is preserved by $L_i[0]$. Thereafter, $flag_i$ is set to $false$ to indicate the completion of the initialization of threshold τ.

In the following, objects of other units come. According to Theorem 3, if the distribution of objects' scores in the subsequent units is not *very different* from that of objects in U_0, we could find $\omega(k)$ objects in each of the subsequent unit with scores higher than τ. For example, when objects of U_1 arrive, we maintain a new subset U_1^τ to preserve all the objects in U_1 with scores higher than τ. If $k < |U_1^\tau| < \zeta_{max}$, we set U_1 as a k-unit but disqualify U_0 from being a k-unit based on Theorem 2. Accordingly, since U_0 is a non-k-unit, $L_i[0]$ only needs to remain the top-1 object in U_0 with the highest score but not U_0^τ (as listed in Lines 12-13); $L_i[1]$ is a k-unit so $L_i[1]$ will keep U_1^τ (as listed in Line 14). The process continues as new objects arrive, until one of the following two events happens.

(i) $|U_v^\tau| > \max(2\zeta^*, \zeta_{max})$ when $|U_v| < l_{min}$ (lines 7-8 of Algorithm 2). Based on Theorem 3, it implies that the scores of objects in U_v demonstrate an uptrend, as compared with scores of objects in previous units. Therefore, we update τ to the ζ^*-th highest score in U_v^τ, and update U_v^τ accordingly.

Algorithm 2: The TBUI Algorithm

Input: a partition P_i, newly arrived objects o, Boolean $flag_i$
Output: list L_i

1 **if** *o is the first object* **then**
2 $\tau \leftarrow 0$, $L_i \leftarrow \emptyset$, $L_i.length \leftarrow 0$, $flag_i \leftarrow true$;
3 **if** $F(o) \geq \tau$ **then**
4 $U_v^\tau \leftarrow U_v^\tau \cup o$;
5 **if** $flag_i = true \wedge |U_v^\tau| = 2\zeta^*$ **then**
6 $\tau \leftarrow$ med-search(U_v^τ, ζ^*), update$(U_v^\tau, \tau, \zeta^*)$;
7 **else if** $flag_i = false \wedge |U_v^\tau| >$ *max*$(2\zeta^*, \zeta_{max})$ **then**
8 $\tau \leftarrow$ med-search(U_v^τ, ζ^*), update$(U_v^\tau, \tau, \zeta^*)$;
9 $flag_i \leftarrow true$;
10 **if** $|U_v| = l_{min}$ **then**
11 **if** $|U_v^\tau| \geq k$ **then**
12 **if** $L_k.length \neq 0$ **then**
13 Int $max \leftarrow$ max$(L_i[L_i.length - 1])$;
$L_i[L_i.length - 1] \leftarrow max$;
14 form(U_v^k), $L_i[L_i.length++] \leftarrow U_v^k$, $flag_i \leftarrow false$;
15 **else**
16 $L_i[L_i.length++] \leftarrow U_v^\tau$, $flag_i \leftarrow true$,$\tau \leftarrow 0$;
17 return L_k;

(ii) $|U_v^\tau| < k$ when $|U_v| = l_{min}$ (lines 15-16 of Algorithm 2). Based on Theorem 3, it implies that the scores of objects in U_v demonstrate a downtrend, as compared with scores of objects in previous units. In this case, we re-initialize τ to 0, and re-use the initial τ initialization to find a proper value for τ. Unit U_{v-1} is labelled as a k-unit. Back to Figure 7, when $v = 8$, objects in U_8 demonstrate a declining tendency and U_7 is confirmed to be a k-unit; when $v = 9$, objects in U_9 also demonstrate a declining tendency and U_8 is confirmed to be another k-unit.

Complexity Analysis. We now analyze the computational complexity of the enhanced partition algorithm, which mainly consists of maintaining U_i^k and using the function $\mathcal{F}$ to evaluate whether the partition is proper. For the former task, given a unit U_i, when $|U_i^\tau|$ achieves max$(2\zeta^*, \zeta_{max})$, we spend $O(\zeta^*)$ cost in finding the median from U_i^τ, and deleting those with scores lower than the median. In the worst case, such operations are executed after max$(2\zeta^*, \zeta_{max})$ objects are processed. Because the size of each unit is l_{min}, the total cost of maintaining U_i^τ is bounded by $O(\zeta^* \frac{l_{min}}{\zeta^*}) = O(l_{min})$. For the latter task, the function $\mathcal{F}$ uses $O((k+1)\eta)$ computational cost in evaluating whether the partition is proper. Totally, the partition cost is bounded by $O(l_{min} + (k+1)\eta) = O(l_{min})$, which is $O(1)$ per object.

5 THE S-AVL STRUCTURE

In the above discussion, we have not yet explained how to construct the meaningful object set M_i for each partition P_i with $P_i.\rho < k$. In this section, we propose a novel structure named S-AVL to maintain the meaningful objects in P_i. We first introduce the S-AVL structure; we then discuss the enhanced S-AVL, which guarantees that $|C \cup M_0|$ is bounded by $O(k\sqrt{\frac{n}{max(s,k)}})$ regardless of the size of P_0.

5.1 The Baseline S-AVL Structure

As M_i is not required for supporting top-k searches until the objects of the corresponding partition P_i start sliding out of the window, we purposely delay the formation of M_i until P_i becomes the first partition of the current search window. Any early formation of M_i might turn out to be *avoidable*, because $P_i.\rho$ value could be increased as new objects slide into the window and it is only finalized when P_i becomes the first partition of the window. In this section, we propose a novel index structure called S-AVL to accommodate M_i. In the following, we explain the basic idea of S-AVL, the structure of S-AVL, the construction of S-AVL, the updating algorithm of S-AVL, and amortization of its construction. Last but not least, we will discuss the further optimization via considering s (see Appendix C).

Basic idea of S-AVL. For a partition P_0 with $P_0.\rho < k$, it means M_0 is not empty, i.e., there are some objects in $P_0 - P_0^k$ which might become result objects as objects of P_0^k slide out of the window. The basic idea of S-AVL is to locate all the meaningful objects in $P_0 - P_0^k$ and to maintain them in an efficient structure to facilitate the formation of M_i. S-AVL locates all the meaningful objects (i.e., those k-skyband objects in $P_i - P_i^k$) via two-level pruning, *global pruning* and then *local pruning*. In the global pruning, we locate the k^{th} object in the current candidate set without considering P_0^k (e.g., object with score 28 in Figure 8) and use its score (denoted as F_θ) to prune objects in P_0. Obviously, an object o with $F(o) < F_\theta$ will be dominated by top-k objects in $C - P_0^k$ and hence will not be a k-skyband object. In the local pruning, we only consider objects in P_0 and prune all the objects that are dominated by at least $(k - P_0.\rho)$ objects in P_0. Objects that pass both global pruning and local pruning are k-skyband objects, as stated in Lemma 2, and they will be maintained by S-AVL structure. Our main objective of S-AVL structure is to ease the identification and manipulation of those k-skyband objects.

Lemma 2. Given a partition $\mathcal{P}(W, m)$, let F_θ denote the k^{th} highest score of objects in $W - P_0$. Given an object $o \in P_0 - P_0^k$, let num_o be the number of objects in P_0 that dominate o (i.e., $num_o = |\{o' \in P_0 | o' \prec o\}|$). If $F(o) < F_\theta$ or $num_o \geq k - P_0.\rho$, object o is not a k-skyband object.

Structure of S-AVL. As plotted in Figure 8, S-AVL consists of stacks and an AVL tree that is constructed by the top entries of all the stacks. We purposely separate stacks from AVL-tree in our plot to ease the understanding, while in our implementation these two structures are well integrated with each node in AVL corresponding to a stack. The number of stacks is set to $k - P_0.\rho$. This is because for any non-candidate object $o \in P_0 - P_0^k$, there are at least $P_0.\rho$ candidates in $P - P_0$ dominating o. If o is dominated by another $(k - P_0.\rho)$ objects in P_0, o can be pruned. Consequently, by comparing an object with top entries of $k - P_0.\rho$ stacks, we can easily prune those objects locally dominated by at least $k - P_0.\rho$ objects (i.e., to facilitate the local pruning). That's why for the example shown in Figure 8, the number of stacks is set to 3 $(= k - P_0.\rho)$. Within each stack S_i, we ensure that following two conditions are always valid: i) $F(S_i[j]) \leq F(S_i[j+1])$; and ii) $S_i[j].t \geq S_i[j+1].t$. Note $S_i[1]$ ($S_i[|S_i|]$) refers to the bottom (top) entry of stack S_i. Take the stack S_2 listed under time stamp t_5 in Figure 8 as an example. The top entry has its value larger than that of the second entry ($35 > 34$) while the top entry arrives earlier than the second entry. We also want to highlight that not all the objects maintained by S-AVL are k-skyband as there might be some *false positive*. This is because an object might be dominated by multiple objects within a stack S_i while we only consider the top-entry in S_i when we perform local pruning. However, all the elements that are not indexed by S-AVL are not k-skybands. Here, we trade in the space for the efficiency and we can easily adjust

the checking process to make sure all the objects maintained in S-AVL are k-skybands.

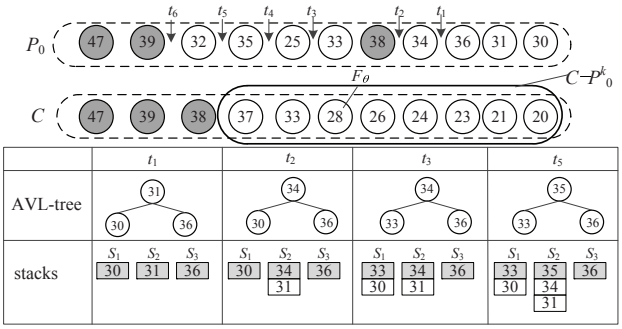


Fig. 8. S-AVL construction ($k = 3$ and $t_i > t_{i+1}$)

Construction of S-AVL. The construction of AVL tree is straightforward as it is built based on $(k - P_0.\rho)$ objects, the top entries of in total $(k - P_0.\rho)$ stacks. For stacks, because of the condition $S_i[j].t \geq S_i[j+1].t$, we scan the objects $o \in P_0 - P_0^k$ one by one according to descending order of $o.t$, i.e., *reverse of their arrival order*. For the first $(k - P_0.\rho)$ objects scanned, each of them forms a new stack, and an AVL tree is initialized when there are $(k - P_0.\rho)$ stacks. As shown in Figure 8 w.r.t. time stamp t_1, the first three objects scanned are 30, 31 and 36. Each of them forms one stack, and an AVL-tree is initialized by the three top entries. Thereafter, for each object o scanned, it is added into the stack S_i whose top entry has a smaller value than $F(o)$. If no such stack is available, o can be pruned as it is dominated by at least the $(k - P_0.\rho)$ top entries. Together with another $P_0.\rho$ objects in the current candidate set $C - P_0^k$, o is guaranteed to be a non-k-skyband object. If there are more than one stack satisfying this condition, we pick the one with the largest top entry value. For example, when object 34 is scanned, it can be added into both stack S_1 and stack S_2. As the top entry of S_2 has a value of 31 that is larger than the value (30) of the top entry of S_1, we add 34 to S_2 but not S_1. The reason we select S_2 for adding is that we do not want to change the order relationship among objects in the AVL-tree so we could avoid adjusting AVL-tree when construction.

Update of P_0^k based on S-AVL. Whenever a candidate object $o_{out} \in P_0^k$ slides out of the window, update on P_0^k is triggered if $P_0.\rho < k$. We find the object o' in S-AVL that has the largest value, delete o' from its original stack S_i, insert it into P_0^k (i.e., the candidate set C), and adjust AVL-tree via inserting the new head entry o'' from S_i, if available, to AVL-tree. The whole process takes $O(\log k)$ time.

Amortized Proactive Formation of S-AVL. Although S-AVL structure could efficiently maintain the k-skybands in $P_0 - P_0^k$, we would like to highlight that the performance of S-AVL could be improved further if we amortize the construction of S-AVL over a period of time when $\frac{n}{m}$ objects slide out of the window. Originally, we construct S-AVL w.r.t. P_0 when the first s objects $o \in P_0$ slides out of the window with its cost denoted as $cost$. Now, we amortize the formation into $\frac{n}{sm}$ steps, with each step scanning only s objects. Obviously, it can effectively reduce the cost from one-step cost of $cost$ to $\frac{n}{sm}$ step cost of $\frac{sm}{n}cost$ per step. To be more specific, we propose to construct S-AVL corresponding to P_0 in advance in an amortized fashion. We construct S-AVL of P_1 when objects slide out of P_0. Every time when s objects of P_0 slide out of the window, we check s objects in P_1 for the formation of S-AVL corresponding to P_1. When all the objects of P_0 slide out of the window, *all* the objects in P_1 are scanned and S-AVL w.r.t. P_1 has been constructed. Then, when objects of P_1 slide out of the window next, the corresponding S-AVL is ready and we scan objects of P_2 to get its S-AVL ready.

Note, amortization technique is general and it can be applied in operations other than construction of S-AVL, such as merge operation that merges C with P_m^k. Because of space limitation, we skip the details. On the other hand, we also want to highlight that amortization technique is general under our SAP framework because of some unique features brought by partitions, but it might not be applicable to other existing works. This also justifies the flexibility of SAP framework.

5.2 Segmentation-based S-AVL Construction

Recall that in Section 4.3, we develop the algorithm TBUI to label k-units and non-k-units. In this section, we propose the algorithm UBSA (short for unit-based segmentation algorithm) to construct S-AVL for P_i, which utilizes the information stored in L_i to bound the size of S-AVL and to avoid the unnecessary unit scanning. In addition, as will be reviewed in Theorem 4, $|M_i|$ is bounded by $O(k\sqrt{\frac{n}{\max(s,k)}})$ regardless of the size of P_i.

Given a partition P_i that contains j units $U_1, U_2, \cdots, U_j$ and has $P_i.\rho < k$, UBSA segments the construction of the S-AVL for P_i into two phases. In the first phase, only objects in non-k-units and the k objects with highest scores in each k-unit U_v are scanned. In other words, objects in $(U_v - U_v^k)$ of all the k-unit U_vs are skipped in the first phase but they will be scanned later. In order to avoid the unnecessary scanning, for each non-k-unit U_l, we access the corresponding element in L_i, i.e., $L_i[l]$, before scanning it. To be more specific, because $L_i[l]$ of a non-k-unit U_l only contains the object with the highest score, we compare $L_i[l]$ with F_θ. If $L_i[l] \leq F_\theta$, $U_l - P_i^k$ does not contain any k-skyband, and hence we can skip the scanning of U_l. Here, F_θ is the k^{th} highest score in $P_{i+1}^k \cup \ldots P_m^k \cup U_{l+1} \cup \ldots U_{length-1}$. Otherwise, we invoke the algorithm discussed in Section 5.1 to scan U_l to find k-skybands in U_l, and insert the k-skybands into the S-AVL of P_i. In Figure 7, in the first phase, non-k-units U_0, U_1, U_2, U_3, U_4, U_5, and U_6, and U_v^k of k-units U_7, U_8 and U_9 are scanned. Based on $L_i[l]$ value and F_θ, we are certain that non-k-units U_0, U_1, U_2, U_3, and U_4 do not have any k-skyband and their detailed scanning can be skipped.

In the second phase, we construct an independent S-AVL for each k-unit U_v in turn. To be more specific, when U_{v-2} slides out of the window, the scanning of U_v is triggered if U_v is a k-unit and its S-AVL is constructed. Theorem 4 guarantees that the size of M_i is bounded by $O(k\sqrt{\frac{n}{\max(s,k)}})$, at any time during its formation. Similar as constructing S-AVL for non-k-units, we also access the corresponding element in L_i, i.e.,$L_i[v]$, before scanning U_v. To be more specific, we find the lowest score $\min(L_i[v])$ in $L_i[v]$. If $\min(L_i[v]) < F_\theta$, we are certain that $L_i[v]$ has already included all the potential k-skybands, and hence scanning of $L_i[v]$ instead of U_v is sufficient to locate all the k-skybands. Otherwise, we scan the entire unit U_v to locate all the k-skybands in U_v. Back to the example shown in Figure 7. For k-unit U_7, the formation of its S-AVL starts when U_5 slides out of the window; for k-unit U_9, the formation of its S-AVL starts when U_7 slides out of the window. Right before the formation of S-AVL for U_9, we compare the minimum value of those preserved by $L_i[9]$ with F_θ. As $F_\theta > \min(L_i[9])$, the scanning of U_9 is actually skipped.

Theorem 4. Given a partition P_i that contains j units $\{U_1, U_2, \ldots, U_j\}$, if we maintain U_v^k $(1 \leq v \leq j)$ for each k-unit U_v, when P_i is scanned, the segmented formation of S-AVL could ensure that $|M_i|$ is bounded by $O(k\sqrt{\frac{n}{\max(s,k)}})$.

Proof. Please refer to Appendix B.

6 PERFORMANCE EVALUATION

In this section, we conduct extensive experiments to demonstrate the efficiency of SAP framework. The experiments are based on both real datasets and synthetic datasets. In the following, we first explain the datasets used in our experiments and the settings of our experiments, and then report our findings.

6.1 Experiment Settings

In total, five datasets are used in our experiments, including three real datasets, namely STOCK, TRIP and PLANET, and two synthetic datasets namely TIME^R and TIME^U. STOCK refers to 1GB stock transactions corresponding to $2,300$ stocks from ShangHai/ShenZhen Stock Exchange in last two years (from 2014 to 2015), with the original size of 30 GB. We cleaned the dataset by only considering four attributes (i.e., stock Id, transaction time, volume, and price), and then sorted all the cleaned records based on ascending order of their transaction time to form STOCK dataset. The preference function F is $price \times volume$.

TRIP contains 1GB taxi trip records from NYC in the last 6 years[3] (from 2010 to 2015). The size of the original dataset is 140 GB. We cleaned this dataset by considering four attributes (i.e., taxi Id, pick-up time t_p, drop-off time t_d, and travel distance dis), and then sorted all the cleaned records based on ascending order of their pick-up time. We use $\frac{dis}{t_d - t_p}$ as the preference function.

PLANET refers to the MPCAT-OBS available from the Minor Planet Center[4]. It contains 126 MB records with the original size of 9.51 GB. Each record r contains an observation coordinate and the preference function is set to the distance $dist(r, o)$ between a given point o and an observation coordinate of r.

Both TIME^R and TIME^U are synthetic. In TIME^R, objects' arrival orders are correlated with their scores, and scores of objects in TIME^R are decided by function $F(o) = sin(\frac{\pi \times o.t}{10^6})$ with $o.t$ $(= 1, 2, 3 \cdots)$ representing the arrival order of object o. In TIME^U, there is no correlation between the objects' arrival orders and their scores.

TABLE 1
Parameter Settings

Parameter	value
n	0.01%, 0.05%, **0.1%**, 0.2%, 0.5%, 1% ($\times|D|$)
k	10, 50, **100**, 200, 500, 1000
s	0.01%, 0.05% ,**0.1%**, 1%, 2%, 5%, 10% ($\times n$)

In our study, we consider three parameters, i.e., the window size n, the number of returned objects k, and the number s of new objects that slide into the window whenever the window slides. The parameter settings are listed in Table 1 with the default values bolded. In order to evaluate the performance of SAP under high speed stream environment, we conduct another set of experiments with different parameters, as to be presented in Appendix D.

The *total running time*, *average number of candidates* and *memory consumption* are employed as the main performance metrics. The total running time refers to the time used to process all the objects in the dataset. It contains the time of partitioning, handling newly arrival objects and constructing M_i for P_i if necessary, which indicates how fast the algorithms respond to the updates caused by window sliding; the average number of candidates refers to the average size of the candidate set C. We count the size of C when the window slides, and report its average size (see Appendix E). *Memory consumption* refers to the amount of memory consumed by different algorithms (see Appendix F).

In addition to SAP framework, we implement SMA, k-skyband, and MinTopK algorithms, three state-of-the-art approaches for answering continuous top-k query as competitors. SMA is a representative of multi-pass based approaches, k-skyband is a representative of one-pass based approaches, and MinTopK is another representative of one-pass based approaches. All the algorithms are implemented with C++, and all the experiments are conducted on a CPU i7 with 32GB memory, running Microsoft Windows 7.

6.2 Effect of Partitions

In our first set of experiments, we evaluate the effectiveness of partitioning algorithms proposed in Section 4. We first study the performance of SAP under equal-partition, and then we compare equal-partition with dynamic partition.

First, we report the running time of top-k query under equal-partition with different partition resolutions m in Table 2. All the parameters are set to their defaults. Recall that we introduce the basic Top-k search algorithm (i.e., Algorithm 1) in Section 3.1, and we also propose a new structure S-AVL to facilitate the formation of meaningful object set M_i in Section 5.1. In order to demonstrate the advantage of the *delay policy* discussed in Algorithm 1, i.e., delaying the formation of M_i for each P_i, we develop another partition algorithm named non-delay. Compared with Algorithm 1, non-delay also partitions the window into m parts. The only difference between Algorithm 1 and its non-delay version is that non-delay constructs P_i^k and M_i at the same time when P_i is constructed. Because of space limitation, we skip the details. In addition, we report the performance of Algorithm 1 in two different scenarios, with and without S-AVL in Table 2, to demonstrate the advantage of S-AVL structure.

Based on the result, we find that the *delay policy* is very effective. Under STOCK dataset, Algorithm 1 helps to significantly cut down the running time of non-delay, ranging from 66.4% under $m = 7$ to 40.5% under $m = 37$. The reason behind is that, for each partition P_i, $P_i.\rho$ is usually larger than k when $P_i.\rho$ is checked. Therefore, we only need to construct the meaningful object set for a few partitions, which significantly reduces the total computation cost. The S-AVL structure could further enhance the performance of top-k searching. Take PLANET dataset as an example. S-AVL structure helps to cut down the running time of Algorithm 1, ranging from 77.6% under $m = 7$ to 90% under $m = 37$. We also observe that the power of S-AVL drops as m value increases. The reason behind is that S-AVL structure helps to improve the formation of M_i. However, the likelihood of M_i being empty is increased as the partition becomes smaller (i.e., m becomes bigger).

In addition to the running time, we also report the value of m^* in Table 2. Note m^* refers to $\lceil\sqrt{\frac{n}{\max(s,k)}}\rceil$, the value of m suggested by the cost model presented in Section 4.1 to minimize the upper bound of candidate set size. The bold values are the real minimal running time under different m values. However, based on the running time under different m values reported in Table 2, the running time of the equal partition under $m = m^*$ value

3. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml
4. http://www.minorplanetcenter.net/

TABLE 2
Running time of equal-partition under different ms (unit: second, $m^* = \lceil\sqrt{\frac{n}{\max(s,k)}}\rceil$, $n = 0.1\%|D|$, $k = 100$, $s = 0.1\%n$)

Dataset	m	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37
STOCK	non-delay	53.6	46.3	42.8	40.5	38.5	36.3	34.6	33.5	31.6	29.7	28.6	27.7	28.1	27.8	27.6	27.3	27.2
$m^* = 33$	Algo 1	18.2	15.6	14.9	14.8	14.3	14.2	13.7	13.8	13.9	14.4	14.8	15.2	15.5	15.7	16.0	16.4	16.6
	Algo 1+S-AVL	14.5	12.6	12.5	**12.5**	12.6	12.7	12.8	13.1	13.4	13.9	14.0	14.8	15.3	15.4	15.8	16.1	16.1
TRIP	non-delay	55.3	45.3	43.1	42.9	41.7	40.3	39.1	38.2	36.7	34.1	32.9	31.8	30.7	28.1	27.9	27.3	27.4
$m^* = 33$	Algo 1	14.9	14.2	14.0	14.2	14.7	14.9	15.4	15.6	15.8	16.2	16.6	16.8	17.2	17.7	18.1	18.3	18.6
	Algo 1+S-AVL	12.8	**11.7**	11.9	12.1	12.6	12.9	13.3	13.9	14.3	14.8	15.1	15.3	15.6	15.9	16.1	16.6	17.3
PLANET	non-delay	6.2	5.3	5.1	5.0	4.9	4.7	4.5	4.4	4.2	4.3	4.3	4.4	4.4	4.5	4.6	4.6	4.7
$m^* = 11$	Algo 1	5.0	4.4	4.3	4.2	4.1	4.1	4.1	4.0	4.0	4.0	3.8	3.8	3.7	3.7	3.7	3.7	3.7
	Algo 1+S-AVL	3.6	3.2	3.0	2.9	**2.9**	2.9	2.9	3.0	3.1	3.1	3.6	3.2	3.2	3.3	3.3	3.3	3.4
TIME^U	non-delay	35.6	34.7	33.2	31.9	30.8	29.9	27.2	27.5	27.9	28.1	28.8	29.7	30.7	31.3	32.6	33.3	33.7
$m^* = 33$	Algo 1	13.2	12.7	12.8	13.2	13.5	14.2	14.4	15.1	15.2	15.7	16.2	16.8	17.2	17.3	17.9	18.4	18.3
	Algo 1+S-AVL	12.9	**12.5**	12.7	13.1	13.4	14.1	14.4	14.9	15.2	15.6	16.1	16.4	16.9	17.3	17.5	17.7	18.1
TIME^R	non-delay	90.3	88.6	86.4	83.2	81.6	78.8	76.5	75.9	74.3	74.1	73.3	72.8	71.7	70.9	71.1	71.7	72.1
$m^* = 33$	Algo 1	73.0	67.5	63.4	60.2	57.6	57.4	55.5	54.9	53.4	52.5	51.3	50.8	49.3	48.8	48.3	47.7	46.2
	Algo 1+S-AVL	57.4	51.6	47.2	46.9	46.6	45.6	44.7	43.9	43.6	42.7	42.3	41.9	41.7	41.4	41.3	**41.2**	41.3

TABLE 3
Running time comparison of dynamic partition and equal-partitions (unit: second)

Data set		**parameter** $n(\times\|D\|)$					**parameter** k					**parameter** $s(\times n)$					
		0.01%	0.05%	0.1%	0.5%	1%	10	50	100	500	1000	0.01%	0.05%	0.1%	1%	5%	10%
STOCK	EN-DYNA	11.2	11.8	12.5	13.5	14.3	9.58	11.2	12.5	14.4	15.4	19.7	15.5	12.5	11.3	10.6	10.1
	DYNA	11.9	12.4	13.7	15.0	15.7	10.64	12.3	13.7	17.1	19.2	26.7	16.5	13.7	12.8	10.8	10.1
	EQUAL	14.9	15.3	15.8	16.8	18.2	12.3	13.8	15.8	23.2	36.2	38.8	27.6	15.8	13.2	10.7	10.2
TRIP	EN-YNA	10.4	10.9	12.2	12.9	13.8	9.5	10.1	12.2	13.4	20.4	15.6	13.9	12.2	11.2	11	10.3
	DYNA	10.7	11.0	12.5	13.1	14.1	9.7	10.2	12.5	13.8	21.1	16.7	14.4	12.5	11.2	11.1	10.4
	EQUAL	15.2	15.9	16.1	18.6	20.3	11.7	12.9	16.1	18.9	26.7	33.6	24.1	16.1	13.4	11.2	10.3
PLANET	EN-DYNA	1.3	1.5	1.6	2.5	2.6	0.9	1.3	1.6	2.2	2.8	2.2	2.1	1.6	1.5	1.8	1.2
	DYNA	1.8	2.0	2.4	3.0	3.1	1.1	1.8	2.4	2.8	3.8	3.3	2.9	2.4	2.0	1.8	1.3
	EQUAL	1.9	2.3	2.9	3.1	3.3	1.1	2.0	2.9	3.0	4.2	3.9	3.1	2.9	2.2	1.8	1.3
TIME^U	EN-DYNA	10.4	10.9	12.2	12.9	13.8	9.5	10.1	12.2	13.4	20.4	15.6	13.9	12.2	11.27	11	10.3
	DYNA	10.5	11.1	12.3	13.0	14.2	9.6	10.1	12.3	13.5	20.7	15.8	14.1	12.3	11.3	11.03	10.4
	EQUAL	16.6	17.1	17.5	19.6	20.3	13.8	16.7	17.5	23.4	34.2	27.9	21.1	17.5	14.1	11.2	10.4
TIME^R	EN-DYNA	32.7	28.1	24.1	17.6	17.2	18.8	21.2	24.1	33.7	39.4	41.3	29.8	24.1	19.8	17.8	15.4
	DYNA	46.7	43.1	38.1	16.3	15.7	23.8	32.2	38.1	46.7	54.4	55.3	45.8	38.1	22.8	17.8	15.6
	EQUAL	50.4	47.5	41.1	36.4	34.8	25.2	33.7	41.1	48.8	56.7	59.4	47.6	41.1	23.4	17.9	15.5

in most cases is not optimized, which suggests that m^* that is supposed to minimize the upper bound of candidate set size does not necessarily minimize the running time, and approximating the running time of top-k search over streaming data is not easy as the search performance depends on many factors, such as the score distribution of the underlying stream and the search parameters. As S-AVL structure is effective, Algorithm 1 with S-AVL is adopted as default implementation for the search algorithm under SAP.

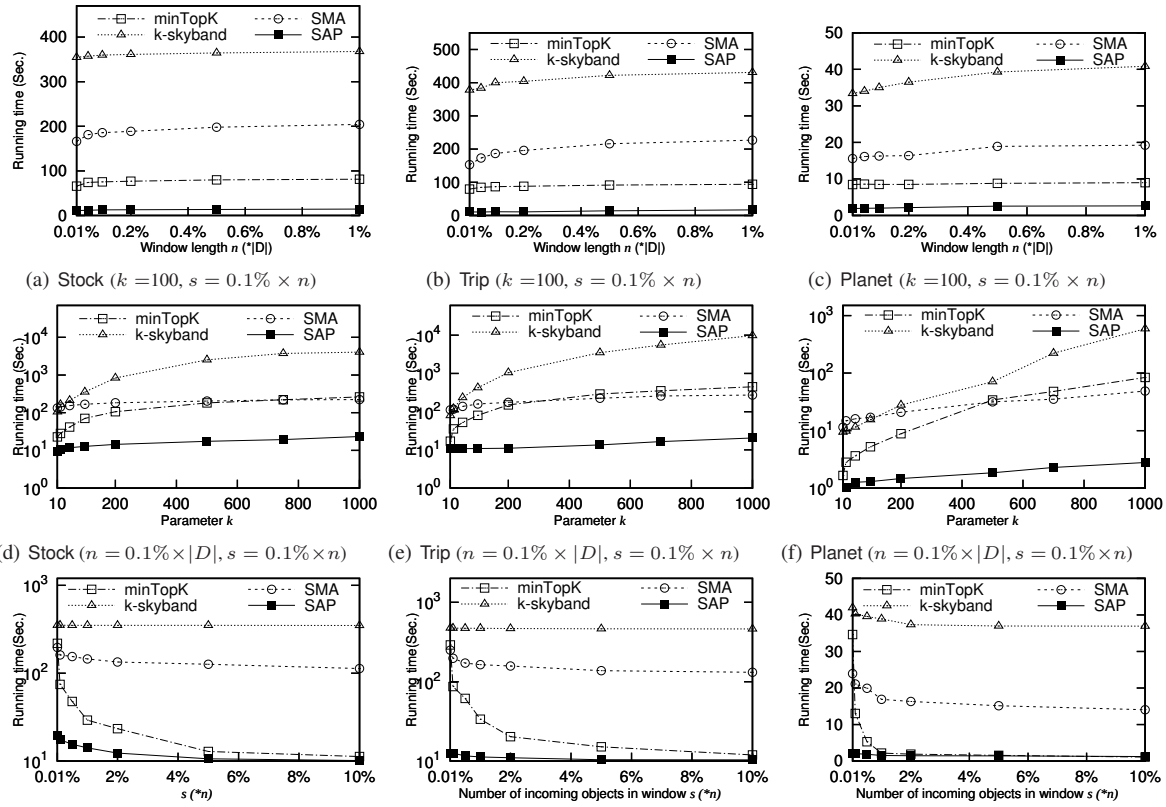
Second, we compare the equal partition with both the dynamic partition and enhanced dynamic partition, denoted as EQUAL DYNA, and EN-DYNA respectively, with the results reported in Table 3. We observe that enhanced dynamic partition performs significantly better than equal-partition. Take TRIP dataset as an example. SAP under dynamic partition only takes about 70% of the running time of SAP under equal partition when N changes its value from 0.01% to 1%. In addition, we find that EQUAL is more sensitive to the parameters such as k, n, and s, while DYNA is able to achieve a more stable performance. Take STOCK dataset as an example. When k varies its value from 10 to 1000, SAP under dynamic partition changes its running time from 9.58 seconds to 23.4 seconds, an increase of 122%, while SAP under equal partition changes its running time from 12.3 seconds to 36.2 seconds, an increase of 194%; when s varies its value from $0.01\%n$ to $10\%n$, SAP under dynamic partition changes its running time from 19.7 seconds to 10.1 seconds, a decrease of 49%, while SAP under equal partition changes its running time from 38.8 seconds to 10.2 seconds, a decrease of 73%.

We also observe that enhanced dynamic partition performs better than dynamic partition. Take STOCK dataset as an example. SAP under EN-DNYA averagely takes about 90% of the running time of SAP under DNYA. More importantly, we find that EN-DNYA is more stable to the data distribution. For PLANET and TIME^R datasets, the running time of SAP under EN-DNYA is 70% of that of SAP under DNYA. The reason behind is that when the scanning of P_0 frequently happens, our proposed EN-DNYA helps us effectively reduce the scanning scope. In general, enhanced dynamic partition is a better option.

As a summary, structure S-AVL is very effective and it can help to improve the performance of SAP. In addition, the distribution of data stream does affect the performance of top-k searches, and enhanced dynamic partition performs better than both the equal-partition and the dynamic partition.

6.3 Comparison between SAP and existing algorithms

In our second set of experiments, we compare the performance of SAP with its competitors. Note we adopt the enhanced dynamic partitioning as the default implementation for SAP in the following experimental study, with the implementation of S-AVL. The running time of different algorithms is reported in Figure 9, corresponding to STOCK, TRIP, and PLANET, the three real datasets used in our experiments. We firstly report the running time of all the algorithms under different n values in Figures 9(a)-9(c). Among all the n values evaluated, we can observe that SAP outperforms existing algorithms consistently, for all three datasets. For example, SAP on average consumes only 17% of MinTopK's, 6% of SMA's, and 3.5% of k-skybands's running time respectively. The significant improvement lies on the fact that SAP could self-adaptively partition the window, and select "high quality" candidates to answer the query. Consequently, fewer insertion/deletion operations are performed, as compared with MinTopK. As compared with SMA, SAP does not need to index all the objects via grid file and hence avoid multi-scanning



(a) Stock (k =100, $s = 0.1\% \times n$) (b) Trip (k =100, $s = 0.1\% \times n$) (c) Planet (k =100, $s = 0.1\% \times n$)

(d) Stock ($n = 0.1\% \times |D|, s = 0.1\% \times n$) (e) Trip ($n = 0.1\% \times |D|, s = 0.1\% \times n$) (f) Planet ($n = 0.1\% \times |D|, s = 0.1\% \times n$)

(g) Stock ($n = 0.1\% \times |D|, k$ =100) (h) Trip ($n = 0.1\% \times |D|, k$ =100) (i) Planet ($n = 0.1\% \times |D|, k$ =100)

Fig. 9. Running time comparison of different algorithms under real data set

of the window whenever $k^{'}$ becomes smaller than k [17]. Lastly, as compared with k-skybands, SAP saves the maintenance of the dominance relationship of all the k-skybands in the window.

We also report the running time of different algorithms under various k values in Figures 9(d)-9(f). Similar as above observations, SAP consistently performs best. We also observe that, as k increases, the running time of both MinTopK and k-skyband goes up rapidly. This is because k-skyband spends $O(k)$ computation cost in removing a non-skyband; while, for MinTopK, the value of $\frac{k}{s}$ becomes bigger with the increase of k, which means fewer objects could be discarded when they arrive in the window. In addition, SAP shows a much more stable performance under various parameter settings, as compared with its competitors. It demonstrates that SAP is resilient to multiple parameters.

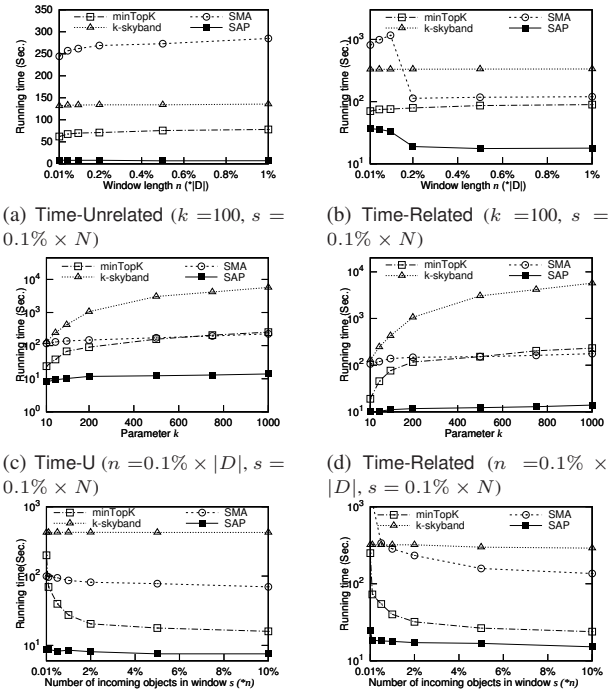
The running time of different algorithms under different s is reported in Figures 9(g)-9(i). We find that SAP performs best again. In addition, SAP is not sensitive to s values, which is contributed by the self-adaptive nature of the partition algorithms employed by SAP. By contrast, as s increases, MinTopK's running time rapidly drops. This is because the larger the s, the lower the proportion of objects that needs to be maintained. When s is $0.001\% \times n$, the running time of MinTopK is almost 19 times higher than that of SAP. When s is $10\% \times n$, the running time of MinTopK is roughly the same as that of SAP. Like MinTopK, SMA also decreases its running time as s increases but at a much slower pace. The reason behind is that the larger the s, the lower the re-scanning frequency with the total number of re-scanning bounded by $\frac{|D|}{s}$. However, SMA has to bear the expensive maintenance cost of its grid file structure that is independent on s. Consequently, s has a much smaller impact on its running time, as compared with MinTopK. k-skyband does not consider the factor s, and hence its running time remains very expensive regardless of s values.

We then report the running time of all the algorithms under different synthetic data sets. As depicted in Figure 10(a)-10(f), SAP outperforms all the competitors again. One important observation is SAP can effectively avoid several extreme cases since SAP considers the unique properties of data streams and adjusts its settings to cater for underlying data streams. For example, in Figure 10(a), when the window size is smaller than the period of the periodic function (e.g., 2MB for function $sin(\frac{o.t\pi}{10^6})$), SMA requires long running time because it has to incessantly apply re-scanning. By contrast, SAP only constructs the S-AVL structure for a small part of partitions.

As a summary, SAP is very efficient. Compared with its competitors, it takes the lowest running time to support the top-k search. In addition, SAP has the ability to adjust the partition size to cater for queries with different parameters and data streams with different distributions.

7 Conclusion

In this paper, we propose a novel and general framework namely SAP, for supporting continuous top-k query over stream data. Different from all the existing works, SAP employs the partition technique to organize the stream data. It can adjust the partition size to cater for data streams with different distribution and queries with different parameters. We have conducted extensive experiments to evaluate the performance of SAP on several datasets with different distributions. The results demonstrate the superior performance of SAP. In the near future, we would like to explore more distribution models and propose partitioning algorithms accordingly.



(a) Time-Unrelated (k =100, s = $0.1\% \times N$)

(b) Time-Related (k =100, s = $0.1\% \times N$)

(c) Time-U (n =$0.1\% \times |D|$, s = $0.1\% \times N$)

(d) Time-Related (n =$0.1\% \times |D|$, $s = 0.1\% \times N$)

(e) Time-Unrelated (n =$0.1\% \times |D|$, k =100)

(f) Time-Related (n =$0.1\% \times |D|$, k =100)

Fig. 10. Running time comparison under synthetic data set

REFERENCES

[1] M. Ajtai, T. S. Jayram, R. Kumar, and D. Sivakumar. Approximate counting of inversions in a data stream. In *STOC*, pages 370–379, 2002.

[2] C. Böhm, B. C. Ooi, C. Plant, and Y. Yan. Efficiently processing continuous k-nn queries on data streams. In *ICDE*, pages 156–165, 2007.

[3] L. Cao, J. Wang, and E. A. Rundensteiner. Sharing-aware outlier analytics over high-volume data streams. In *SIGMOD*, pages 527–540, 2016.

[4] L. Cao, D. Yang, Q. Wang, Y. Yu, J. Wang, and E. A. Rundensteiner. Scalable distance-based outlier detection over high-volume data streams. In *ICDE*, pages 76–87, 2014.

[5] L. Cormen, Thomas H. *Introduction to Algorithms The Third Edition*. The MIT Press, 2009: 213-223.

[6] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas. Ad-hoc top-k query answering for data streams. In *VLDB*, pages 183–194, 2007.

[7] B. Gedik, K. Wu, P. S. Yu, and L. Liu. A load shedding framework and optimizations for m-way windowed stream joins. In *ICDE*, pages 536–545, 2007.

[8] L. Golab, D. DeHaan, E. D. Demaine, A. López-Ortiz, and J. I. Munro. Identifying frequent items in sliding windows over on-line packet streams. In *IMC*, pages 173–178, 2003.

[9] X. Gu, P. S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In *ICDE*, pages 146–155, 2007.

[10] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *STOC*, pages 471–475, 2001.

[11] P. Haghani, S. Michel, and K. Aberer. Evaluating top-k queries over incomplete data streams. In *CIKM*, pages 877–886, 2009.

[12] M. Kontaki, A. Gounaris, A. N. Papadopoulos, K. Tsichlas, and Y. Manolopoulos. Continuous monitoring of distance-based outliers over data streams. In *ICDE*, pages 135–146, 2011.

[13] K. C. K. Lee, B. Zheng, C. X. Chen, and C. Chow. Efficient index-based approaches for skyline queries in location-based applications. *TKDE*, 25(11):2507–2520, 2013.

[14] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *ICDE*, pages 502–513, 2005.

[15] M. J. Morris H. DeGroot. *Probability and Statistics*. China Machine Press, 2012.

[16] M. J. Morris H. DeGroot. *Probability and Statistics*. China Machine Press, 2012:535.

[17] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD*, pages 635–646, 2006.

[18] K. Mouratidis and D. Papadias. Continuous nearest neighbor queries over sliding windows. *TKDE*, 19(6):789–803, 2007.

[19] Z. Shen, M. A. Cheema, X. Lin, W. Zhang, and H. Wang. Efficiently monitoring top-k pairs over sliding windows. In *ICDE*, pages 798–809, 2012.

[20] C. Sheng and Y. Tao. On finding skylines in external memory. In *PODS*, pages 107–116, 2011.

[21] M. Stern, K. Böhm, and E. Buchmann. Processing continuous join queries in sensor networks: a filtering approach. In *SIGMOD*, pages 267–278, 2010.

[22] L. A. Tang, Y. Zheng, J. Yuan, J. Han, A. Leung, C.-C. Hung, and W.-C. Peng. On discovery of traveling companions from streaming trajectories. In *ICDE*, pages 186–197, 2012.

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

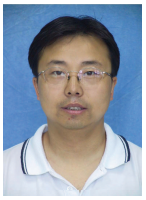
[24] E. W. Weisstein. de moivre-laplace theorem. From MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/deMoivre-LaplaceTheorem.html.

[25] D. Yang, A. Shastri, E. A. Rundensteiner, and M. O. Ward. An optimal strategy for monitoring top-k queries in streaming windows. In *EDBT*, pages 57–68, 2011.

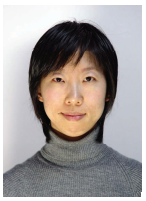
[26] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, pages 189–200, 2003.

[27] A. Yu, P. K. Agarwal, and J. Yang. Processing a large number of continuous preference top-k queries. In *SIGMOD*, pages 397–408, 2012.

Rui Zhu received the M.Sc degree in computer science from the Department of Computer Science, Northeastern University, P.R.China, in 2008. Currently, he is a Ph.D. candidate of Northeastern University. His research interests include design and analysis of algorithms, databases, data quality, and distributed systems.



Bin Wang received the PhD degree in computer science from Northeastern University in 2008. He is currently an associate professor in the Computer System Institute at Northeastern University. His research interests include design and analysis of algorithms, queries processing over streaming data, and distributed systems



Xiaochun Yang received the PhD degree in computer science from Northeastern University, China, in 2001. She is a professor in the Department of Computer Science at Northeastern University, China. Her research interests include data quality, and data privacy. She is a member of the ACM, the IEEE Computer Society, and a senior member of the CCF.

Baihua Zheng is an associate professor in the School of Information Systems, Singapore Management University. She received her her PhD degree in computer science from Hong Kong University of Science and Technology. Her research interests include data management and mobile/pervasive computing. She has published more than 100 technical papers in these areas. She is a member of the IEEE.

Guoren Wang received Ph.D. from the Department of Computer Science, Northeastern University, P.R.China, in 1996. Currently, he is a professor in the Department of Computer Science, Northeastern University, P.R.China. His research interests are XML data management, query processing and optimization.

APPENDIX A

This section briefly explains the *partition algorithms* under time-based window. A time-based query window $W\langle n, s\rangle$ consists of a set of objects in $\{s_1, \cdots, s_{\frac{n}{s}}\}$. Different from count-based window, time-based window does not have the same number of objects in each window/slide. Objects in each s_i have the same arrival time and only the top-k objects have the chance to become query results. Therefore, the query results must be bounded by a subset that contains $\frac{nk}{s}$ objects, and we could employ the partition algorithms proposed in Section 4 for partitioning the time-based window.

In the following, we first discuss the equal partition algorithm. For simplicity, we assume that the window W is partitioned into $\{P_0, P_1, \ldots, P_{m-1}\}$ and each P_i contains the objects in $\{s_{\frac{n\times i}{ms}}, \cdots, s_{\frac{n\times(i+1)}{ms}-1}\}$. The goal of equal partition is to minimize the upper-bound of $|C|$ according to query parameters. Under time-based window, we still could bound the size of candidate set C to mk. For M_0, because the number of meaningful objects in each s_i is bounded by k, $|M_0|$ is bounded by $\frac{nk}{sm}$. Therefore, $|C \cup M_0|$ is bounded by $mk + \frac{nk}{sm}$ and the upper bound of $|C \cup M_0|$ could be minimized when $m = \lceil\sqrt{\frac{n}{\max(s,k)}}\rceil$. Consequently, our goal of equal partition could be achieved under time-based window.

Next, we discuss the dynamic partition algorithm. Its key idea is to evaluate whether the partition is proper based on the function $\mathcal{F}(P'^k_m, \mathcal{I}^{\eta k})$. Since this function only concerns a few high score objects in P'_m and $\mathcal{I}$, this algorithm is unrelated with the window type. To be more specific, this algorithm scans objects unit by unit. Every time when a new unit U of objects arrives, it merges U with current P_m to construct a new partition P'_m (i.e., $P'_m = P_m \cup U$), and then performs the evaluation $\mathcal{F}(P'^k_m, \mathcal{I}^{\eta k})$. Because $|P_m| > k$ and $|\mathcal{I}| > k$, they could provide enough elements for evaluation, and the dynamic partition algorithm is also workable under time-based window.

Last but not least, we discuss the enhanced dynamic partition algorithm. As presented in Section 4.3, the key of the enhanced dynamic partition algorithm is to maintain a threshold τ to differentiate k-units from non-k-units. Because the update on τ is only dependent on the size of U_i^τ, Algorithm 2 is workable under time-based window as long as $|U_i| >> \tau$. To be more specific, we still could initialize the threshold τ using Lines 1-2 and Lines 3-6 of Algorithm 2. In addition, when objects of other units (e.g., U_v) come, we decide whether to update τ according to the size of U_v^τ (see Lines 7-9 and Lines 10-16 of Algorithm 2).

APPENDIX B

The Proof Theorem 1. If both $|SD_1|$ and $|SD_2|$ are big enough, we could use normal distribution $N(np, \sqrt{np(1-p)})$ to approximate $\Pr(|SD_2^{\theta_1^k}| \leq \zeta)$ according to *demovire-laplace theorem* [24], where $p = \frac{k}{|SD_1|}$, $n = |SD_2|$, and $SD_2^{\theta_1^k}$ refers to the set of objects in SD_2 with values larger than θ_1^k (i.e., $SD_2^{\theta_1^k} = \{o | o \in SD_2 \wedge F(o) > \theta_1^k\}$). Since $\Pr(\theta_1^k > \theta_2^k) = \Pr(|SD_2^{\theta_1^k}| < k) \approx \Phi(\frac{k-np}{\sqrt{np(1-p)}}) \approx \Phi(\frac{\eta k - k}{\sqrt{\eta k}})$, if $\frac{(\eta k - k)}{\sqrt{\eta k}} \geq 3$, $\Pr(\theta_1^k > \theta_2^k) \approx 1$ according to 3-sigma-rule. □

The Proof Theorem 2. Given two adjacent units U_v and U_{v+1}, U_v^τ refers to a set of objects in U_v with scores higher than τ, i.e., $U_v^\tau = \{o | o \in U_v \wedge F(o) > \tau\}$. $\forall o' \in U_v - U_v^\tau$ and $\forall o \in U_{v+1}^\tau$, it must satisfy that $o \prec o'$. Since $|U_{v+1}^\tau| = \omega(k)$, all the objects in $U_v - U_v^\tau$ are dominated by at least $\omega(k)$ objects in U_{v+1} and hence will not be k-skybands. In other words, $U_v.u$ is bounded by $|U_v^\tau|$ and hence it must be a non-k-unit. □

The Proof Theorem 3. Like Theorem 1, we could use normal distribution $N(np, \sqrt{np(1-p)})$ to approximate $\Pr(|U_2^{\theta_1^{\zeta^*}}| \leq k)$, where $p = \frac{\zeta^*}{|U_1|}$, $n = |U_2|$, $np \geq 5$ and $U_2^{\theta_1^{\zeta^*}}$ refers to the set of objects in U_2 with scores higher than $F(\theta_1^{\zeta^*})$. Therefore, $\Pr(|U_2^{\theta_1^{\zeta^*}}| \leq k) \approx \Phi(\frac{k-\zeta^*}{\sqrt{\zeta^*}}) = 1 - \Phi(\frac{\zeta^*-k}{\sqrt{\zeta^*}})$. According to 3-sigma-rule, if ζ^* is the solution of $\frac{\zeta-k}{\sqrt{\zeta}} = 3$, $\Phi(\frac{\zeta^*-k}{\sqrt{\zeta^*}}) \approx 1$, and we have $\Pr(|X| \leq k) \approx 0$. Similarly, we have $\Pr(|X| \geq \zeta_{max}) \approx 1 - \Phi(\frac{\zeta_{max}-\zeta^*}{\sqrt{\zeta^*}})$. If we set ζ_{max} to the solution of $\frac{\zeta_{max}-\zeta^*}{\sqrt{\zeta^*}} = 3$, $\Pr(|X| \leq \zeta_{max}) \approx 1$. □

The Proof Theorem 4. Let M_{non} be a set of k-skybands in the non-k-units. Because the number of k-skybands in each non-k-unit is bounded by $O(k)$, the total number of k-skybands in M_{non} is bounded by $O(k\frac{l_{max}}{l_{min}}) = O(k\sqrt{\frac{n}{\max(s,k)}})$. Let M_k be a subset k-skybands in the k-units. Because the segmented formation of S-AVL ensures that we maintain *at most* two k-units' k-skybands, together with U_v^k for each k-units, M_k is also bounded by $O(k\sqrt{\frac{n}{\max(s,k)}})$. Because $|M_i| = |M_i|.non + |M_i|.k$, the segmented formation of S-AVL could ensure that $|M_i|$ is bounded by $O(k\sqrt{\frac{n}{\max(s,k)}})$. □

APPENDIX C

Under sliding window model, since only the k objects with highest scores in each s objects have the chance to become the query results, S-AVL could be further optimized via considering the parameter s. To be more specific, given the partition P_1 $\{s_0, s_1, \ldots, s_j\}$, if $P_1.\rho < k$, we construct M_1 for P_1. For each s objects in s_i $(i < j)$, we scan them one by one, and use F_θ for global pruning. For those objects with scores higher than F_θ, we push them into a temporary buffer $\mathcal{B}$. Before the objects in s_i are accessed, if $|\mathcal{B}|$ reaches $2(k - P_1.\rho)$, we invoke the *median searching algorithm* to find the median, remove the ones with scores lower than the median, and update F_θ to the median. When all the objects in s_i are accessed, we insert the $\min(k - P_1.\rho, |\mathcal{B}|)$ objects with highest scores in $\mathcal{B}$ into the S-AVL of P_0.

Similar as the analysis presented in Section 4.3, we *at most* use $O(s)$ computational cost to find the $k - P_1.\rho$ objects with highest scores in each s_i. Together with the cost of inserting meaningful objects into M_1, the process cost of each s objects is $O(s \times cost_F + (k - P_0.\rho)\log(k - P_0.\rho))$. Therefore, the cost of constructing M_1 is bounded by $O(\frac{n}{m} \times cost_F + \frac{n(k-P_0.\rho)}{ms}\log(k - P_0.\rho))$.

TABLE 4
Parameter Settings

Parameter	value
n	10%, 20%, 30%, 40%, **50%** ($\times|D|$)
k	500, **1000**, 5000, 10000,50000
s	0.01%, 0.1%, 1%, **2%**, 5%, 10% ($\times n$)

TABLE 5
Running time comparison of different algorithms under high-speed streams (unit: second)

Data set		**parameter** $n(\times\|D\|)$					**parameter** k					**parameter** $s(\times n)$					
		10%	20%	30%	40%	50%	500	1000	5000	10000	50000	0.01%	0.1%	1%	2%	5%	10%
STOCK	minTopK	9.4	9.5	9.58	9.7	9.8	7.6	9.8	12.5	15.4	37.1	16.7	13.5	12.5	9.8	8.3	7.1
	SAP	6.2	6.4	6.5	6.7	6.9	6.2	6.9	7.7	9.1	14.2	12.6	8.9	7.1	6.9	6.8	6.7
TRIP	minTopK	8.9	9.2	9.6	9.7	10.4	8.7	10.4	12.5	17.4	35.1	14.7	11.9	11.5	10.4	8.6	7.5
	SAP	5.9	6.1	6.4	6.8	7.1	6.2	7.1	8.9	10.1	15.6	11.5	9.8	7.9	7.1	6.9	6.6
PLANET	minTopK	1.2	1.24	1.35	1.47	1.6	1.3	1.6	3.7	5.4	11.7	3.1	2.2	1.85	1.6	1.4	1.2
	SAP	0.9	0.97	1.03	1.1	1.17	1.1	1.17	1.3	1.8	3.23	1.74	1.4	1.29	1.17	1.1	1.02
TIME^U	minTopK	9.6	9.8	10.0	10.3	10.7	10.2	10.7	14.2	19.8	36.9	17.6	13.2	11.3	10.7	8.5	7.7
	SAP	7.2	7.4	7.6	7.7	7.9	6.9	7.9	11.3	12.6	14.7	13.4	9.9	8.1	7.9	7.5	7.4
TIME^R	minTopK	24.1	25.1	26.3	26.4	26.8	22.9	26.8	41.2	55.6	64.1	40.9	37.7	31.1	26.8	21.7	15.1
	SAP	15.9	16.1	16.2	16.3	16.6	14.7	16.6	19.7	22.1	25.9	24.3	18.9	17.2	16.6	15.6	14.8

TABLE 6
Candidates amount comparison of different algorithms

Data set		**parameter** $n(\times\|D\|)$					**parameter** k					**parameter** $s(\times n)$					
		0.01%	0.05%	0.1%	0.5%	1%	10	50	100	500	1000	0.01%	0.05%	0.1%	1%	5%	10%
STOCK	SAP	461	488	495	531	551	57	249	495	2683	4895	603	533	495	436	391	369
	minTopK	1123	1319	1428	1493	1564	132	643	1428	4323	21500	2013	1890	1428	1069	621	422
	k-skyband	1628	1922	2359	2677	2953	266	1886	2359	9020	30763	2391	2386	2382	2372	2371	2368
TRIP	SAP	387	401	453	466	498	82	226	453	2151	4888	510	486	453	383	367	345
	minTopK	453	507	673	799	822	128	432	673	3500	9752	866	737	673	582	419	369
	k-skyband	677	811	931	1165	1285	191	634	931	6538	12832	953	948	931	932	925	922
PLANET	SAP	361	388	455	467	488	67	211	455	1409	5261	496	460	455	415	379	336
	minTopK	438	488	636	774	811	177	269	636	1799	8878	1142	976	636	552	455	346
	k-skyband	1346	1493	1522	1724	1811	230	479	1522	2338	11541	1532	1527	1522	1507	1501	1494
TIME^U	SAP	228	241	286	337	360	28	130	286	1644	2782	303	298	286	269	263	257
	minTopK	639	713	784	853	1015	128	458	784	3204	5462	905	820	784	469	421	383
	k-skyband	710	873	940	1043	1149	188	647	940	3958	6364	942	941	940	935	932	931
TIME^R	SAP	986	1355	1902	650	598	320	670	1902	7500	15620	4650	2628	1902	920	350	260
	minTopK	1239	3169	5104	10352	10352	562	2010	5104	12212	48360	5364	3309	3017	1453	401	277
	k-skyband	4618	10158	17258	23558	23598	1383	2775	17258	27019	98403	17258	17258	17258	17258	17258	17258

TABLE 7
Candidates amount comparison of different algorithms under high-speed streams(unit: KB)

Data set		**parameter** $n(\times\|D\|)$					**parameter** k					**parameter** $s(\times n)$					
		10%	20%	30%	40%	50%	500	1000	5000	10000	50000	0.01%	0.1%	1%	2%	5%	10%
STOCK	minTopK	4.0	4.7	5.5	6.5	6.8	3.0	6.8	38.4	73.1	355.0	8.4	7.6	7.0	6.8	4.3	3.8
	SAP	3.4	3.5	3.7	3.8	4.0	2.7	4.0	13.3	26.9	187.0	4.7	4.5	4.2	4.0	3.6	3.3
TRIP	minTopK	4.2	4.9	5.5	6.4	6.7	3.1	6.8	29.7	59.2	327.6	9.7	8.4	7.7	6.8	4.2	3.5
	SAP	3.5	3.6	3.7	3.8	3.9	2.0	3.9	12.3	38.4	157.7	4.9	4.7	4.6	3.9	3.5	3.1
PLANET	minTopK	3.7	4.7	5.3	6.1	6.6	2.6	6.6	32.4	61.2	435.0	10.2	9.0	7.4	6.6	3.9	3.0
	SAP	2.9	3.1	3.2	3.3	3.6	1.4	3.6	14.2	42.3	209.1	4.2	4.0	3.8	3.6	3.0	2.4
TIME^U	minTopK	2.8	3.4	4.1	4.7	5.4	2.6	5.4	26.5	58.0	339.5	6.6	5.9	5.6	5.4	3.9	3.1
	SAP	2.4	2.9	3.2	3.4	3.8	1.8	3.8	17.1	34.0	187.5	5.4	4.7	4.2	3.8	3.3	2.9
TIME^R	minTopK	3.4	4.1	4.9	5.7	5.9	2.1	5.9	23.2	33.6	183.8	9.3	7.1	6.7	5.9	4.4	3.9
	SAP	3.2	3.6	3.7	4.1	4.3	1.6	4.3	19.2	27.5	122.2	5.2	4.7	4.6	4.3	4.1	3.8

APPENDIX D

In this Section, we supplement a set of experiments to compare SAP with its competitors under high-speed stream environment. The parameters are shown in Table 4. Note that, under this setting, both the window size and the slide size are much larger than that of parameters in section 6.1. For example, under the dataset STOCK, by default the window is set to 500MB and the slide is set to 10MB respectively. As reported in Section 6.3, minTopK performs better than both SMA and k-skyband when s is large, and hence we only compare SAP with minTopk in this new set of experiments.

First, we report the running time in Table 5. We observe from the results that the running time difference between SAP and minTopk is smaller than that of the results reported in Section 6.3. That is because the larger the s, the stronger the pruning ability it has, and the fewer the candidates it has to maintain.

Although the performance gap between SAP and minTopk becomes smaller, SAP still performs better. On average, SAP only consumes around 70% of minTopK's running time. The reason behind is that when the window slides, minTopk selects the k objects with highest scores in the last *slide* and SAP selects the k objects with highest scores in the last *partition* as candidates respectively. Compared with minTopk, SAP selects fewer objects as candidates and hence SAP takes fewer insertion/deletion operations to process newly arrival objects.

We also find that when $\frac{n}{s}$ is small, the running time of minTopK is roughly the same as that of SAP. The reason is that when $\frac{n}{s}$ is small, $\frac{n}{s}$ approaches to the partition amount, which leaves very limited space for SAP to demonstrate its advantage.

APPENDIX E

In this section, we report the size of the candidate set maintained by each algorithm in Table 6. Note that SMA indexes all the objects in the window via grid file, so we do not report its average candidate amount. We find that k-skyband has to maintain the largest number of candidates, averagely 1.5 times of that maintained by MinTopK, and 3 times of the number maintained by SAP. This is because MinTopK maintains all the k-skybands; and k-skyband maintains a *superset* of k-skybands since it does not consider the dominance relationships among objects in the same slide. By contrast, SAP only maintains a small set of "high quality" candidates.

Another observation is that both MinTopK and k-skyband are sensitive to the data distribution. For example, under the TIME^R, they have to maintain a very large candidate set because

TABLE 8
Memory consumption comparison of different algorithms (unit: KB)

Data set		parameter $n(\times\|D\|)$					parameter k					parameter $s(\times n)$					
		0.01%	0.05%	0.1%	0.5%	1%	10	50	100	500	1000	0.01%	0.05%	0.1%	1%	5%	10%
STOCK	SAP	15.6	16.5	16.8	18.5	18.7	1.9	8.4	16.8	91.2	166.4	20.5	18.1	16.8	14.8	13.2	12.5
	minTopK	31.0	36.0	39.0	41.0	43.0	3.6	18.0	39.0	161.0	602.0	56.3	52.9	39.0	29.8	17.0	13.2
	k-skyband	42.0	49.0	62.0	69.0	76.0	6.8	49.0	62.0	334.0	799.0	63.1	62.8	62.0	61.5	61.1	60.4
TRIP	SAP	10.8	11.2	12.6	13.0	13.8	2.4	6.4	12.6	60.2	136.8	14.2	13.6	12.6	10.8	10.2	9.6
	minTopK	14.6	16.2	19.8	23.8	25.0	3.6	12.2	19.8	98.0	273.0	22.7	20.6	19.8	14.2	12.7	10.0
	k-skyband	16.2	19.4	22.4	27.6	30.8	4.6	15.2	22.0	156.0	307.0	23.5	22.9	22.4	22.3	22.2	21.6
PLANET	SAP	12.8	13.2	15.4	15.8	16.8	2.4	7.2	15.4	47.8	178	16.8	15.6	15.4	14	12.8	11.2
	minTopK	19.8	21.2	21.9	24.8	26.5	4.6	13.9	21.9	48.2	239.0	30.8	26.6	21.9	14.8	13.2	11.4
	k-skyband	32.4	35.8	36.8	41.2	43.4	5.2	11.6	36.8	56.2	276.0	37.1	36.9	36.8	36.0	35.4	35.2
TIME^U	SAP	6.6	6.8	7.2	9.0	9.2	0.9	3.4	7.2	44.8	75.4	8.8	8.0	7.2	7.6	7.0	6.6
	minTopK	18.4	20.6	22.6	24.2	26.3	2.1	13.2	22.6	92.6	158.8	26.4	23.8	22.6	13.0	12.2	11.0
	k-skyband	19.4	21.9	24.4	26.2	29.4	3.7	5.2	24.4	112	210.2	26.9	26.6	26.4	26.2	25.9	25.6
TIME^R	SAP	28.4	39.2	55.8	18.8	17.2	9.8	19.4	55.8	217.0	452.0	134.0	76.0	55.8	26.0	10.0	7.6
	minTopK	35.0	91.2	148.0	300.2	300.2	16.2	58.0	148.0	354.0	1402.0	205.0	175.0	148.0	83.0	41.0	12.9
	k-skyband	120.0	264.0	448.0	612.0	613.0	35.0	72.0	448.0	702.0	2558.0	448.0	448.0	448.0	448.0	448.0	448.0

TABLE 9
Memory consumption comparison of different algorithms under high-speed streams(unit: KB)

Data set		parameter $n(\times\|D\|)$					parameter k					parameter $s(\times n)$					
		10%	20%	30%	40%	50%	500	1000	5000	10000	50000	0.01%	0.1%	1%	2%	5%	10%
STOCK	SAP	115	124	126	129	136	93	136	453	917	6358	162	153	142	136	122	108
	minTopK	136	159	187	221	231	102	231	1305	2485	12070	285	258	238	231	146	113
TRIP	SAP	115	127	128	130	132	66	132	418	1302	5365	166	159	156	132	115	104
	minTopK	144	167	182	215	233	102	233	1009	2016	11131	332	287	265	233	143	111
PLANET	SAP	96	104	109	116	122	45	122	486	1437	7109	142	136	129	122	102	88
	minTopK	122	158	180	207	224	83	224	1107	2060	14750	346	306	251	224	132	102
TIME^U	SAP	81	96	105	117	132	64	129	581	1156	6375	187	157	144	129	112	97
	minTopK	95	115	139	159	173	88	173	921	1952	11843	224	200	190	173	129	105
TIME^R	SAP	108	122	125	139	146	54	146	652	935	4154	176	159	156	146	134	129
	minTopK	115	136	162	193	203	71	203	788	1142	6249	316	241	227	203	149	132

each newly arrived object located at the monotone decreasing interval will be maintained for a long time until the inflection point appears. In addition, the factor s could be effectively used to reduce the number of candidates we need to maintain. For example, when $s = 10\% \times n$, the size of the candidate set under MinTopK is roughly the same as that of SAP. The reason is that the candidates amount under MinTopK is bounded by $\frac{nk}{\max(s,k)}$. When $s = 10\% \times n$, the candidates amount under MinTopK is no less than $10k$, which leaves very limited space for SAP to demonstrate its advantage.

Under high-speed stream environment, SAP still uses fewer candidates for supporting the top-k search, as reported in Table 7. Similar to the reason discussed before, SAP allows us to use a subset of meaningful objects for supporting the query. By contrast, minTopK maintains all the k-skybands and k-skyband maintains a *superset* of k-skybands.

APPENDIX F

In the following, we supplement a set of experiments, where *memory consumption* is employed as the performance metrics. We compare SAP with its competitors under the regular-speed streaming environment and the high-speed streaming environment respectively. A workload for top-k query consists of 100 random queries, and their average performance is reported.

We now report the *memory consumption* of each algorithm under regular-speed streaming. From Table 8, we find that SAP spends the smallest memory space. That is because SAP only maintains a small set of "high quality" candidates. Therefore, it occupies the smallest memory space. By contrast, minTopK maintains all the k-skybands and it requires additional memory space to maintain the *lbp* pointer for each predicted window. Therefore, SAP incurs a smaller memory space cost than that of minTopK. k-skyband maintains a *superset* of k-skybands, and hence it requires the largest memory space.

In the high-speed stream environment, minTopK performs better than k-skyband and hence we only compare SAP with minTopK. Similar to the results presented above, SAP still performs the best as reported in Table 9. Another observation is that minTopK is more sensitive to the query parameters. For example, when $s = 0.1\% \times n$, the candidate set size of minTopK is much larger than that of SAP. This is because when $s = 0.1\% \times n$, minTopK has to maintain meaningful objects of 1000 predicted windows. By contrast, SAP only maintains a few "high quality" candidates in the window. However, when $s = 10\% \times n$, the memory consumed by minTopK is roughly the same as that of SAP. This is because the number of candidates maintained by minTopK when $s = 10\% \times n$ is bounded by $10k$, which leaves very limited space for SAP to demonstrate its advantage.