Zhu R, Wang B, Luo SY *et al.* Approximate continuous top-k query over sliding window. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 32(1): 93–109 Jan. 2017. DOI 10.1007/s11390-017-1708-0

Approximate Continuous Top-k Query over Sliding Window

Rui Zhu, *Member, CCF, ACM*, Bin Wang*, *Member, CCF*, Shi-Ying Luo, *Member, CCF, ACM*
Xiao-Chun Yang, *Senior Member, CCF, IEEE, Member, ACM*, and Guo-Ren Wang, *Member, CCF, ACM, IEEE*

College of Computer Science and Engineering, Northeastern University, Shenyang 110819, China

E-mail: neuruizhu@gmail.com; binwang@mail.neu.edu.cn; neulsy@hotmail.com
{yangxc, wanggr}@mail.neu.edu.cn

Received February 29, 2016; revised August 17, 2016.

Abstract Continuous top-k query over sliding window is a fundamental problem in database, which retrieves k objects with the highest scores when the window slides. Existing studies mainly adopt exact algorithms to tackle this type of queries, whose key idea is to maintain a subset of objects in the window, and try to retrieve answers from it. However, all the existing algorithms are sensitive to query parameters and data distribution. In addition, they suffer from expensive overhead for incremental maintenance, and thus cannot satisfy real-time requirement. In this paper, we define a novel query named (ε, δ)-approximate continuous top-k query, which returns approximate answers for top-k query. In order to efficiently support this query, we propose an efficient framework, named PABF (Probabilistic Approximate Based Framework), to support approximate top-k query over sliding window. We firstly maintain a self-adaptive pruning value, which could filter out newly arrived objects who have a probability less than $1 - \delta$ of being a query result. For those objects that are not filtered, we combine them together, if the score difference among them is less than a threshold. To efficiently maintain these combined results, the framework PABF also proposes a multi-phase merging algorithm. Theoretical analysis indicates that even in the worst case, we require only logarithmic complexity for maintaining each candidate.

Keywords continuous top-k query, approximate, sliding window

1 Introduction

Continuous top-k query over sliding window[1] is a classic problem in the data stream environment[2]. It has various applications[3], such as sensor data analysis[4], economic decision making, wireless sensor, and market surveillance. Let W be the query window, and F be a preference function. When the window slides, a continuous top-k query returns k objects with the highest scores in the current window[5].

Many exact algorithms have been studied to tackle this problem[1]. When the window slides, their main idea is to maintain a relatively small set of objects for each current window such that the top-k results can be retrieved from this set as many as possible. Nevertheless, the cost of maintaining such set of objects is significantly high, which has a complexity linear to the window size in the worst case. In real applications, this complexity usually fails to satisfy a real-time requirement. Therefore an approximate algorithm that could quickly answer the query with bounded deviation is desired.

For example, a stock recommendation system may retrieve 10 most significant transactions within the last one hour. In this scenario, the system is expected to process 1 MB objects per second, while the state-of-the-art algorithms could only process roughly 100 KB objects per second when answering top-k query[1]. Therefore, existing algorithms are usually unable to return the exact results in real time, leading to frequent system congestion. As a consequence, the system usually provides overdue results to users. Given the importance

Regular Paper
This work is partially supported by the National Natural Science Fund for Distinguish Young Scholars of China under Grant No. 61322208, the National Basic Research 973 Program of China under Grant No. 2012CB316201, the National Natural Science Foundation of China under Grant Nos. 61272178 and 61572122, and the Key Program of the National Natural Science Foundation of China under Grant No. 61532021.

*Corresponding Author

of results' timeliness in stock recommendation systems, it is more reasonable to provide users with approximate results so as to speed up query processing. The rationality here is that if the deviation between exact and approximate results can be bounded by a user-specified threshold, approximate results are also acceptable.

Another example is in network traffic analysis, where analysts can discover potential victims of an on-going distributed denial of service (DDoS) attack by monitoring nodes with the top-k largest throughput. Since the analysis system may receive 1 MB data packets per second, making it unrealistic to process streaming data in real time under state-of-the-art algorithms, users cannot be protected. In this case, it is more rational to provide approximate top-k results with relatively high throughput to the system.

In a nutshell, in high-speed stream environment, since the system usually requires processing huge amount of objects, an approximate algorithm which provides deviation-bounded results with high efficiency is desired.

In this paper, we propose an efficient framework, named PABF (Probabilistic Approximate Based Framework), to support approximate top-k query over sliding window. PABF consists of two steps. In the first step, when the window slides, it scans the newly arrived objects in batch, and utilizes a pruning value to filter out objects whose scores are lower than the pruning value. After pruning, the second step adopts the idea of merge sort to merge the remaining objects to the candidate set.

Challenges. For the first step, it is difficult to determine a suitable pruning value for the first step, since the distribution of streaming data changes with the time in high-speed stream environment. For the second step, when data distribution is highly skewed, the candidate set may be large. As a result, the cost of maintaining candidate set would be large. Therefore, how to design a general scheme that can efficiently maintain candidates under different data distributions is also a challenge.

To deal with the challenges above, the contributions of this paper are as follows.

Contributions. 1) We define a novel query called (ε, δ)-approximate continuous top-k query over sliding window. Both ε and δ are thresholds specified by users, where δ is a probabilistic threshold. As for ε, it is a threshold used for bounding the score difference between exact and approximate results (see Subsection 2.2). 2) We propose several pruning algorithms to filter newly arrived objects. By learning the feature of stream data distribution, these algorithms generate a corresponding pruning value to filter objects. More importantly, the pruning value is self-adaptive, adjusted according to the variation of data distribution. Suppose s objects flow into the window when the window slides, according to theoretical analysis, this filtering mechanism requires only linear time to prune $O(s-k)$ objects directly in most cases (see Section 4). 3) We propose a stable and efficient algorithm for maintaining candidates by converting candidate maintenance to the problem of merge sort. In order to reduce the merge cost, we firstly propose a novel scheme to combine candidates whose scores are roughly the same, and generate summary information for these combined candidates. In addition, we also propose a multi-phase merge approach. Given a sliding window with size N, we prove that after processing N objects, the total merge cost is bounded by $O(\frac{Nk}{s}\log\frac{R}{\varepsilon k})$. R refers to the range of stream data's score here (see Section 5).

The rest of this paper is organized as follows. Section 2 reviews related work and presents the definition of $(\varepsilon,\ \delta)$-approximate continuous top-k query. Section 3 gives an overview of the framework. Section 4 and Section 5 discuss the filter and the merge algorithm respectively. We demonstrate how PABF supports the query in Section 6; and finally in Section 7, we report our experimental result. Section 8 concludes this paper.

2 Preliminary

2.1 Related Work

Top-k query is an important query in the domain of text processing, bio-informatics[6-9], streaming data management, and so on. In this subsection, we focus on the problem of continuous top-k query over data stream. The state-of-the-art work can be categorized into two classes, which are multi-pass and one-pass approaches. Some classic multi-pass approaches maintain top-k' objects in the query window W as candidates with $k \leqslant k' \leqslant 2k$[10]. The main limitation of these approaches is: if many candidates leave the window but few ones are supplied into the candidate set (i.e., $k \geqslant k'$), they have to re-construct the candidate set by re-scanning the window. To reduce the overhead for re-scanning, SMA[2] utilizes a grid structure to index streaming data. Each time when re-scanning is required, SMA[2] could construct the candidate set by accessing a few cells in grid file. However, SMA is sensitive to the distribution of streaming data. If object scores are crowded in only a small number of cells, the overhead of re-scanning would still be high.

In order to overcome the re-scanning problem, one classic one-pass based approach is proposed, which maintains all the k-skybands in the window[5]. Here, an object o is called as a k-skyband if there exist less than k objects which come earlier than o and have scores higher than $F(o)$. Shen *et al.*[5] proved that a top-k query can be answered from the k-skybands in the window. However, all the objects are maintained as k-skybands when they flow into the window. Therefore, it has no ability to prune newly arrived objects even if they can never be query results. This issue leads to an incremental cost of this algorithm linear to the window size. Pripužić *et al.*[11] adopted the probabilistic k-skyband to answer the top-k query. However, their proposed approach only works when processing random-order data streams. In addition, they cannot provide a theoretical bound for the scores of the query results.

Algorithm minTopK improves k-skyband based algorithm via considering an important parameter s under the sliding window model. Given a query window W (N, s), it can be either count- or time-based. For ease of presentation, we only consider the count-based window, where N denotes the total count of objects contained in the window, and s denotes the number of objects that arrive when the window slides. Since s objects flow into (or leave) the window at the same time when the window slides, it is enough to support the query via maintaining top-k objects in every s objects as candidates, and the other $s - k$ objects could be safely pruned. Meanwhile, MinTopK introduces the concept of predicted result set to further prune meaningless candidates. As shown in [1], the incremental maintenance cost of MinTopK is $O(\log \frac{Nk}{s} + \frac{N}{s})$. Intuitively, the performance of MinTopK is better than that of the other algorithms. However, when s is small, it has the similar problem to that of k-skyband-based algorithm.

In the streaming data management field, a large number of approximate algorithms have been well studied, including quantile query, heavy hitter[12-17], and so on. For quantile query, there are numerous algorithms proposed in this setting. For example, Shrivastava *et al.*[18] designed a deterministic, fixed universe algorithm, named q-digest. q-digest consumes $O(\frac{1}{\varepsilon} \log u)$ space for quantile computation. Ganguly and Mayumder[17] proposed an algorithm using $O(\frac{1}{\varepsilon^2} \log^5 u \log(\frac{\log u}{\varepsilon}))$ space for quantile computation. Later, Cormode and Muthukrishnan[19] applied Count-Min sketch in the dyadic structure, which can reduce the overall space cost to $O(\frac{1}{\varepsilon} \log^2 u \log(\frac{\log u}{\varepsilon}))$.

Heavy hitters problem also has been well studied, which is used for catching the most frequent items in stream. Alon *et al.*[12] gave an $\Omega(n)$ lower bound when estimating the frequency of the largest item. Charikar *et al.*[16] proposed an one-pass algorithm for estimating the most frequent items in streaming data using very limited space.

2.2 Problem Definition

Given a set of objects D and a user-specified preference function F, a top-k query retrieves k objects with the highest scores in D. In the sliding window scenario, a continuous top-k query $q(N, s, k, F)$ returns k objects with the highest scores in the window when the window slides. Without loss of generality, the query window $W(N, s)$ can be count- or time-based. For ease of presentation, in the rest of the paper, we only consider the count-based windows. Under this setting, parameter N denotes the number of objects in the window, and s denotes the number of objects that arrive when the window slides.

For example, let W_0 be the current window. Based on the arrival order of each object in W, the objects in window W could be partitioned into $\{s_0, s_1, ..., s_{n-1}\}$ (i.e., $n = \frac{N}{s}$). Here, s_i refers to object set with arrival order i. When W slides to $W_1\{s_1, s_2, ..., s_n\}$, the objects in s_n flow into the window, and top-k results should be updated accordingly. Next, we will formally introduce the (ε, δ)-approximate continuous top-k query over sliding window. For simplicity, we use $\{o_1, o_2, ..., o_k\}$ to represent the exact results, and $\{r_1, r_2, ..., r_k\}$ to represent the approximate results. $F(o_i)$ refers to the i-th highest score in the exact results, and $F(r_i)$ indicates the i-th highest score in the approximate results.

Definition 1 ((ε, δ)-Approximate Continuous Top-k Query). *Given a (ε, δ)-approximate continuous top-k query q, it returns k objects with relatively high score in the window, such that $\forall i$, $F(o_i)$ and $F(r_i)$ satisfy the inequality* $\Pr(|F(r_i) - F(o_i)| \leqslant \varepsilon) \geqslant \delta$.

According to Definition 1, the difference of the i-th highest score between the exact and approximate results set should be smaller than a threshold ε with the probability δ. For example, let W_0 be the current window, $\{96, 86, 50, 98, 63, 94, 97, 79, 13, 88, 93, 85\}$ be the objects contained in W_0, $k = 2$, $s = 3$, $\varepsilon = 2$ and $\delta = 0.99$. Our algorithm returns $\{98, 96\}$ as the results (see Section 6). Since the exact results here are

$\{98, 97\}$, $98-98 \leqslant 2$, and $97-96 \leqslant 2$, we call these results as acceptable approximate results. When the window slides to W_3, i.e., the objects are updated to {88, 93, 85, 77, 60, 82, 73, 70, 48, 60, 71, 66}, our algorithm outputs $\{93, 88\}$. They are also acceptable approximate results. When the window slides to W_4, the objects are updated to {77, 60, 82, 73, 70, 48, 60, 71, 66, 65.5, 54, 70}, and our algorithm outputs $\{73, 71\}$. Since the exact results are $\{82, 77\}$, in which $82-73 \geqslant 2$, we regard 73 as an unacceptable approximate result.

Assuming the window slides from W_0 to W_{n-1}, our proposed TAHM algorithm outputs $2 \times n$ results during this process. For each approximate result o_{ij}, given $\Pr(|F(r_{ij}) - F(o_{ij})| \leqslant 2) \geqslant 0.99$, the expected number of unacceptable approximate results among these results is $(1-\delta) \times n \times k = 0.02n$. Here, $F(r_{ij})$ is the j-th highest score in the approximate result set of W_i, and $F(o_{ij})$ is the j-th highest score in W_i.

3 Framework Overview

As has been discussed, the main challenges of answering approximate top-k query are: 1) for each newly arrived object o, the algorithm should efficiently prune it if o is impossible to become a query result; 2) if o is not pruned, it should be inserted into candidate set with low computation cost. In this paper, we tackle the challenges by designing an efficient framework, named PABF (Probabilistic Approximation Based Framework) for supporting approximate continuous top-k query over sliding window.

As shown in Algorithm 1, PABF mainly consists of the following four modules: Filter, Local-Merge, Global-Merge, and TA-Heap. Filter here is used for pruning newly arrived objects (lines 3~7). To be more specific, for each object $o \in s_n$, we determine whether it is a query result (or may become a result for a future query window with a probability of at least $1-\delta$). If so, o is selected as a candidate, and inserted into a temporary buffer $\mathcal{M}$; otherwise, o is ignored. Compared with other one-pass algorithms, one advantage of our algorithm is that it can directly prune $O(s-k)$ objects in s_n with the help of a suitable pruning value.

After scanning s_n, we merge candidates in $\mathcal{M}$ with candidate set $\mathcal{B}$. To speed up candidate maintenance, we partition $\mathcal{B}$ into a group of buckets. Formally, given the candidate set $\mathcal{B}$, a partition $\mathcal{P}(\mathcal{B}, m) = \{b_1, b_2, \ldots, b_m\}$ is to partition the elements in $\mathcal{B}$ into m buckets $\{b_1, b_2, \ldots, b_m\}$ such that 1) $0 < |b_i| < (\varphi)^{m-i+1}k$; 2) $\forall o \in b_i, o' \in b_{i-1}$, in which $T(o)$ should be larger than $T(o')$; 3) objects in each bucket are sorted. $T(o)$ here refers to the arrival order of o, and φ is a coefficient whose optimal value will be studied in Section 5.

Algorithm 1. PABF Framework

```
Input: query window $W$, current result set $\mathcal{R}$ and
       candidate set $\mathcal{B}$
Output: updated candidate set $\mathcal{B}$, and updated
        result set $\mathcal{R}$
1  Window maintenance: $W \leftarrow W - s_0$, $W \leftarrow W \cup s_n$;
2  for $i$ from 0 to $s-1$ do
3      Bool $bPruned \leftarrow$ Filter ($s[i]$);
4      if $bPruned \neq$ false then
5          $\mathcal{M} \leftarrow \mathcal{M} \cup s[i]$;
6      else
7          Ignore $s[i]$;
8  for $i$ from 0 to $|\mathcal{M}|-1$ do
9      Local-Merge ($s[i]$, $b_m$);
10 if $|b_m| \geqslant \varphi k$ then
11     Int $i \leftarrow m-1$, Bool $bMerge \leftarrow$ true;
12     while $bMerge$ do
13         Golbal-Merge ($b_i$, $b_{i+1}$);
14         if $|b_i| > \varphi^{m-i+1}k$ then
15             $bMerge \leftarrow$ false;
16 $\mathcal{R} \leftarrow$ TA-Heap ($\mathcal{B}$, $\mathcal{M}$);
17 Return;
```

Based on the partition result, the merge operation can be divided into two phases: Local-Merge and Global-Merge. In the Local-Merge phase, we sort candidates according to their scores via merge sort. We then try to combine candidates in $\mathcal{M}$ with the ones in b_m together, if their scores are roughly the same. Besides, we also maintain the dominate number for each element in b_m, and delete meaningless ones. Through the above operations, the size of b_m can be effectively reduced. Since the overhead of merge cost mainly depends on the size of b_m, a smaller size of b_m helps us further reduce the cost of merge. After Local-Merge, if $|b_m| > \varphi k$, our algorithm enters the Global-Merge phase that further merges b_m with b_{m-1} (lines 10~15). The following operations are repeated until the condition $|b_i| < \varphi^{m-i+1}k$ is satisfied. Note that in this phase, we could further reduce the cost of candidates maintenance by computing the optimal partition.

Now we demonstrate how PABF supports the query. We maintain k' $(k < k' < 2k)$ objects with the highest scores in set $\mathcal{R}$. When a query result leaves the window, we retrieve the new results from $\mathcal{R}$ if $k < k'$. Otherwise, we directly retrieve new answers from $\mathcal{B}$, i.e., $\{b_1, b_2, \cdots b_m\}$. Since objects in each bucket are sorted, we can re-construct $\mathcal{R}$ using heap-merge.

As shown in Fig.1, we partition the candidate set $\mathcal{B}$ into four parts, which are b_1, b_2, b_3 and b_4. Newly arrived objects o are inserted into $\mathcal{M}$ if they cannot be pruned. After processing all the objects in s_n, we combine elements in $\mathcal{M}$ with b_4 via Local-Merge. If b_4 becomes full after Local-Merge, we then continuously merge b_4 into b_3 via Global-Merge. In order to support top-k query, if $k < k'$, we select $2k - k'$ objects from b_1, b_2, b_3 and b_4. The detail is elaborated in Section 6.

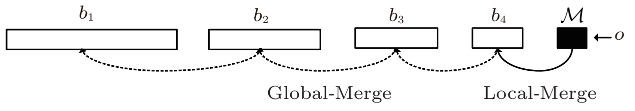


Fig.1. PABF framework.

We want to highlight that compared with minTopK, PABF has the following two advantages. Firstly, we could use a powerful filter to prune most of the streaming data directly. Secondly, since we combine the candidates together if their score difference is small, we could bound $|\mathcal{B}|$ by $\frac{R}{\varepsilon}$. Therefore, the total computation cost of processing N stream data is $O(N(\frac{k}{s}\log\frac{R}{\varepsilon k} + cost_F + \log k))$. Here, R is the range of objects' score, and $cost_F$ refers to the cost of deriving the score for each object. The detailed theoretical analysis will be shown in Section 6.

4 Filter Algorithms

In this section, we discuss how to filter newly arrived objects. We firstly introduce a baseline algorithm, and then propose a group of self-adaptive algorithms to enhance the filter efficiency.

4.1 Baseline Algorithm

As discussed above, since s objects flow into the window at the same time when the window slides, only the k objects with the highest scores in s_n have the chance to become the query result. Therefore, our basic idea is to scan the objects in s_n one by one, and maintain the k objects with the highest scores. Here, s_n refers to a set of objects that have just flowed into the window.

Specifically, for each object $o_i \in s_n$, we firstly check whether o_i will contribute to $\mathcal{M}$ according to the pruning value $\min(\mathcal{M})$ and the size of $\mathcal{M}$, i.e., $|\mathcal{M}|$. If $F(o_i) > F(\min(\mathcal{M}))$ or $|\mathcal{M}| < k$, we insert o_i into $\mathcal{M}$; otherwise, we ignore o_i. Here, $\mathcal{M}$ is a min-heap with maximum capacity k, and $\min(\mathcal{M})$ refers to the object with the lowest score in $\mathcal{M}$. After insertion, if $|\mathcal{M}| > k$, we remove $\min(\mathcal{M})$ from $\mathcal{M}$. After scanning s_n, we output $\mathcal{M}$.

We want to highlight that the overhead of processing a newly arrived object is $O(\log k + cost_F)$ using Baseline algorithm. As discussed in Subsection 2.1, when objects' scores keep increasing (e.g., $s_n = \{1, 2, 3, ..., 10\}$), each object in s_n will be maintained as a candidate when it is accessed, which incurs expensive computational overhead. An ideal solution to tackle this issue is to identify false candidates early. Here, false candidates refer to those objects en-heaped to $\mathcal{M}$ and then de-heaped from $\mathcal{M}$ when the objects with higher scores are accessed. In the following, we will propose a group of algorithms, which can maintain a self-adaptive pruning value for filtering false candidates.

4.2 Self-Adaptively Filtering Algorithms

In this subsection, we firstly explain how to maintain this pruning value when we have priori knowledge on the data distribution. We then introduce an algorithm to filter newly arrived objects even when the data distribution is not known beforehand.

4.2.1 Algorithm Under Time-Unrelated Data Distribution

We now present the first algorithm. Intuitively, if objects' scores are not correlated with their arrival orders, the distribution of the streaming data in each s_i may be roughly the same. Therefore, we select the pruning value, denoted as F_θ, from the objects in s_{n-1}. Following Theorem 1, we set F_θ to $F(o^\zeta_{n-1})$. For each object $o \in s_n$, if $F(o) \leqslant F_\theta$, it could not become the query result with probability δ. Here, o^ζ_{n-1} is the object with the ζ-th highest score in s_{n-1}.

Theorem 1. *Given s_{n-1} and s_n, let $F(o^\zeta_{n-1})$ be the ζ-th highest score in s_{n-1}, and ζ be the solution of $\Phi(\frac{\zeta-k}{\sqrt{\zeta}}) = \delta$. If the objects in both s_{n-1} and s_n follow the same distribution, and $F(o) < F(o^\zeta_{n-1})$, the probability for o to be a top-k object in s_n is at most δ.*

Proof. Given an object $o_i \in s_n$, if s_{n-1} and s_n are big enough, we could use normal distribution $N(np, \sqrt{np(1-p)})$ to approximate $\Pr(|X| \geqslant k)$ according to the Demovire-Laplace theorem, with $n = s$ and $p = \frac{\zeta}{s}$ to approximate $\Pr(|X| \geqslant k)$, i.e., $\Pr(|X| \geqslant k) = 1 - \Pr(|X| \leqslant k) \approx \Phi(\frac{\zeta-k}{\sqrt{\zeta}})$. Therefore, we set ζ to the solution of the equation $\Phi(\frac{\zeta-k}{\sqrt{\zeta}}) = \delta$. Because $\Pr(|X| \geqslant k) \geqslant \delta$, we deduce that if $F(o_i) \leqslant F(o^\zeta_{n-1})$, the probability for o to be a top-k object in s_n is at most $1-\delta$. $\square$

We now explain how to integrate this algorithm into our Baseline algorithm seamlessly. We employ a parameter $flag$ to indicate whether F_θ is valid or not, with the flow depicted in Fig.2. If $flag = 1$ which indicates F_θ is valid, we discard all the objects with scores smaller than F_θ; otherwise, F_θ is invalid, and we follow the Baseline algorithm to manipulate the objects in s_n, i.e., using $F(\min(\mathcal{M}))$ as the the pruning value. Here, we evaluate whether F_θ is valid according to the size of $\mathcal{M}$. When $|\mathcal{M}|$ reaches k, we set $flag$ to 0. After scanning s_n, we output $\mathcal{M}$, and reset $flag$ to 1.

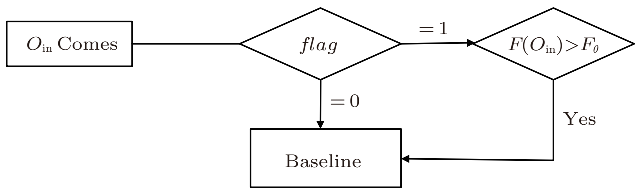


Fig.2. Pruning value-based candidate selection flow.

Note that when we finish processing s_n, $\Pr(|X| \geqslant k)$ is almost δ according to Theorem 1. In most applications, since δ is approximate to 1, if $|\mathcal{M}| < k$, we can know that the distribution of the streaming data in s_n may be changed, and F_θ should be updated. In this case, when we access the objects in s_{n+1}, we maintain the ζ objects with the highest score in $\mathcal{M}$ via the Baseline algorithm. When we finish scanning s_{n+1}, we reset F_θ to the ζ-th highest score in s_{n+1}, and re-use the above algorithm for pruning the objects in s_{n+2}, i.e., setting $flag$ to 1. Otherwise, if $|\mathcal{M}| = k$, we do not update F_θ, and still use it to prune objects in s_{n+1}.

As shown in Fig.3, given a top-2 query, because $\delta = 0.9$, we set ζ to the solution of $\frac{\zeta-3}{\sqrt{\zeta}} = 2$, which is approximate to 8. Therefore, we use the 8th highest score in s_{n-1} as the pruning value F_θ, i.e., equal to 83. When scanning s_n, the first five objects are pruned directly since their scores are lower than 83. We insert 86 and 89 into $\mathcal{M}$ because their scores are higher than 83. Accordingly, we update the pruning value to $\min(\mathcal{M})$, i.e., 89. After scanning s_n, only four objects, i.e., 86, 89, 94, 98, are selected as candidates, and only two of them are false candidates. After processing all the objects in s_{n+1}, since no object's score in s_{n+1} is higher than 83, we reset F_θ to 0 when processing the objects in s_{n+2}.

We want to highlight that if the data distribution in s_n and s_{n-1} is roughly the same, $\Pr(|X| = O(k))$ is almost 1 after scanning s_n. Therefore, only $O(k)$ objects are en-heaped to $\mathcal{M}$ with a very high probability ($\geqslant \delta$). More importantly, we could guarantee that an exact result is falsely pruned with the probability at most $1-\delta$.

s_{n-1} 89 76 77 $\underline{83}$ 65 41 90 88 64 72 79 93 84 75 65 53 18 36 22
95 38 60 29 30 86
s_n 65 30 46 53 80 **86** 75 80 44 82 **89** 23 **94** 60 35 63 28 06 12
85 **98** 30 19 40 54
s_{n+1} 63 56 61 70 36 76 65 70 34 72 79 13 84 50 25 53 18 96 82
75 88 20 39 30 44

Fig.3. Pruning algorithm ($k = 2$, $\delta = 0.9$).

4.2.2 Algorithm Under Time-Related Data Distribution

In some applications, there is a strong correlation between the objects' arrival orders and their scores (e.g., the temperature in one day heavily relies on their measured time). Intuitively, if we can predict the peak values from a set of streaming data (e.g., p_a and p_b in Fig.4), we could find a stricter pruning value F_θ, which could filter more newly arrived objects. In the following, we mainly discuss how to find the proper pruning value, i.e., F_θ.

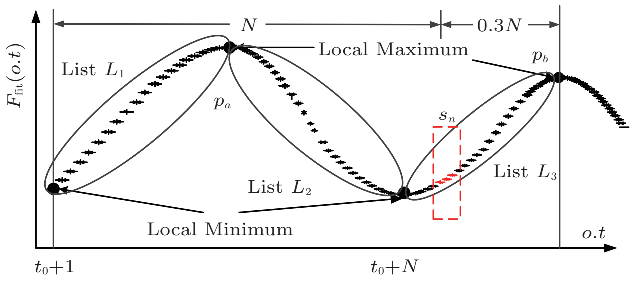


Fig.4. Predicted points selection algorithm.

We first model the correlation between objects' arrival orders and their scores. To achieve this, we try to fit objects' scores as a function of their arrival order, denoted as $F_{\text{fit}}(o.t)$ with $o.t$ referring to the arrival order of o. We use a small set of historical streaming data as our training set for training the fitting function. Given the fact that there are many approaches that can find the fitting function based on training values and the space is limited, we skip the details here. In our implementation, we adopt curving fitting.

After we acquire the fitting function $F_{\text{fit}}(o.t)$, we compute the pruning value F_θ according to (1). Here, $F_{\text{fit}}(\theta_1)$ equals the k-th highest fitting value among all the objects in $\{s_i, s_{i+1}, \ldots, s_{n-1}\}$, and $F_{\text{fit}}(\theta_2)$ equals the k-th highest fitting value among all the future objects in $\{s_{n+1}, s_{n+2}, \ldots, s_{n+i}\}$. σ^2 is the variance of the observed value and the fitted value, θ_{ub} is the k-th highest predicted score in s_n, and $\Phi(\Phi^{-1}(\delta)) = \delta$. Following Theorem 2, for each object in $o \in s_n$, if $F_\theta - F_{\rm fit}(o) \geqslant 0$, o cannot become the query result with the probability at least δ.

Theorem 2. *If $F_\theta - F_{\rm fit}(o) \geqslant 0$, the probability for o to be a top-k object is almost $1-\delta$.*

Proof. Let $W_1\{s_1, s_2, \ldots, s_n\}$ be the current window, and W_i be a future window formed by $\{s_i, s_{i+1}, \ldots, s_{n+i-1}\}$. For each object $o \in s_n$, if $F(o) \leqslant \theta_1$ and $F(o) \leqslant \theta_2$, o must be a meaningless object. Based on this observation, we compute θ_1 and θ_2 according to the fitting function. Because the difference between the observed value and fitted value follows normal distribution $N(\mu, \sigma^2)$, given two fitted values, the difference between their observed values follows normal distribution $N(\mu, 2\sigma^2)$. Thus, if (1) is satisfied, we have $\Pr(F_\theta - F(o) \geqslant 0) = \Phi(\frac{\min(F_{\rm fit}(\theta_2), F_{\rm fit}(\theta_1)) - F_{\rm fit}(o)}{2\sigma^2}) \geqslant \delta$. Thus, when $\min(F_{\rm fit}(\theta_2), F_{\rm fit}(\theta_1)) - F_{\rm fit}(o) \geqslant 2\Phi^{-1}(\delta)\sigma^2$, it is improbable that o can become the query result of any window. □

$$\min(F_{\rm fit}(\theta_2), F_{\rm fit}(\theta_1)) - 2\Phi^{-1}(\delta)\sigma^2 > \theta_{ub}. \quad (1)$$

In the following, we explain how to efficiently compute F_θ. We first compute θ_{ub} in s_n according to the fitting function. Second, we compute $F_{\rm fit}(\theta_1)$, where we reversely scan each peak point in the current window (e.g., p_a) according to the fitting function. For each peak point p, we search objects whose predicted scores are higher than $\theta_{ub} + 2\Phi^{-1}(\delta)\sigma^2$ around p. If we find k such objects $\{o_1, o_2, \ldots, o_k\}$, we set $F_{\rm fit}(\theta_1)$ to the smallest predicted score among these k objects. Third, we repeat the following operations to compute $F_{\rm fit}(\theta_2)$. After finding $F_{\rm fit}(\theta_1)$ and $F_{\rm fit}(\theta_2)$, we compute their maximum arrival order difference, denoted as d_t, among these $2k$ objects. If $d_t < \frac{N}{s}$, we set F_θ to $\min(F_{\theta_1}, F_{\theta_2})$. Otherwise, we set F_θ to $\theta_{ub} - 2\Phi^{-1}(\delta)$. Notice that in the streaming data environment, F_θ could be computed incrementally without incurring high cost.

Back to the example in Fig.4, we first access p_a to find k objects with fitting scores higher than $\theta_{ub} + 2\Phi^{-1}(\delta)\sigma^2$. We then access p_b to find k such objects. Since $F_{\rm fit}(\theta_2) < F_{\rm fit}(\theta_1)$ and $F_{\rm fit}(\theta_2) - 2\Phi^{-1}(\delta)\sigma^2 > \theta_{ub}$, we use $F_{\rm fit}(\theta_2)$ as the pruning value F_θ to prune the objects in s_n.

4.2.3 Algorithm with No Prior Distribution Knowledge

Obviously, the probability models help us enhance the filtering ability. However, if we have no prior knowledge on the distribution of streaming data, the above algorithms cannot work. In the following, we propose a novel algorithm to generate this pruning value when we have no prior knowledge on the distribution of streaming data.

To be more specific, given the object set s_n, we scan s_n to compute $EX(s_n)$. Here, $EX(s_n)$ refers to the expectation of the objects' scores in s_n. Based on $EX(s_n)$, we could generate the proper pruning value F_θ according to Theorem 3. Due to the limitation of space, we skip the details.

Theorem 3. *Given the value θ_x and an object $o_i \in s_n$, if θ_x satisfies $\Pr(F(o_i) > \theta_x) = p^*$, θ_x could be used as F_θ. Here, p^* is the solution of $\Phi(\frac{sp-k}{\sqrt{sp(1-p)}}) = \delta$.*

Proof. Let $\{o_1, o_2, \ldots, o_s\}$ be a set of objects in s_n such that $a \leqslant F(o_i) \leqslant b$ for all i and $\mu = E(X)$. Given object $o \in s_n$, $\Pr(F(o) > \theta_x) = p$. If s is big enough, we could use normal distribution $N(np, \sqrt{np(1-p)})$ to approximate $\Pr(|X| > k)$ according to the Demovire-Laplace theorem, with $n = s$, $p = \Pr(F(o_i) > F_\theta)$, $X = \{o | o \in s_n \wedge F(o) > \theta_x\}$. We then have $\Pr(|X| > k) = 1 - \Pr(|X| \leqslant k) = \Phi(\frac{sp-k}{\sqrt{sp(1-p)}})$. Lastly, we set p^* to the solution of the equation $\Phi(\frac{sp-k}{\sqrt{sp(1-p)}}) = \delta$, and compute the suitable F_θ according to Chernoff-Bound[20] (see (2)). Intuitively, for each object $o \in s_n$, if it is not contained in X, i.e., $F(o) < F_\theta$, the probability that o is a query result is less than $1-\delta$. Thus, θ_x could be used as the pruning value.

$$P(X \geqslant (1+\delta)\mu) \leqslant e^{-\frac{2\delta^2\mu^2}{n(b-a)^2}}. \quad (2)$$

□

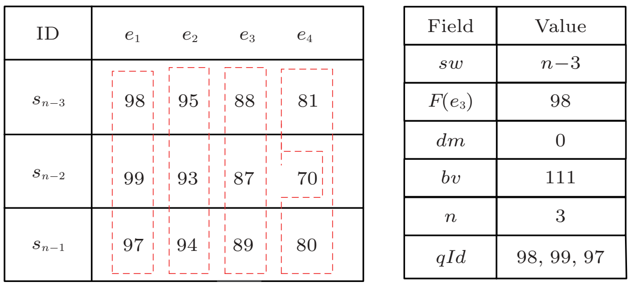
5 Merge Algorithms

After scanning s_n, we merge the non-pruned objects into the candidate set. In this phase, we should achieve the following two goals. First, we should efficiently maintain the score relationship among candidates, and remove meaningless ones. More importantly, to reduce the size of candidate set, we should combine candidates whose scores are roughly the same. In the following, we first propose a novel structure named QS to maintain the summary information of the combined candidates. We then propose two algorithms named Local-Merge and Global-Merge to maintain candidates.

After scanning s_n, we merge the non-pruned objects into the candidate set. In this phase, we should achieve the following two goals. First, we should efficiently maintain the score relationship among candidates, and remove meaningless ones. More important, to reduce the size of candidate set, we should combine candidates whose scores are roughly the same. In the following, we first propose a novel structure named QS to maintain the summary information of the combined candidates. We then propose two algorithms named Local-Merge and Global-Merge to maintain candidates.

5.1 QS Structure

In the following, we propose a structure named QS to maintain the summary information of candidates. Given a QS node e, it is represented as a tuple $(sw, F(e), dm, bs, qId, n)$ (see Fig.5). Let $O\{o_1, o_2, \ldots, o_u\}$ be the candidates combined by e. $e.sw$ records the lower-bound of these candidates' arrival orders; $F(e)$ is used for representing the approximate score of these candidates. Besides, $e.qId$ is a queue that maintains the "object Id" of these candidates; $e.n$ equals $|O|$; $e.bs$ is a bit string which will be discussed later; $e.dm$ denotes the dominate number of e. Note that the dominate number of QS node is defined in Definition 2, which is different from the explanation in Section 2.



ID	e_1	e_2	e_3	e_4
s_{n-3}	98	95	88	81
s_{n-2}	99	93	87	70
s_{n-1}	97	94	89	80

(a)

Field	Value
sw	$n-3$
$F(e_3)$	98
dm	0
bv	111
n	3
qId	98, 99, 97

(b)

Fig.5. QS structure, with $k = 4$, and $\varepsilon = 4$.

Definition 2 (Q-Dom). *Given an object o and a QS node e, e is dominated by o, denoted as $e \prec o$, if $\forall o' \in e$, $o.t \geqslant o't$ and $|F(o) - F(e)| \leqslant \frac{\varepsilon}{4}$. Here, $o.t$ refers to the arrival order of o. Given the window W, e's dominate number, denoted as $e.dm$, refers to the number of objects in W that can dominate e.*

We create e when a candidate $o' \in \mathcal{M}$ is combined with an object in $o \in b_m$. Here, b_m contains a subset of candidates, which has been introduced in Section 3. At that moment, we set $e.sw$ to the arrival order of o (i.e., $T(o)$). In addition, we set $e.F(e)$ to $F(o)$. Take Fig.5 as an example, in which a QS node e_1 summarizes three candidates. When 99 is inserted into b_m, we combine 99 with 98 and create e_1. Because $98 \in s_{n-3}$, we set $e_1.sw$ to $n-3$, and $F(e_1)$ to 98. The combination rule here will be discussed later. $e_1.dm$ is 0 since no object can dominate e_1. In addition, as e_1 summarizes the objects in s_{n-3}, s_{n-2} and s_{n-1}, we set $e_1.bs$ to 111, and set $e_1.n$ to 3. Lastly, we use qId to maintain the "object Id" of each candidate summarized by e_1. Note that we use the score of objects to represent "object Id" for ease of discussion here.

We want to highlight that QS nodes could efficiently summarize the arrival order of candidates, and provide valid objects to the query. An object is considered as valid if it is contained in the query window. For example, let the current moment be t. Given a QS node e, if $t < e.sw$, we deduce that no candidate summarized by e leaves the window. Otherwise, we could figure out which candidates summarized by e leave the window. In Fig.5, when the window slides from W_0 to W_{n-2}, because $n-2 > e_1.sw$ and $e_1.bs = 111$, we conclude that the first candidate summarized by e_1 has left the window. Above all, a QS node could use a few fields to maintain the summary of many candidates, and helps us effectively reduce the space cost.

5.2 Local Merge Algorithm

In this subsection, we propose the Local-Merge algorithm, which merges $\mathcal{M}$ into b_m. As discussed in Section 3, given the candidate set $\mathcal{B}$, a partition $\mathcal{P}(\mathcal{B}, m) = \{b_1, b_2, \ldots, b_m\}$ is to partition the elements in $\mathcal{B}$ into m buckets $\{b_1, b_2, \ldots, b_m\}$. The reason why we introduce partition will be discussed later. In the following, we first discuss how to combine the candidates together. Next, we explain how to maintain the QS nodes.

Since the objects (or QS node) in $\mathcal{M}$ and b_m are ordered, the combination can be converted to the merge sort problem. One apparent benefit is that we could combine as many elements as possible in $\mathcal{M}$ and b_m together if their scores are roughly the same. As shown in Algorithm 2, we sequentially scan these two sets, and try to find the m-pair (see Definition 3) for each object $o \in \mathcal{M}$. If we cannot find the corresponding m-pair for o, we insert o into b_m according to its score. Following Theorem 4, if an object is combined with an m-pair, our query algorithm could guarantee the deviation of the approximate results is no more than parameter ε.

Definition 3 (m-Pair). *Given an object $o \in s_n$ and a QS node e, if $|F(o) - F(e)| \leqslant \frac{\varepsilon}{4}$ and e has not com-*

bined other objects in s_n, we call e as a pair node of o. Among all the pair nodes of o, we call the node with minimal score as m-pair.

Algorithm 2. Local-Merge Algorithm

```
Input: candidate set $\mathcal{M}$, bucket $b_m$
Output: bucket $b_m$
Initialization: int $index \leftarrow |b_m| - 1$;
for $i$ from $k-1$ to 0 do
    if $|F(b_m[j]) - F(\mathcal{M}[i])| \geqslant \frac{\varepsilon}{4} \wedge F(\mathcal{M}[i]) > F(b_m[j])$ then
        $b_m[j].dm \leftarrow b_m[j].dm + i$;
        if $b_m[j].dm \geqslant k$ then
            Delete $(b_m[j])$;
        $j \leftarrow j - 1$, break;
    if $|F(b_m[j]) - F(\mathcal{M}[i])| \leqslant \frac{\varepsilon}{4}$ then
        Int $sum \leftarrow b_m[j].dm + i$;
        if $b_m[j].dm \geqslant k$ then
            Clear $(b_m[j])$;
            Re-build $(b_m[j], \mathcal{M}[i])$;
        Combine $(b_m[j], \mathcal{M}[i])$;
        furtherCheck $(b_m, \mathcal{M}[i], index)$;
        $j \leftarrow j - 1$, break;
    if $|F(b_m[j]) - F(\mathcal{M}[i])| \geqslant \frac{\varepsilon}{4} \wedge F(\mathcal{M}[i]) > F(b_m[j])$ then
        Insertion $(\mathcal{M}[i])$;
Return;
```

In Fig.6, objects in $\mathcal{M}$ are scanned based on the order of 82 first and then 91; the nodes in b_m are scanned based on the order of e_4 first, e_3 second, and so on. Because $F(e_4) - 82 \leqslant \frac{\varepsilon}{4}$ and e_4 has not combined other candidates, we regard e_4 as the m-pair of 82. For 91, since we cannot find m-pair for it, it is inserted into b_m. Similarly, we combine 94 and 98 with e_2 and e_1 respectively. Notice that nodes in b_m can be candidates or QS nodes, but for ease of presentation, we only consider QS nodes in the rest of this paper.

Theorem 4. *Given a QS node e in $\mathcal{B}$, and an object o summarized by e, if $|F(o) - F(e)| \leqslant \frac{\varepsilon}{4}$, we could guarantee the score deviation between exact results and approximate result is smaller than ε.*

Proof. Let $\{e_1, e_2, \ldots, e_n\}$ be the nodes contained in $\mathcal{B}$, and they are sorted in descending order by their $F(e)$ field. Assuming the query results are from $\{e_1, e_2, \ldots, e_i\}$, i.e., $\sum e_i.n \geqslant k$ and $\sum e_{i-1}.n \leqslant k$, we first prove $F(o_k) - F(r_k) \leqslant \varepsilon$. Here, r_k and o_k are the objects with the k-th highest score in the exact result set and the approximate result set respectively, where we should guarantee that $\max(e_{i+1}) - \min(e_i) \leqslant \varepsilon$. If $|F(e_{i+1}) - F(e_i)| \geqslant \frac{\varepsilon}{2}$, because $|F(o) - F(e)| \leqslant \frac{\varepsilon}{4}$, we have $\max(e_{i+1}) - \min(e_i) \leqslant \varepsilon$. In this case, the deviation is no more than ε. Otherwise, if $|F(e_{i+1}) - F(e_i)| \leqslant \frac{\varepsilon}{2}$, because $|\max(e_{i+1}) - \min(e_i)| = \max(e_{i+1}) - F(e_{i+1}) + F(e_{i+1}) - \min(e_i)$, and $F(e_i) - \min(e_i) \leqslant \frac{\varepsilon}{4}$, we have $|\max(e_{i+1}) - \min(e_i)| \leqslant \varepsilon$. $\min(e_i)$ refers to the minimal score in e_i, and $\max(e_{i+1})$ refers to the maximal score in e_i. Therefore, if $|F(o) - F(e)| \leqslant \frac{\varepsilon}{4}$, $F(o_k) - F(r_k) \leqslant \varepsilon$. Similarly, if $|F(o) - F(e)| \leqslant \frac{\varepsilon}{4}$, $F(o_i) - F(r_i) \leqslant \varepsilon$ for all i. □

Next, we explain how to maintain QS nodes and delete meaningless QS nodes. As discussed above, the nodes in b_m can be divided into two types: paired nodes and non-paired nodes. Here, paired nodes (or non-paired nodes) refer to the nodes in b_m that have (or not) combined objects in s_n. In the following, we first explain how to maintain paired nodes.

Given a paired node e, if $|\mathcal{D}_e| = i$, we update $e.dm$ to i, where $\mathcal{D}_e$ refers to a set of candidates o in $\mathcal{M}$ that can dominate e. The reason is: given an object $o \in \mathcal{M}$ and an object o' combined by e, o must satisfy: 1) $F(o) > F(o')$ or 2) $|F(o) - F(o')| \leqslant \frac{\varepsilon}{4}$. In other words, we could guarantee there are k objects satisfying $F(o) \geqslant F(e) - \varepsilon$. According to the problem definition, when $e.dm$ reaches k in the future, even if we delete e, these k objects could be used for substituting the candidates summarized by e to answer the query.

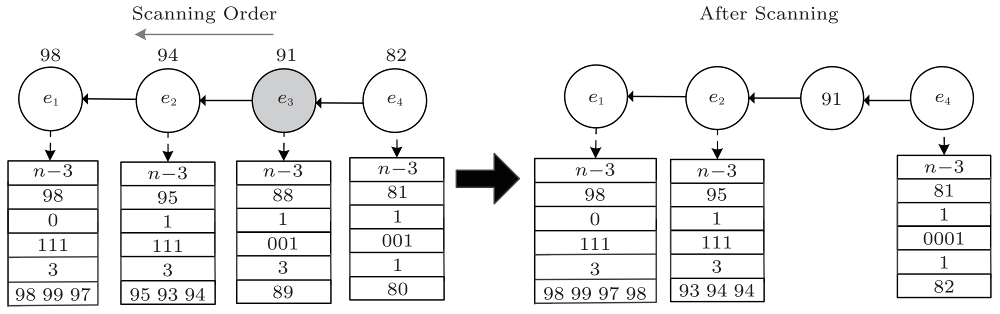


Fig.6. Example of candidates combination and QS-node maintenance.

The other fields of e are updated according to the value $e.dm+i$. If $e.dm+i \geqslant k$, we clear $e.qId$ since all the elements combined in e become meaningless. We then update $e.bs$ according to (3). Lastly, since o becomes the first object combined by e, we update $e.n$ to 1, push $o.Id$ into $e.qId$, and set $e.sw$ to n. Otherwise, if $e.dm+i<k$, we firstly push $o.Id$ into $e.qId$, and then update $e.n$ to $e.n+1$. Lastly, we update $e.bs$ according to (3).

$$e.bs = \begin{cases} (e.bs << l)+1, & \text{if } e.dm+i<k, \\ 1, & \text{if } e.dm+i \geqslant k. \end{cases} \qquad (3)$$

Given a non-paired node e', because it has been dominated by $e'.dm$ objects before the combination algorithm is invoked, if there exist j objects in $\mathcal{M}$ dominating e', we update $e'.dm$ to $e'.dm+j$. In particular, if $e'.dm$ reaches k, we delete it (see e_3 in Fig.6). Otherwise, we firstly push $o.Id$ into $e.qId$, and then update $e.n$ to $e.n+1$. Lastly, we update $e.bs$ according to (3).

In Fig.6, since e_4 combines 82, we call it as a pair node. Because $e_4.dm+4>k$, we clear $e_4.qId$. We then set $e_4.n$ to 1, push 82 into $e_4.qId$, and update $e_4.bs$ to 1. By contrast, since e_3 is a non-paired node, and there are three objects in s_n dominating e_3, we update $e_4.dm$ to 4 $(=1+3)$. Therefore, we delete it directly.

5.3 Global Merge Algorithm

After Local-Merge, if $|b_m| \leqslant \varphi k$, the candidates maintenance algorithm is terminated. Otherwise, if $|b_m|>\varphi k$, we merge b_m with b_{m-1}. Here, φk is the maximal capacity of b_m, and b_{m-1} is another bucket with the capacity $\varphi^2 k$. Similar to Local-Merge, Global-Merge also needs to achieve the following goals: 1) combining QS nodes together if the scores of QS nodes are roughly the same; 2) deleting the meaningless nodes. In the following, we first propose the structure, named G-QS (short for group QS), to summarize the QS nodes that have similar "score". Thereafter, we explain the G-QS-based global merge algorithm.

The G-QS node is used for summarizing QS nodes. To be more specific, given a G-QS node g, and the QS nodes $\{u_1,u_2,\ldots,u_s\}$ combined in g, we partition $\{u_1,u_2,\ldots,u_s\}$ into a group of subsets $\{u_1,u_2,\ldots,u_{s_1}\}$, $\{u_{s_1+1},u_{s_1+2},\ldots,u_{s_2}\}$, $\ldots$, $\{u_{s_{v+1}},u_{s_{v+2}},\ldots,u_{s_s}\}$. In each group, the elements should be from the same bucket before they are merged into b_{m-1}, e.g., u_1,u_2,u_3 in s_2. They are stored in a structure called s-unit. In order to further reduce the space cost, an s-unit node only maintains the bs, qId, sw fields for each QS node (see Fig.7). In addition, an s-unit node maintains the lower-bound of these nodes' dominate number, i.e., $s_2.dm=\min(u_1.dm,u_3.dm,u_3.dm)$ in Fig.7. Among all the s-units in g, we use a queue $g.qv$ to maintain them.

For the other fields of G-QS node, the meaning of $g.F(g)$, $g.sw$, and $g.n$ is similar to that of QS node respectively. The difference is that $g.dm$ equals the lower-bound of $|\mathcal{D}_w|$. Here, $\mathcal{D}_w$ refers to a set of objects that can dominate the QS nodes stored at the head of $g.qv$ (i.e., the nodes stored in s_1 in Fig.7). We will discuss its function later.

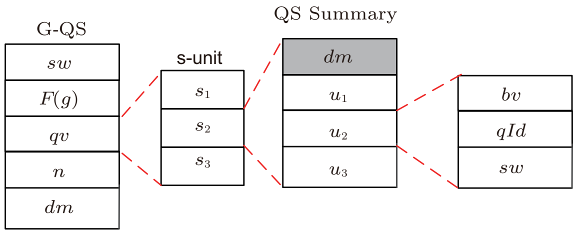


Fig.7. Queue domination based G-QS structure.

In the following, we first discuss the G-QS-based combination algorithm. Compared with the algorithm discussed in Subsection 5.2, we allow a G-QS node to combine several QS nodes. Therefore, the combination rule is as follows: given a G-QS node g and a QS node e, if $|F(g)-F(e)| \leqslant \frac{\varepsilon}{4}$, g could combine e. The other details are similar to that of Local Merge. For the limitation of space, we skip the details.

Next, we explain the meaningless nodes deletion algorithm. Similar to the Local-Merge algorithm, the combination algorithm discussed in this subsection also divides the nodes in b_{m-1} into two types: paired nodes and non-paired nodes. In the following, we first explain how to maintain the dominate number of paired nodes, and how to delete the meaningless summaries contained in them. Here, the paired nodes refer to the nodes in b_{m-1} that have combined QS nodes in b_m.

Given a paired G-QS node g, if g combines $\{e_i,e_{i+1},\ldots,e_s\}$, we first compare $\sum \mathcal{D}_m[i].n$ with k. Here, $\mathcal{D}_m$ refers to a subset of b_m, where the elements e in b_m should satisfy $F(g)-F(e) \leqslant \frac{\varepsilon}{4}$, i.e., $\{e|e \in b_m \wedge F(g)-F(e) \leqslant \frac{\varepsilon}{4}\}$. If $\sum \mathcal{D}_m[i].n \geqslant k$, we clear $g.qv$. Otherwise, we add $g.dm$ to $g.dm+\sum \mathcal{D}_m[i].n$. After adding, if $g.dm>k$, we deduce that the candidates summarized by $g.qv[0]$ are all meaningless, and we pop it accordingly. We then update $g.dm$ to $g.dm$-$g.qv[0].dm$. Here, $g.qv[0]$ is the s-unit at the head of $g.qv$. We repeat the above operations until $g.dm<k$.

Lastly, we initialize a new s-unit u for $\{e_i, e_{i+1}, \ldots, e_j\}$, and push u into $g.qv$. For the other non-paired nodes g', we also compute lb_m firstly. If $lb_m \geqslant k$, we delete g' directly. Otherwise, if $lb_m \leqslant k$, we repeat the operations discussed above to delete meaningless summaries from $g'.qv$. For the other fields of G-QS nodes, the maintenance approach is similar to that of G-QS node. For the limitation of space, we skip the details.

After merging b_m into b_{m-1}, if $|b_{m-1}|$ is larger than $\varphi^2 k$, we merge b_{m-1} with b_{m-2} via the Global-Merge algorithm. The Global-Merge merge will be terminated when we could find bucket b_i that satisfies $|b_i| < \varphi^{m-i+1}k$.

There are two points we want to highlight. First, we combine the candidates together if their scores are roughly the same, and the size of each bucket could be effectively reduced. Because we employ the merge sort algorithm, and the cost of merge heavily lies on the size of merged sets, the combination scheme helps us reduce the computation cost a lot. In addition, because we partition the candidate set into a group of buckets, the merge cost could be further reduced.

5.4 Cost Analysis

As discussed above, both Local-Merge and Global-Merge can be converted to the merge sort problem. Therefore, the computation cost mainly depends on the size of each bucket. Let the candidate set $\mathcal{B}$ be $\{b_1, b_2, \ldots b_m\}$. If b_{i+1} merges into b_i, the total computation cost is $O((\varphi^{m-i+1} + \varphi^{m-i+2})k)$. If we amortize this part of cost to every element in b_{i+1}, the computation cost is $O(\varphi)$. In addition, since each candidate is merged at most m times before it leaves the window, and the amount of nodes in $\mathcal{B}$ is bounded by $O(\frac{R}{\varepsilon})$, m is bounded by $O(\log_\varphi \frac{R}{\varepsilon k})$. R here is the range of objects' scores. Therefore, the total computation cost spent in each candidate is shown in (4). Given a sidling window with the size N, the total computing cost of processing N is $O(N \log k + \frac{Nk}{s} \log_\varphi \frac{R}{\varepsilon k} + N \times cost_F)$. We use $cost_F$ to represent the cost of deriving the score for each object o. As for φ, we can find its optimal value (denoted as φ^*) by solving the function $(\varphi \log_\varphi \frac{R}{\varepsilon k})' = 0$.

$$M_{\text{cost}} = \varphi \log_\varphi \frac{R}{\varepsilon k}. \quad (4)$$

6 Query Algorithm

In this section, we propose the TAHM algorithm to support the top-k query, which employs the key of the TA-algorithm and the Heap-Merge algorithm. Specially, given a top-k query q, we maintain the k' objects with the highest scores in $\mathcal{R}$. The elements in $\mathcal{R}$ may be exact results or approximate results, and $|\mathcal{R}|$ should satisfy $k < |\mathcal{R}| < 2k$. We maintain $\mathcal{R}$ after scanning s_n, and compare each object o_i in $\mathcal{M}$ with $\mathcal{R}$. If $F(o_i) > F(\min(\mathcal{R}))$, we insert o_i into $\mathcal{R}$. After insertion, if $|\mathcal{R}| > 2k$, we delete $\min(\mathcal{R})$. Notice that, to maintain the summary information of o_i even when it is removed from $\mathcal{R}$, we insert a copy of o_i into $\mathcal{B}$ when inserting it into $\mathcal{R}$. When an object in $\mathcal{R}$ leaves the window, we delete it from $\mathcal{R}$. In particular, if $|\mathcal{R}| < k$, we retrieve the new answer from $\mathcal{B}$.

Since the elements in $\mathcal{B}$ are distributed in the buckets $\{b_1, b_2, \ldots, b_m\}$, and the elements in each b_i are sorted in descending order of their scores, i.e., the $F(e)$ field of each node, we employ Heap-Merge algorithm to re-construct $\mathcal{R}$. Specifically, we use a max-heap $\mathcal{H}$ to maintain the nodes with the highest scores in each b_i. When $\mathcal{R}$ is re-constructed, we first access the root of $\mathcal{H}$. Let e be the root, and it is a QS node. We compute how many valid objects are summarized by e according to $e.n$, $e.sw$ and $e.bs$. If there exist more than k valid objects, we insert k objects in $e.qId$ into $\mathcal{R}$. Otherwise, we insert all the valid objects summarized by e into $\mathcal{R}$. After insertion, we adjust $\mathcal{H}$. If the root is a G-QS node, the maintenance algorithm is similar, and we skip the details. We repeat the above operations until $|\mathcal{R}|$ reaches $2k$.

As depicted in Fig.8, we re-construct $\mathcal{R}$ when the window slides to W_{41}. For simplicity, we assume that each b_i only has QS-nodes. We first access the QS-node e_1. Since $e_1.sw = 40$, there may exist invalid candidates in e_1, which is 99. Therefore, we ignore it, and only insert 100 into $\mathcal{R}$. We then access e_2, where we insert 99, 98 and 97 into $\mathcal{R}$. After insertion, since $|\mathcal{R}|$ increases to 6, we finish re-constructing $\mathcal{R}$.

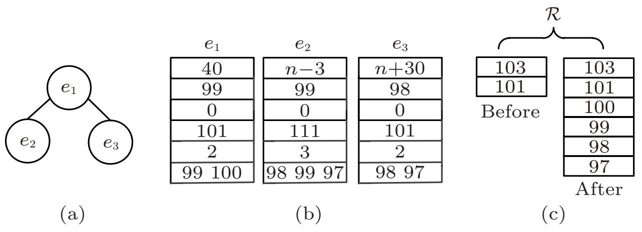


Fig.8. Query algorithm under W_{41}, with the parameters $k = 3$ and $\varepsilon = 4$.

Note that, PABF could catch most of high-score objects. Among all the elements in $\mathcal{R}$, only 97 may be an approximate result. Therefore, when answering top-k query, PABF usually could return a high-quality result to the system.

7 Performance Evaluation

In this section, we conduct extensive experiments to demonstrate the efficiency of the PABF framework. The experiments are based on one real dataset, namely Stock. Stock refers to 1 GB stock transactions from ShangHai/ShenZhen Stock Exchange in 2011∼2014, with the original size of 30 GB. Each record contains four attributes (i.e., stock Id, transaction time, volume, and price). They are sorted based on the ascending order of their transaction time. The preference function F is $price \times volume$.

In addition, we also generate 10 synthetic datasets. The first two datasets are called as Time^R and Time^U respectively. Both sets have 1 GB objects where the scores of objects in Time^U are not related to their arrival orders, and the scores of objects in Time^R are decided by the function $F(o) = \sin(\frac{\pi \times o.t}{2N})$. We assume the score distributions of both datasets are known, and employ them to simulate the cases where the distribution of objects' scores is known in advance.

The 3rd to 10th synthetic datasets are used for evaluating the performance of algorithms w.r.t. data skew. Skew-I(a)∼Skew-I(d) correspond to the 3rd∼6th dataset respectively, where objects' scores are unrelated to their arrived time. Their scores are crowed in the range of preference function value $\{80\%, 60\%, 40\%, 20\%\}$. The 7th∼10th synthetic datasets correspond to Skew-II(a)∼Skew-II(d), where objects' scores are related to their arrived time. Given two objects o_1 and o_2, if $T(o_1) > T(o_2)$, $\Pr(F(o_1) < F(o_2))$ is $\{0.1, 0.4, 0.7, 0.9\}$ in Skew-II(a)∼Skew-II(d) respectively.

In our study, we consider five parameters, including the window size N, the number of returned objects k, the value s, the deviation ε, and the probability δ. The parameters settings are listed in Table 1 with the default values bolded. R refers to the range of the preference function; $|D|$ refers to the size of dataset. To give a comprehensive evaluation on the performance, we take the total running time, memory size, actual deviation and accuracy as the main metrics. Here, the total running time records the time used to process all the objects in the dataset. Memory size refers to the space cost used for maintaining candidates. Deviation indicates the score difference between the approximate results and the exact results. Accuracy evaluates the quality of results. To be more specific, let the approximate result set be $\{r_1, r_2, \ldots, r_k\}$. c records how many objects in the approximate result set are also contained in the exact result set. The accuracy is calculated as $\frac{c}{k}$.

Table 1. Parameter Settings

Parameter	Value
N	0.01%, 0.05%, **0.1**%, 0.2%, 0.5%, 1% (×$\|D\|$)
k	10, 50, **100**, 200, 500, 750, 1 000
s	0.01%, 0.05%, **0.1**%, 1%, 2%, 5%, 10% (×N)
ε	0.000 1, **0.001**, 0.01, 0.05, 0.1 (×R)
σ	0.999 9, 0.999, 0.99, 0.95, 0.9

In addition to PABF framework, we implement SMA and MinTopK algorithms, two state-of-the-art approaches for answering continuous top-k query, as the representatives of multi-pass based approaches and one-pass based approaches respectively. All the algorithms are implemented with C++, and all the experiments are conducted on a PC with an i7 CPU and 32 GB memory, running Microsoft Windows 7.

7.1 Effect of Approximation

In our first set of experiments, we first evaluate the effectiveness of PABF under different ε and δ. We report the running time of PABF with different ε and δ in Table 2. From Table 2 (column 3∼column 7), we find that with the increase of ε, the running time of PABF decreases. The reason is that the higher ε is, the more candidates could be combined together, and the lower the merge cost is. However, after ε increases to 0.01, the running time of PABF is not drastically changed.

Table 2. Performance Analysis Under Different Parameters

Dataset	Parameter	Running Time Analysis (s)					Deviation Analysis (%)					Accuracy Rate Analysis (%)				
		0.01%	0.05%	0.1%	0.5%	1%	0.01%	0.05%	0.1%	0.5%	1%	0.01%	0.05%	0.1%	0.5%	1%
Stock	ε	36.64	32.38	29.11	27.13	26.82	0.003 0	0.021 0	0.030	0.240	0.460	86.3	64.5	53.3	48.6	45.1
	δ	30.64	29.68	29.11	28.84	28.12	0.027 0	0.028 0	0.029	0.030	0.033	66.0	64.5	63.9	63.1	61.3
Time^U	ε	32.70	30.80	28.90	25.10	22.30	0.002 8	0.009 7	0.033	0.130	0.430	86.0	64.0	59.0	53.0	49.0
	δ	27.60	28.50	28.90	30.60	31.30	0.037 0	0.035 0	0.033	0.032	0.031	65.9	64.0	63.2	62.3	61.7
Time^R	ε	52.70	43.10	38.10	31.60	27.80	0.005 0	0.023 0	0.034	0.120	0.330	82.1	73.0	71.9	65.8	60.3
	δ	40.40	38.60	38.10	37.40	36.80	0.030 0	0.031 0	0.033	0.034	0.036	75.5	73.0	72.4	70.1	69.2

The reason is that when ε is high, the running time of PABF is mainly spent on pruning the objects in s_n. As for parameter δ, we can see that the running time of PABF is not obviously affected by changing the value of δ. The main reason is that PABF could effectively prune newly arrived objects in s_n regardless of the value of δ. Thus, we can conclude that parameter δ does not heavily impact the running time of PABF.

We next report the actual deviation of our PABF under different ε and δ. According to Table 2 (column 8~column 12), we can see that the actual deviation under PABF is increasing with ε, which means parameter ε impacts the quality of the query results. However, we also notice that the actual deviation of PABF is much smaller than ε. The reason is that both QS nodes and G-QS nodes could summarize the feature of high scores in the window. Therefore, even if ε is large, the actual deviation is also not high. For the parameter δ, with the increase of δ, the actual deviation under PABF does not change a lot. The reason is that even if δ is high, because the objects pruned by the filter do not have high score, they are still almost impossible to become the query result. Therefore, parameter δ also does not greatly impact the deviation of PABF.

When it comes to accuracy, from Table 2 (columns 13~18), we can figure out there is a negative correlation between accuracy and ε. This is because the higher ε is, the more the candidates combined together, leading to relatively larger score difference among them. Thus, when ε is $0.01 \times R$, the accuracy of PABF is relatively low. We can also see that the accuracy of PABF does not fluctuate too much when δ is increasing, which means δ has no significant impact on the accuracy. The reason for this is similar to the one discussed above.

Lastly, we report the performance of our proposed algorithms w.r.t. data skew. From Skew-I(a) to Skew-I(d), where objects' scores are irrelevant to their arrival orders, we can see that neither the running time nor the deviation of PABF is impacted too much when data distribution gets more skewed (see lines 2~5 of Table 3). The reason is that the pruning algorithms and candidates maintenance algorithms are unsensitive to data distribution. The accuracy of PABF is slowly decreased when data distribution becomes increasingly skew. The reason is that Local-Merge and Global-Merge could effectively summarize the summary information of candidates. Therefore, even if the data distribution is highly skew, the quality of result is not decreased a lot.

When the objects' scores are related to their arrival order, i.e., under Skew-II(a)~Skew-II(d), we find that the performance of PABF is stable. The reason is that our pruning algorithms could self-adaptively adjust the pruning value, and filter most of the newly arrived objects. In addition, our proposed merge algorithms could support the query via maintaining the summary information of candidates. Therefore, this part of cost is also stable under various data distributions.

Table 3. Performance Analysis w.r.t. Data Skew

Dataset	Parameter	Running Time Analysis (s)					Deviation Analysis (%)					Accuracy Rate Analysis (%)				
		0.01%	0.05%	0.1%	0.5%	1%	0.01%	0.05%	0.1%	0.5%	1%	0.01%	0.05%	0.1%	0.5%	1%
Skew-I(a)	ε	31.5	28.8	26.9	23.2	21.3	0.003 9	0.022 0	0.032	0.260	0.350	82.3	65.5	63.7	52.6	44.3
	δ	28.8	27.6	26.9	25.7	23.8	0.030 0	0.031 0	0.032	0.033	0.034	65.9	65.6	63.7	62.9	61.4
Skew-I(b)	ε	30.1	28.1	27.4	25.3	24.7	0.003 7	0.002 0	0.029	0.300	0.340	80.9	69.1	64.5	50.7	40.6
	δ	28.2	27.9	27.4	27.0	26.6	0.027 0	0.028 0	0.029	0.031	0.032	66.2	65.6	64.5	62.9	62.1
Skew-I(c)	ε	29.8	26.8	25.4	24.3	22.8	0.003 5	0.001 7	0.027	0.270	0.320	78.6	60.1	59.2	48.5	39.4
	δ	26.4	24.8	25.4	26.1	25.1	0.025 0	0.026 0	0.027	0.028	0.029	65.2	62.6	59.2	58.9	57.1
Skew-I(d)	ε	27.7	25.1	23.1	21.8	20.5	0.003 3	0.016 0	0.025	0.250	0.310	76.1	60.6	57.0	52.9	38.8
	δ	23.8	23.4	23.1	22.9	22.6	0.023 0	0.024 0	0.025	0.026	0.027	60.5	58.6	57.0	55.4	55.1
Skew-II(a)	ε	29.7	27.8	26.9	25.1	24.3	0.004 0	0.019 0	0.028	0.120	0.260	81.9	74.3	62.0	51.4	38.8
	δ	28.4	27.6	26.9	26.4	25.8	0.026 0	0.027 0	0.028	0.028	0.029	63.9	63.6	62.0	61.4	60.1
Skew-II(b)	ε	31.2	30.3	29.4	28.7	27.8	0.004 0	0.021 0	0.029	0.140	0.270	80.2	72.1	59.3	46.9	36.2
	δ	30.1	29.8	29.4	28.6	28.4	0.027 0	0.028 0	0.029	0.029	0.030	61.1	60.7	59.3	58.4	57.1
Skew-II(c)	ε	34.2	33.7	32.1	30.5	28.6	0.005 0	0.022 0	0.029	0.150	0.280	78.1	65.6	54.0	44.1	34.8
	δ	35.3	33.4	32.1	31.5	30.6	0.028 0	0.028 0	0.029	0.029	0.029	57.5	55.6	54.0	51.4	50.1
Skew-II(d)	ε	36.5	34.5	33.4	32.1	31.9	0.005 0	0.023 0	0.030	0.160	0.290	76.3	63.2	51.4	42.8	33.5
	δ	34.2	33.8	33.4	33.1	32.7	0.029 0	0.029 0	0.030	0.030	0.031	52.5	51.6	51.4	50.1	49.7

7.2 Comparison Between PABF and Existing Algorithms

In the following, we compare the performance of PABF with MinTopK and SMA. we set ε and δ to $0.1\% \times R$ and 0.99 respectively. We first evaluate the performance of different algorithms under STOCK, where the distribution of the score is unknown. From Fig.9(a) to Fig.9(f), we first observe that PABF performs the best in terms of running time. For example, PABF on average consumes only 40% of MinTopK's running time and 20% of SMA's running time under various N. This is no surprise since PABF can prune most of the newly arrived objects through the pruning value. In addition, because elements in each bucket are much fewer than the corresponding candidates, the merge cost could be reduced a lot. In terms of space cost, the memory consumed by MinTopK is higher than that of PABF (1.6 times). This is because MinTopK maintains all the meaningful objects, while PABF only needs to maintain the summary information of meaningful candidates. Since SMA indexes all the objects in the window, we do not compare its space cost with other algorithms.

We then demonstrate the performance of different algorithms under TIME^R and TIME^U. As shown in Fig.10 and Fig.11, PABF performs the best in terms of both running time and space cost with significant advantage. In addition, there are a few new observations that worth discussion.

For example, in Fig.11, when the window size is smaller than that in the period of the periodic function (e.g., 2 MB for function $\sin(\frac{o.t\pi}{1\ \text{MB}})$, SMA requires long running time because it has to incessantly re-scan. By contrast, both QS node and G-QS node could use only a few bytes to maintain the summary information of many candidates, and the performance of PABF is still high.

Lastly, we report the performance of different algorithms w.r.t. data skew. We first evaluate the performance of different algorithms under SKEW-I(a)$\sim$SKEW-I(d). According to Fig.12, as data distribution gets more and more skewed, the difference of the running time between PABF and SMA becomes larger. It is because when the distribution of score is skew, the re-scanning cost of SMA is high.

We then evaluate the performance of PABF when objects' scores are related to their arrived time, under SKEW-II(a)$\sim$SKEW-II(d). We find that the running time of PABF remains stable. However, the running time of both SMA and minTopk dramatically increases. The reason is that if the scores of objects in the window demonstrate a downtrend, minTopk has to maintain many candidates, and the re-scanning time of SMA is larger.

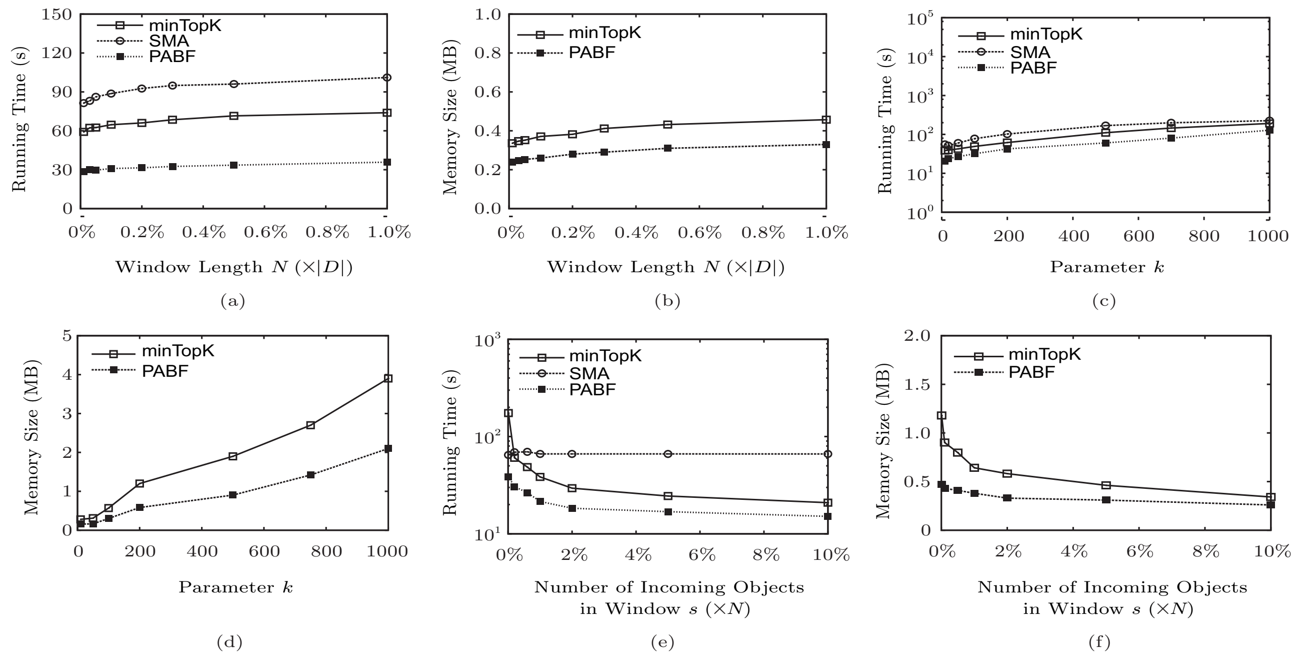


Fig.9. Performance under STOCK. (a) Running time vs N. (b)Memory size vs N. (c) Running time vs k. (d) Memory size vs k. (e) Running time vs s. (f) Memory size vs s.

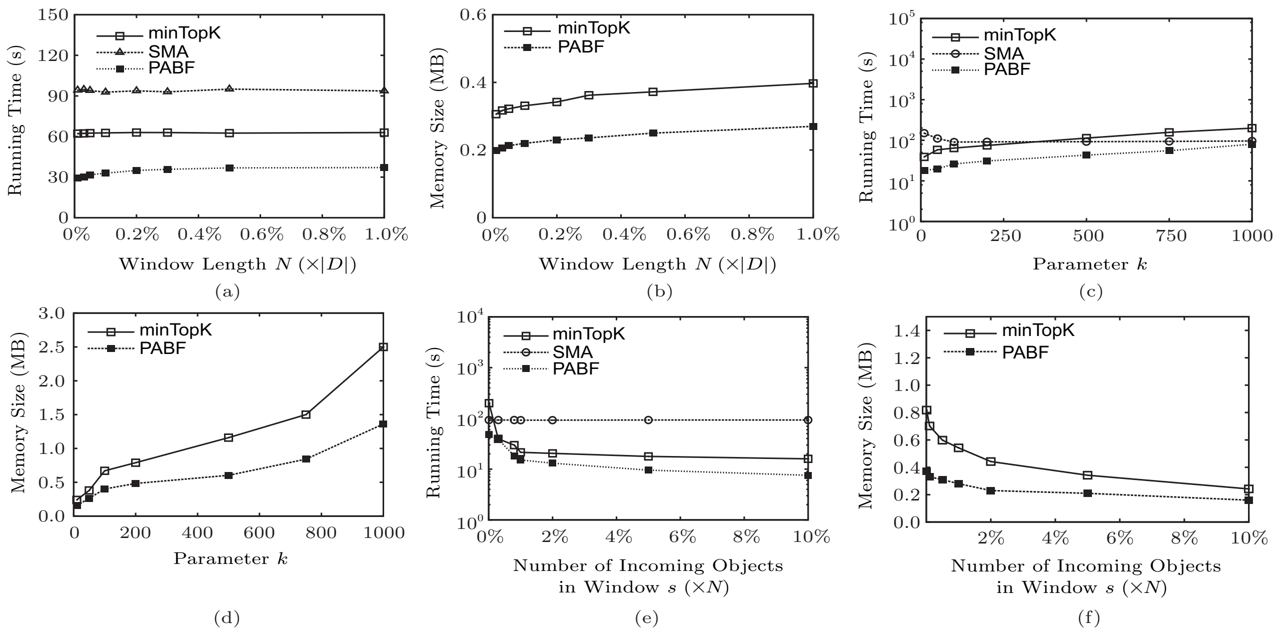


Fig.10. Performance under TIME^U. (a) Running time vs N. (b) Memory size vs N. (c) Running time vs k. (d) Memory size vs k. (e) Running time vs s. (f) Memory size vs s.

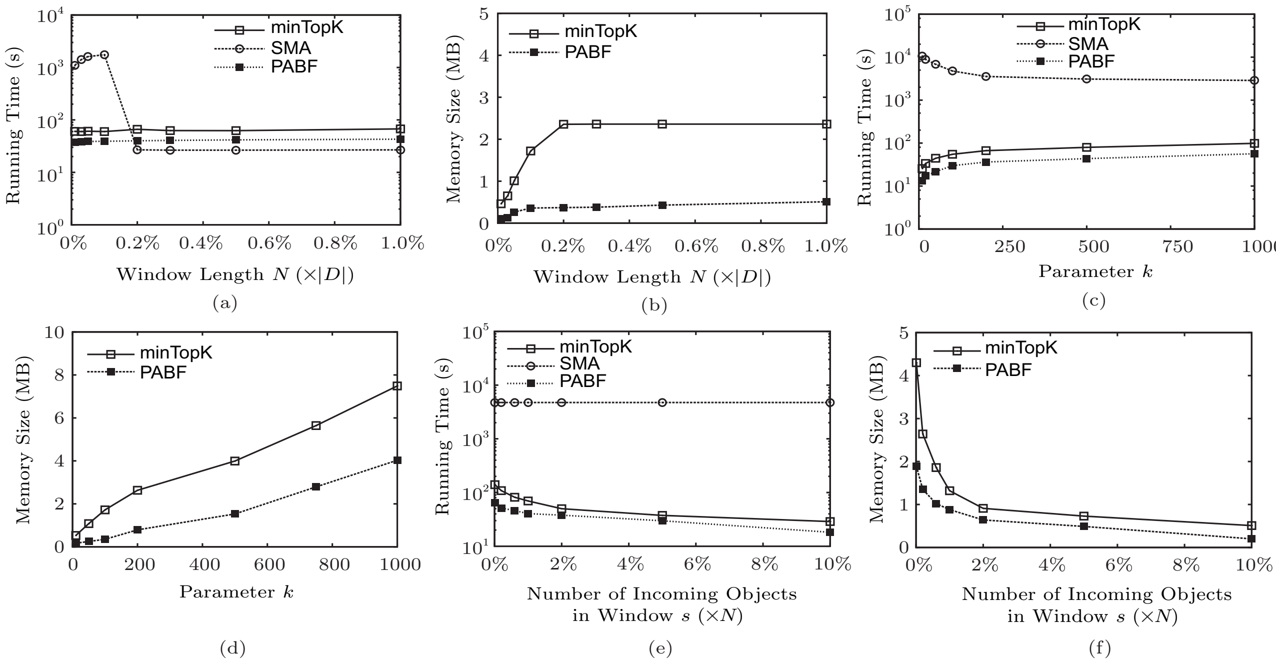


Fig.11. Performance under TIME^R. (a) Running time vs N. (b)Memory size vs N. (c) Running time vs k. (d) Memory size vs k. (e) Running time vs s. (f) Memory size vs s.

8 Conclusions

In this paper, we proposed a novel and general framework named PABF, for supporting approximate continuous top-k query over stream data. Unlike all the existing studies, PABF returns a set of k objects with relatively high scores in the window instead of exact top-k results. PABF firstly prunes most of the "low quality" objects directly when they arrive in the window. It then uses a few QS nodes and G-QS nodes to maintain the summary information of candidates. In addition, the multi-phases merge approach could help

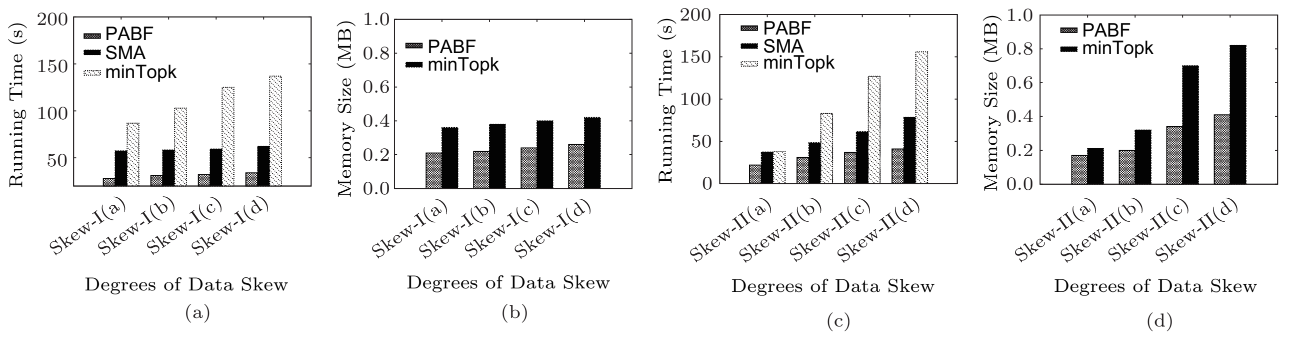


Fig.12. Performance under different algorithms w.r.t. data skew. (a) Running time vs N. (b) Memory size vs N. (c) Running time vs k. (d) Memory size vs k.

us further reduce the computation cost. We conducted extensive experiments to evaluate the performance of PABF. The results demonstrated the superior performance of PABF.

References

[1] Yang D, Shastri A, Rundensteiner E A, Ward M O. An optimal strategy for monitoring top-k queries in streaming windows. In *Proc. the 14th International Conference on Extending Database Technology*, March 2011, pp.57-68.

[2] Mouratidis K, Bakiras S, Papadias D. Continuous monitoring of top-k queries over sliding windows. In *Proc. ACM SIGMOD International Conference on Management of Data*, June 2006, pp.635-646.

[3] Bai M, Xin J C, Wang G R, Zhang L M, Zimmermann R, Yuan Y, Wu X D. Discovering the k representative skyline over a sliding window. *IEEE Transactions on Knowledge and Data Engineering*, 2016, 28(8): 2041-2056.

[4] Yu A, Agarwal P K, Yang J. Processing a large number of continuous preference top-k queries. In *Proc. ACM SIGMOD International Conference on Management of Data*, June 2012, pp.397-408.

[5] Shen Z T, Cheema M A, Lin X M, Zhang W J, Wang H X. Efficiently monitoring top-k pairs over sliding windows. In *Proc. the 28th International Conference on Data Engineering*, April 2012, pp.798-809.

[6] Yang X C, Qiu T, Wang B, Zheng B H, Wang Y S, Li C. Negative factor: Improving regular-expression matching in strings. *ACM Transactions on Database Systems*, 2016, 40(4): 25.

[7] Yang X C, Liu H L, Wang B. ALAE: Accelerating local alignment with affine gap exactly in biosequence databases. *Proceedings of the VLDB Endowment*, 2012, 5(11): 1507-1518.

[8] Yang X C, Wang B, Qiu T, Wang Y S, Li C. Improving regular-expression matching on strings using negative factors. In *Proc. ACM SIGMOD International Conference on Management of Data*, June 2013, pp.361-372.

[9] Xie X H, Yang X C, Wang J Y, Wang B, Li C. Efficient direct search on compressed genomic data. In *Proc. the 29th International Conference on Data Engineering*, April 2013, pp.961-972.

[10] Yi K, Yu H, Yang J, Xia G Q, Chen Y G. Efficient maintenance of materialized top-k views. In *Proc. the 19th International Conference on Data Engineering*, March 2003, pp.189-200.

[11] Pripužić K, Žarko I P, Aberer K. Time- and space-efficient sliding window top-k query processing. *ACM Transactions on Database Systems*, 2015, 40(1): Article No. 1.

[12] Alon N, Matias Y, Szegedy M. The space complexity of approximating the frequency moments. In *Proc. the 28th Annual ACM Symposium on the Theory of Computing*, May 1996, pp.20-29.

[13] Datar M, Gionis A, Indyk P, Motwani R. Maintaining stream statistics over sliding windows. In *Proc. the 13th Annual ACM SIAM Symposium on Discrete Algorithms*, January 2002, pp.635-644.

[14] Harvey N J A, Nelson J, Onak K. Sketching and streaming entropy via approximation theory. In *Proc. the 49th Annual IEEE Symposium on Foundations of Computer Science*, Oct. 2008, pp.489-498.

[15] Tong Y X, Zhang X F, Chen L. Tracking frequent items over distributed probabilistic data. *World Wide Web*, 2016, 19(4): 579-604.

[16] Charikar M, Chen K, Farach-Colton M. Finding frequent items in data streams. In *Proc. the 29th International Conference on Automata, Languages and Programming*, July 2002, pp.693-703.

[17] Ganguly S, Majumder A. Cr-precis: A deterministic summary structure for update data streams. In *Proc. the 1st Int. Symp. Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, April 2007, pp.48-59.

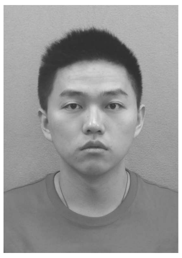
[18] Shrivastava N, Buragohain C, Agrawal D, Suri S. Medians and beyond: New aggregation techniques for sensor networks. In *Proc. the 2nd International Conference on Embedded Networked Sensor Systems*, November 2004, pp.239-249.

[19] Cormode G, Muthukrishnan S. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 2005, 55(1): 58-75.

[20] DeGroot M H, Schervish M J. Probability and Statistics (4th edition). China Machine Press, 2012.

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

Bin Wang received his Ph.D. degree in computer science from Northeastern University, Shenyang, in 2008. He is currently an associate professor of College of Computer Science and Engineering and the Computer System Institute at Northeastern University, Shenyang. His research interests include design and analysis of algorithms, queries processing over streaming data, and distributed systems. He is a member of CCF.



Shi-Ying Luo is a graduate student at Northeastern University, Shenyang. His research interest primarily includes spatial database, social networks, and big data management.

Xiao-Chun Yang received her Ph.D. degree in computer science from Northeastern University, Shenyang, in 2001. She is a professor in the College of Computer Science and Engineering at Northeastern University, Shenyang. Her research interests include data quality and data privacy. She is a senior member of CCF, IEEE, and a member of ACM.

Guo-Ren Wang received his B.S., M.S. and Ph.D. degrees in computer science from Northeastern University, Shenyang, in 1988, 1991 and 1996, respectively. Currently, he is a professor in the College of Computer Science and Engineering, Northeastern University, Shenyang. His research interests are XML data management, query processing and optimization, bioinformatics, high-dimensional indexing, parallel database systems, and P2P data management.