Dynamic Multiple Continuous Top-k Queries Over Streaming Data

BaoJie Jing[1] Xin Zhang[2] Rui Zhu[1,3] Wenju Li[2] Tao Qiu[1] Hong Jiang[4] Xiaochun Yang[5]

[1] Shenyang Aerospace University, China
[2] Shenyang Aircraft Corporation, China
[3] Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, JiLin University, Changchun, China
[4] Shenyang university of technology, China
[5] Northeastern University, China
jingbaojie@stu.sau.edu.cn, 13514284007@163.com, zhurui@sau.edu.cn, 8899@163.com, qiutao@sau.edu.cn, jianghong@sut.edu.cn, yangxc@mail.neu.edu.cn

Abstract. This paper studies the problem of DMCTQ (Dynamic Multiple Continuous Top-k Queries) over streaming data, a fundamental problem in the domain of streaming data management. Let $\mathcal{S}$ be the set of streaming data, and $\mathcal{Q}$ be the query workload. It contains a set of queries with different query parameters. Each query $q_i(n, s, k, F)$ in $\mathcal{Q}$ monitors objects generated in the last $q_i(n)$ time units, and returns $q_i(k)$ objects with the highest scores to the system whenever $q_i(s)$ time units pass. Some existing methods support DMCTQ but they incur high computational costs, especially when queries within $\mathcal{Q}$ are allowed to be changed.
In this paper, we propose a novel framework named Skyline-based Top-k Query Framework (STKQF for short) over streaming data. It is based on the following observation. For two queries q_1 and q_2 in the query workload $\mathcal{Q}$, if $q_1(n) > q_2(n)$ and $q_1(k) > q_2(k)$, the meaningful objects of q_2 must be meaningful objects of q_1. Therefore, we can support DMCTQ based on partial queries. Based on this, we propose a grid based index to maintain these queries, form a small number of virtual queries, support DMCTQ via monitoring meaningful objects under these virtual queries. Finally, we conduct extensive performance studies on large real and synthetic datasets, which demonstrate that our new framework could efficiently support DMCTQ over streaming data.

Keywords: Streaming Data · Dynamic Multiple Continuous Top-k Queries · PH-Tree

1 Introduction

Dynamic multiple continuous top-k queries(DMCTQ for short) over data streams is an important problem in the domain of streaming data management [6, 9, 11,

12, 1, 2]. It finds applications in various domains, including stock recommendation, detecting abnormal vehicles in traffic systems, DDoS attack detection, and more.

Let $\mathcal{S}$ be the set of streaming data, and $\mathcal{Q}$ be a workload consisting of a set of dynamic continuous top-k queries. Each query in $\mathcal{Q}$ is denoted as $q_i(n, s, k, F)$, which monitors objects generated in the last $q_i(n)$ time units, and returns $q_i(k)$ objects with the highest scores to the system whenever $q_i(s)$ time units pass. Here, $q_i(F)$ refers to the preference function, where all queries within $\mathcal{Q}$ have the same preference function. Compared with other efforts, we consider a more general case, where queries within $\mathcal{Q}$ are allowed to be changed, i.e., new queries could be entered into $\mathcal{Q}$, invalid queries could be deleted from $\mathcal{Q}$. We call this type of query as DMCTQ (short for Dynamic Multiple Continuous Top-k Queries).

We use sliding window to model streaming data. The window can be either time-based or count-based. In this paper, we only consider time-based windows [3–5, 8]. Additionally, we regard n_{max} as the query window length, with n_{max} being the maximal n parameter among all queries in $\mathcal{Q}$. Other queries in $\mathcal{Q}$ can be considered as monitoring top-k objects within sub-windows of the whole window W_q.

Many studies have focused on supporting continuous top-k queries (CTOPK for short) [6, 9, 11] over data stream. Their key idea is to use all meaningful (or partial meaningful) objects within the window as candidate objects, use them to support top-k searches. Here, given two objects o_1 and o_2 within the query window, if $F(o_1) \geq F(o_2)$ and o_1 arrives later than o_2, we say that o_1 q-dominates o_2. If o_1 is q-dominated by fewer than k objects, o_1 is a meaningful object. However, when extending them to support DMCTQ, they have to maintain a group of candidate sets, with each set corresponding to ONE query. This may incur highly running cost, especially when the scale of $\mathcal{Q}$ is large.

The algorithm named MTopList[7] is a representative one that can support DMCTQ, which predicts query results for each query in future windows. However, as stated in [10], this algorithm is sensitive to query parameters $q(s)$, $q(n)$ and $q(k)$. A novel index named PH-Tree can be applied to support DMCTQ. Compared with MTopList, it is efficient when $\mathcal{Q}$ is not allowed to be changed. However, this algorithm cannot support DMCTQ when the query workload is allowed to be changed as it has to frequently scan the query window. An efficient algorithm that has the ability of handling newly arrived/expired queries is desired.

In this paper, we propose a novel framework named STKQF (short for Skyline-based Top-k Query Framework) to support dynamic multiple continuous top-k queries(DMCTQ for short). It uses the following observation to support DMCTQ. That is, given two queries q_1 and q_2 within the query workload $\mathcal{Q}$, if $q_1(n) > q_2(n)$ and $q_1(k) > q_2(k)$, we say q_1 q-dominates q_2. In particular, if a query q within $\mathcal{Q}$ is not q-dominated by any queries within $\mathcal{Q}$, we say it is a skyline query.

An useful property is if q_1 q-dominates q_2, meaningful objects under q_1 must be meaningful under q_2. Therefore, when supporting DMCTQ, we can form another query set $\mathcal{Q}_S$, which contains all skyline queries within $\mathcal{Q}$. Additionally, we can form a PH-Tree $\mathcal{I}$ based $\mathcal{Q}_S$, use candidate objects maintained by $\mathcal{I}$ to

support DMCTQ. When a new query q_{new} is submitted to the system, if it is not a skyline query, we could ignore it. Otherwise, we should update PH-Tree based q_{new}. In other words, only when new skyline queries are submitted to system, we should update PH-Tree so as to cater the changing of $\mathcal{Q}$. It helps us reduce the impact of query insertion/deletion to the algorithm performance. However, we need to address the following challenges.

Firstly, when skyline queries are submitted to the system, we also need to spend highly running cost in scanning the query window. Secondly, the scale of skyline queries may be large, while queries within $\mathcal{Q}$ keep change, it is difficult to monitor skyline queries via as low as running cost. To deal with the above challenges, the contributions of this paper are as follows.

- We propose a novel query named dynamic multiple continuous top-k queries (DMCTQ for short) over streaming data. To the best of our knowledge, this is the first work to address the problem of DMCTQ over streaming data.
- We propose a novel index named SQ-grid $\mathcal{G}$ to maintain queries within the query workload $\mathcal{Q}$. Based on $\mathcal{G}$, we further generate a group of virtual query points, and we could guarantee that the number of virtual query points is bounded by a constant. These virtual query points could support DMCTQ alternatively.
- We propose the algorithm named PHSSC (short for PH-Tree-based Searching Scope Calculation) to evaluate whether we should scan the window to search meaningful objects under new queries. It uses a group of methods to avoid scanning the window, which help us avoid unnecessary running cost as much as possible. Even if needed, this algorithm helps us fix the scanning scope.

The rest of this paper is as follows. Section 2 reviews the related work and proposes the problem definition. Section 3 explains the framework STKQF. Section 4 evaluates the performance of STKQF. Section 5 concludes this paper.

2 Preliminary

In this section, we only review some important existing algorithms about CTopk [6, 9, 11, 12] and efforts that are relevant to DMCTQ over streaming data. Next, we explain the problem definition. Last of this section, we introduce the index named PH-Tree.

2.1 Related Work

One classic algorithm uses domination relationships among objects to support CTopk. However, as stated in [11], its performance is sensitive to data distribution. In the worst cases, its incremental maintenance cost is linear to the scale of objects in the window. The algorithm minTopK[9] improves this algorithm by using a natural property of the sliding window, i.e., the parameter s. If $s >> k$, its running cost is low. As a contrast, its overall running (or space) cost remains

high. SAP [11] improves minTopK via using the *partition* technique. Theoretical analysis shows that SAP is both efficient and stable. However, when extending it to support DMCTQ, it has to handle a group of candidate sets, leading higher running costs.

The algorithm named MTopList[7] can support MCTOPK via predicting query results for each query in the future windows. However, the performance of MTopList is sensitive to parameters n, k, and s of each query within the query workload set $\mathcal{Q}$. If some queries have small $q(s)$ parameters but large $q(n)$ and $q(k)$ parameters, its running cost is high. Zhu et al. propose a novel index PH-Tree to support multiple continuous top-k queries. As will be reviewed in Section 2.3, it only can support DMCTQ when query workload is not changed.

2.2 Problem Definition

In this section, we mainly introduce the concepts of multi-top-k-query and dynamic multi-top-k-query over data stream.

A continuous top-k query q, represented by the tuple $q(n, s, k, F)$, monitors objects within the query window W. It returns k objects with the highest scores to the system whenever s time units pass. Here, $q(n)$ represents the size of the query window, which monitors objects that are generated within the last $q(n)$ time units. $q(F)$ is the preference function. The query workload $\mathcal{Q}$ consists of a set of top-k queries, expressed as $\{q_1(n, s, k, F), q_2(n, s, k, F), \cdots, q_m(n, s, k, F)\}$. In $\mathcal{Q}$, each query $q_i(n, s, k, F)$ monitors objects generated within the last $q_i(n)$ time units. When $q_i(s)$ time units pass, it returns $q_i(k)$ objects with the highest scores to the system.

As shown in Fig. 1(a)-(b), the query workload $\mathcal{Q}$ contains two queries,i.e., $\mathcal{Q} = \{q_1(12, 2, 4, F), q_2(6, 2, 2, F)\}$. At the moment t_0, query results under q_1 and q_2 are $\{99, 98, 95, 93\}$ and $\{93, 88\}$, respectively. At the moment t_2, query results under q_1 and q_2 are updated to $\{98, 95, 93, 88\}$ and $\{88, 78\}$. In the following, we will formally explain the problem definition.

Definition 1. Dynamic Continuous Top-k Multi-Query. A dynamic query workload $\mathcal{Q}$ consists of a set of top-k queries $\{q_1(n, s, k, F), q_2(n, s, k, F), \cdots, q_m(n, s, k, F)\}$. Elements within $\mathcal{Q}$ keep change with the slide of the window,i.e., new queries are allowed to be entered into $\mathcal{Q}$, while invalid queries are deleted from $\mathcal{Q}$. Each query $q_i(n, s, k, F)$ monitors objects generated within the last $q_i(n)$ time units, returns $q_i(k)$ objects to the system whenever $q_i(n)$ time units pass.

As shown in Fig. 1(c), a new query $q_3(8, 2, 1, F)$ is inserted into the query workload $\mathcal{Q}$ at the moment t_3. At the moment t_4, we should retrieve query results for queries q_1, q_2, and q_3, which are $\{95, 93, 90, 88\}$, $\{90, 78\}$, and $\{90\}$, respectively.

2.3 PH-Tree

The PH-Tree is a partition-based binary tree index structure, designed to support dynamic multiple top-k queries over data stream. Its function is to select select

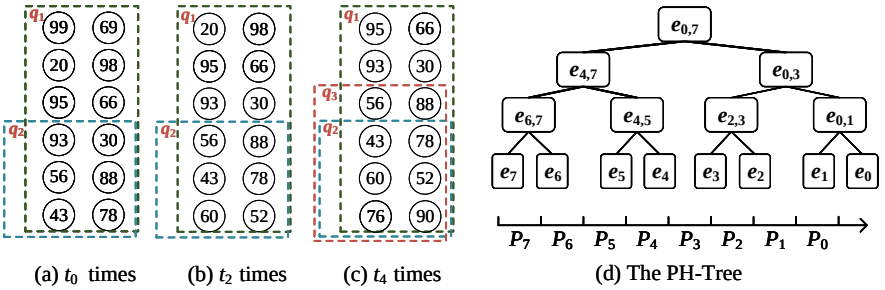

Fig. 1. Running Example of the SQ-Grid Structure.

high-quality candidate objects within the window. Specifically, let n_{min} and n_{max} be the minimal and maximal n parameters among all queries within the query workload $\mathcal{Q}$. PH-Tree firstly forms a partition $\mathcal{P}$ that partitions the whole window into a group partitions, denoted as $\{P_0, P_1, \ldots, P_{m-1}\}$. Each partition P_i corresponds a leaf node e_i within the PH-Tree $\mathcal{I}$, and each interval node $e(i,j)$ corresponds to a group of partitions from P_i to P_j, while the root node $e_{0,(m-1)}$ corresponds to all partitions from P_0 to P_{m-1}. Next, PH-Tree selects a group of objects with the highest scores in each partition as candidate objects, maintain them in the PH-Tree, and finally support query processing via searching on the PH-Tree.

Consider the example shown in Fig. 1(d). We use a PH-Tree $\mathcal{I}$ of height 4 to manage partitions in P. This tree contains 8 leaf nodes, with leaf node e_0 corresponding to partition P_0, e_1 to P_1, and so on. The interval node $e_{1,2}$ corresponds to partitions P_1 and P_2. We say the scope of $e_{1,2}$ (or $P_{1,2}$) is $\{P_1, P_2\}$.

3 The Framework STKQF

In this section, we propose a novel framework named STKQF(Skyline-based Top-k Query Framework) to support DMCTQ. Firstly, we explain two useful observations, where we use them to avoid scanning the window as much as possible when new queries are submitted to the system. Next, we propose a grid-based index named SQ-Grid to organize query points within the query workload $\mathcal{Q}$, further form a group of virtual queries based on $\mathcal{Q}$, and finally support DMCTQ via monitoring query results of these virtual queries. Last of this section is incremental algorithms.

3.1 Two Useful Observations

When a new query q arrives, our main goal is to avoid scanning the window, or minimize the scanning scope based on queries within the query workload. We achieve this goal based on the following observations.

Before introducing these observations, we first explain the concepts of q-dominates. Formally, given two queries $q_1(n,s,k)$ and $q_2(n,s,k)$ within the query

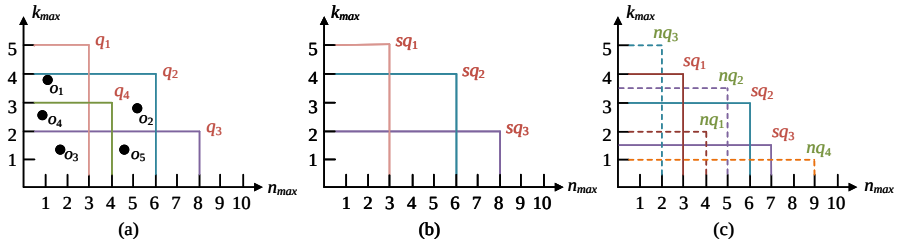

Fig. 2. The Running example about Two Observations.

workload $\mathcal{Q}$, if $q_1(n) \geq q_2(n)$ and $q_1(k) \geq q_2(k)$, we say q_1 q-dominates q_2. In particular, if q_1 is not q-dominatedby any query within $\mathcal{Q}$, q_1 is regarded as a *skyline query*. All skyline queries within $\mathcal{Q}$ form the skyline query set, i.e., called $\mathcal{Q}_S$.

Take an example in Fig. 2(a). There are 4 queries $q_1(3,2,5)$, $q_2(6,2,4)$, $q_3(8,2,2)$ and $q_4(4,2,3)$ within the query workload $\mathcal{Q}$. Since $q_2(6) > q_4(4)$ and $q_2(4) > q_4(3)$, q_2 q-dominates q_4. Moreover, as q_2 is not q-dominatedby any query in the query set $\mathcal{Q}$, q_2 is a skyline query. Similarly, q_1 and q_3 are also skyline queries. Therefore, q_1, q_2 and q_3 form the skyline query set, i.e., $\mathcal{Q}_S = \{q_1, q_2, q_3\}$.

observation 1. *Given two queries $q_1(n,s,k)$ and $q_2(n,s,k)$ within the query workload $\mathcal{Q}$, if q_1 q-dominates q_2, meaningful objects under q_2 must be meaningful objects, i.e., $C(q_2) \subseteq C(q_1)$.*

Observation 1 implies, given two queries $q_1(n,s,k)$ and $q_2(n,s,k)$, if q_1 q-dominates q_2, each meaningful object under q_2 must be a meaningful object under q_1. In this way, we can support DMCTQ via considering q_1. Here, we call an object o as a meaningful object under q_1 if fewer than $q_1(k)$ objects within the window have scores larger than $F(o)$ (also having arrival order larger than o). As shown in Fig. 2(a), $q_2(6,2,4)$ q-dominates $q_4(4,2,3)$, meaningful objects under q_2 are $\{o_1, o_2, o_3, o_4, o_5\}$, while meaningful objects under q_4 are $\{o_3, o_4\}$, i.e., $C(q_2)$ is contained in $C(q_1)$. Therefore, we can answer q_2 and q_4 via monitoring meaningful objects under $q_2(6,2,4)$.

Theorem 1. *Let $\mathcal{Q}$ be the query workload. We can use meaningful objects under all skyline queries to support DMCTQ.*

Proof. It is equivalent to prove that if an object o is a meaningful object under a non-skyline query q, it must be a meaningful object under a skyline query. Let q_s be a skyline query that q-dominates q. We have $q_s(k) \geq q(k)$ and $q_s(n) \geq q(n)$. Therefore, o must be within the query window under q_s. Also, there exists less than $q_s(k)$ objects having their scores larger than o, as well as having their arrived order larger than o. Therefore, o is a meaningful object under q_s, and we can use meaningful objects under all skyline queries to support DMCTQ.

observation 2. *Let $\mathcal{Q}_S$ be the skyline query set formed by skyline queries $\{sq_1, sq_2, \cdots, sq_{|S|}\}$. They are sorted in ascendant order by their $sq(n)$, i.e., for each sq_i, we have $sq_i(n) \leq sq_{i+1}(n)$. We can ensure that $sq_i(k) \geq sq_{i+1}(k)$.*

As shown in Fig. 2(b), the skyline query set is $\mathcal{Q}_S = \{sq_1(3, 2, 5), sq_2(6, 2, 4), sq_3(8, 2, 2)\}$. Since $sq_1(3) < sq_2(6) < sq_3(8)$, it follows that $sq_1(5) > sq_2(4) > sq_3(2)$. Based on the above facts, we can further use the following observation to answer DMCTQ when a new query q_{new} is submitted. It is based on the following four cases. Here, $M(sq_i)$ refers to the meaningful objects under sq_i, and $W(t)$ refers to the current time unit.

- Case I: If q_{new} is q-dominatedby element(s) within $\mathcal{Q}_S$, we could ignore it.
- Case II: Let we assume that q_{new} is a skyline query, and sq_i is the skyline query having the smallest $sq_i(k)$ satisfying $sq_i(k) > q_{new}(k)$. We could retrieve meaningful objects under $q_{new}(k)$ via considering $M(sq_i)$ or objects having arrived orders within $[W(t) - sq_i(n), W(t) - q_{new}(n)]$.
- Case III: Let we assume that q_{new} is a skyline query, and no skyline query within $\mathcal{Q}_S$ has $sq(k)$ larger than $q_{new}(k)$. We could retrieve meaningful objects under $q_{new}(k)$ via considering objects having arrived orders within $[W(t) - q_{new}(n), W(t)]$.
- Case IV: Let we assume that q_{new} is a skyline query, and no skyline query within $\mathcal{Q}_S$ has $sq(n)$ larger than $q_{new}(n)$. We could retrieve meaningful objects under $q_{new}(k)$ via $M(sq_{|S|})$ or objects having arrived orders within $[W(t) - n_{max}, W(t) - q_{new}(n)]$.

As shown in Fig. 2(c), the skyline query set is $\mathcal{Q}_S = \{sq_1(3, 2, 4), sq_2(6, 2, 3), sq_3(7, 2, 1.5)\}$. Under Case I, when a new query $nq_1(4, 2, 2)$ arrives, since $sq_2(n) > nq_1(n)$ and $sq_2(k) > nq_1(k)$, sq_2 q-dominates nq_1, so the query nq_1 can be ignored. Under Case II, when a new query $nq_2(5, 2, 3.5)$ arrives, it is a skyline query. In $\mathcal{Q}_S$, we find that $sq_1(k) > nq_2(k)$ and $sq_1(k)$ is the smallest value that is greater than $nq_2(k)$. Therefore, we only need to consider the meaningful objects within the range $[3, 5]$. Under Case III, when a new query $nq_3(2, 2, 5)$ arrives, it is a skyline query, and it satisfies $nq_3(k) > sq_1(k) > sq_2(k) > sq_3(k)$. Thus, we only need to consider the meaningful objects within the range $[0, 2]$. Under Case IV, when a new query $nq_4(9, 2, 1)$ arrives, it is also a skyline query, and it satisfies $nq_4(n) > sq_3(n) > sq_2(n) > sq_1(n)$. Therefore, we only need to consider the meaningful objects within the range $[7, 9]$.

Discussion. We understand that we can use skyline queries within the query workload $\mathcal{Q}$ to answer DMCTQ. Also, when new queries are submitted, we can evaluate whether we should scan the query window via q-domination relationships among queries within the query workload, avoid scanning the whole window as much as possible. However, as queries within $\mathcal{Q}$ may be frequently changed and the scale of skyline queries may be large, it is difficult to monitor skyline queries in real time. To address this issue, we propose a novel structure called the SQ-Grid to address this issue, in the following section.

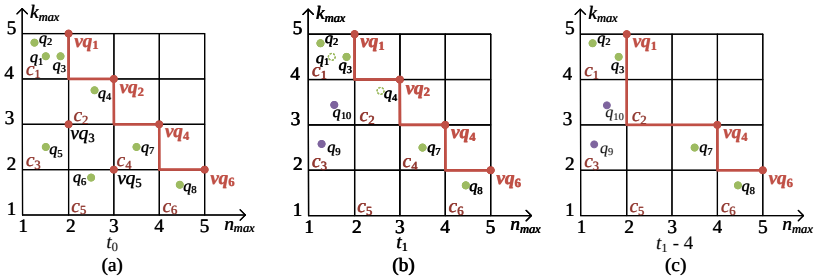

Fig. 3. Running Example of the SQ-Grid Structure.

3.2 The Index SQ-Grid

We are going to form a grid-based index named SQ-Grid to manage queries within $\mathcal{Q}$. Compared with monitoring skyline queries within $\mathcal{Q}$, we form a smaller number of queries, use them to support DMCTQ, alternatively. For simplicity, n_{max} and k_{max} refer to the maximal *allowed* query window length and maximal *allowed* k value among queries within $\mathcal{Q}$.

To be more specifically, SQ-Grid $\mathcal{G}$ is a 2-dimension-based grid, which maintains queries via considering sliding window length, as well as parameter k. Since these two dimensions have different value ranges, we map them into an unified standardized space. Once query points are mapped to $\mathcal{G}$, we generate a set of virtual queries, which are the coordinates of the upper-right corners of the non-empty grid cells. Given a non-empty cell c within the index $\mathcal{G}$, as the virtual query corresponding to c q-dominates all queries within c, we could use this for substituting queries within c. Accordingly, as stated in Theorem 2, we could use skyline virtual queries to support DMCTQ.

Theorem 2. *Let $\mathcal{Q}$ be the query workload. Queries within $\mathcal{Q}$ are maintained by the SQ-Grid $\mathcal{G}$, and $\mathcal{V}_S$ are the skyline virtual queries that are generated via $\mathcal{G}$. We can use meaningful objects under all skyline virtual queries to support DMCTQ.*

Proof. Let v be a virtual skyline point, c be the corresponding cell within $\mathcal{G}$, and sq be a skyline query within $\mathcal{Q}$. As v is the right-upper coordinate of c, it q-dominates sq. In other words, meaningful objects under v must be meaningful objects under sq. Therefore, we only need to consider v other than sq, and we can support DMCTQ via considering meaningful objects under these skyline virtual queries. The proof complete.

As shown in Fig. 3(a), taking c_1 as an example, queries q_1, q_2 and q_3 are contained in the cell c_1. We then form a virtual query point vq_1 based on c_1. Obviously, it q-dominates q_1, q_2 and q_3. Similarly, we can form other virtual query points based on other non-empty cells, which are vq_2, vq_3, vq_4, vq_5 and vq_6. Since vq_2 q-dominates vq_3, and vq_4 q-dominates vq_5, both vq_3 and vq_5 are not skyline virtual points. The skyline virtual query set is $\{vq_1, vq_2, vq_4, vq_6\}$.

3.3 The Incremental Maintenance Algorithm

Let $\mathcal{G}$ be the grid-based SQ-Grid, Q_v be the skyline virtual query set that are formed based on cells within $\mathcal{G}$. Similar to the logic discussed in [10], we use a PH-Tree-based index $\mathcal{I}$ to maintain candidate objects. To be more specially, PH-Tree firstly forms a partition $\mathcal{P}$ that partitions the whole window into a group partitions, denoted as $\{P_0, P_1, \cdots, P_{m-1}\}$. Next, we select a group of candidate objects based on virtual queries within Q_v.

As the window slides, some objects enter the window, while another group of objects expires from the window. We first update $\mathcal{I}$ based on the newly arrived/expired objects. Additionally, if new queries are submitted, we should evaluate whether we should search candidate objects for these new queries. If some queries are deleted from $\mathcal{Q}$, we evaluate whether we should remove some candidate objects from $\mathcal{I}$ so as to reduce candidate object maintenance cost. As the logic of maintaining PH-Tree is the same as that of [10], we skip the details for saving space.

In the following, we only explain algorithms when new queries are submitted to the system, or some queries are deleted from $\mathcal{Q}$.

Handling New Queries. When a new query q_{new} is submitted to the system, we first find the cell c within $\mathcal{G}$ that contains q_{new}. If some queries have (or one query has) been contained in c, we just insert it into c, and we need not to search meaningful objects for q_{new}. Otherwise(c is empty), we first insert it into c. Additionally, we evaluate whether it is a skyline virtual point v. If the answer is no, we also need not to search meaningful objects for q_{new}. If v is a skyline virtual point, we may have to scan the window, select candidate objects for v. In order to further avoid the scanning (or reduce the scanning scope), we use score (also arrived order) relationships among for further evaluation.

To be more specific, let we assume that the whole window be partitioned into $\{P_0, P_1, \cdots, P_m\}$, the virtual point v is located at the partition P_i. We first calculate the searching scope, i.e., $[P_u, P_i]$, following the logic discussed in Section 3.1. Next, we evaluate whether we could further reduce the scanning scope based on the PH-Tree, i.e., $\{C_u, C_{u+1}, \cdots, C_i\}$ via Theorem 3. For simipicty, we assume that $v(k)$ is the smallest k value among all virtual skyline queries having $v(k)$ larger than $v_{new}(k)$. The virtual skyline query v and v_{new} are located at partitions P_i and P_u, respectively. C_i refers to the candidate set corresponding to P_i.

Theorem 3. *Let $\theta_{v(k)}$ be the $v(k)$-th highest score among all candidate objects within partitions from P_s to P_m. If $\theta_{v(k)}$ is larger than min(C_i) ($s \leq i \leq e$), no object within P_i is a meaningful object under $v_{new}(k)$.*

Proof. Let o be an object within $P_i - C_i$. As $\theta_{v(k)} > \min(C_i)$, and $F(o) < \min(C_i)$, we ensure that $\theta_{v(k)}$ is larger than $F(o)$. As objects within partitions from P_s to P_m all have their arrived order larger than o, it is guaranteed that each object $o \in P_i - C_i$ is a meaningless object under $v_{new}(k)$.

Algorithm 1: The Incremental Maintenance Algorithm

Input: New query q_{new}, query set $\mathcal{Q}$, maximum window slides n_{max}
Output: Updated query set $\mathcal{Q}$, PH-Tree $\mathcal{I}$

```
if New query q_new arrives then
    Locate the cell c containing q_new;
    if c is non-empty then
        Insert q_new into cell c;
    else
        Insert q_new into empty cell c;
    Evaluate if q_new is a virtual skyline point vq;
    if q_new is not a virtual skyline point then
        No access to I;
    if q_new is a virtual skyline point then
        Compute search range [P_u, P_i] based on;
        Case I: If q_new is dominated by queries within Q, ignore it;
        Case II: If ∃sq_i with sq_i(k) > q_new(k), set range
          [W(t) − sq_i(n), W(t) − q_new(n)];
        Case III: If no sq(k) > q_new(k), set range [W(t) − q_new(n), W(t)];
        Case IV: If no sq(n) > q_new(n), set range
          [W(t) − n_max, W(t) − q_new(n)];
        Refine search range using PH-Tree I;
if Query q_old deleted from Q then
    Delete cell c containing q_old from Q;
    if After n_max slides, c becomes empty then
        Delete vq and update I;
return;
```

Theorem 3 implies we can use $\theta_{v(k)}$ as a threshold to evaluate whether we should scan partitions from P_u to P_i. In real implementation, we reversely access each partition $P_j (u \leq j \leq i)$. When evaluating P_u, we first calculate $\theta_{v(k)}$. If $\theta_{v(k)}$ is larger than $\min(C_j)$, we skip accessing P_u as no object within $P_u - C_u$ are meaningful objects under v_{new}. Otherwise, we apply the algorithm discussed in [11] to find all meaningful objects under v_{new}, insert them into the PH-Tree $\mathcal{I}$. We next evaluate P_{u-1}. We first update $\theta_{v(k)}$, and then repeat the above operations. From then on, we access each partition following the logic discussed before until accessing all partitions.

Handling Expired Queries. When a query q_{old} is deleted from the query workload $\mathcal{Q}$, we first delete it from the cell c that contains q_{old}. Compared with handling new queries, we do not update virtual skyline query set even some cells turn to empty immediately. The reason is when other new queries are submitted, if they are located in the cell c, we need not to re-form virtual query points, as well as search meaningful objects for them immediately. It also helps us reduce the scanning frequency.

In order to keep the scale of PH-Tree small, we evaluate whether we should delete some virtual skyline query points whenever the window slides n_{max} times. To be more specific, let we assume the current moment is t, if the last time we delete virtual skyline query points is $t - n_{max}$,we attempt to delete some invalid virtual skyline query points from the virtual skyline query point set. At that moment, we access each virtual skyline query point vq. If the corresponding cell c had turned to meaningless for a long time, i.e., c keeps empty before $t - n_{max}$, we delete vq. After deleting all these virtual skyline points, we select new virtual skyline points. Last of all, we update PH-Tree,i.e., remove meaningless objects from PH-Tree. As the logic is simple, we skip the details for saving space.

Taking Fig. 3(b) as an example. At time t_0, the query set is $\mathcal{Q} = \{q_1, q_2, q_3, q_4, q_7, q_8\}$. At time t_1, the query set updates to $\mathcal{Q} = \{q_2, q_3, q_7, q_8, q_9, q_{10}\}$. As shown in Fig. 3(c), when query q_4 is deleted, the cell c_2 becomes empty. At that moment, we do not immediately delete c_2, as well as the virtual skyline query point vq_2. However, if c_2 remains empty until time $t_1 - 4$, we must delete the virtual skyline query point vq_2.

Cost Analysis. When handling newly arrived objects or expired objects, the running cost of the proposed algorithm is as the same as that has been discussed in [10], which is $\mathcal{O}(\log k_{max})$ per object. When a query is submitted to the system, the running cost of inserting it into SQ-grid is bounded by $\mathcal{O}(1)$. Let the side length of each grid be ρ. The number of skyline virtual queries is bounded by $\frac{\sqrt{2}}{\rho}$. Therefore, when a new query q is submitted to the system, we have to form a new skyline virtual query vq in the worst case. The running cost of updating skyline virtual query set is bounded by $\mathcal{O}(\frac{1}{\rho}\log\frac{1}{\rho})$. If we need to scan the window based on q, the overall running cost is bounded by $\mathcal{O}(n_{max} \cdot \log k_{max})$. As we do not delete vq until the window slides n times, the amortized running cost is bounded by $\mathcal{O}(\log k_{max})$ per object.

4 Performance Evaluation

In this section, we will first provide a detailed explanation of the experimental design and setup. Then, through extensive experimental results, we will demonstrate the excellent performance of our proposed methods over streaming data.

4.1 Experimental Settings

Data sets. This experiment is based on two real-world datasets (Stock and Trip) and two synthetic datasets (Normal and Uniform). The stock dataset contains trading records of 2,300 stocks from the Shanghai and Shenzhen stock exchanges, totaling 1 G, covering a time span of 24 months. During the data cleaning process, we retained four attributes (Stock ID, transaction time, transaction volume v and price p) and sorted the cleaned records in ascending order based on the transaction time. The preference function F is defined as $v \times p$. The trip dataset includes 1.6 G of trip records collected in New York City over a span of 72 months. For the trip data, we retained four attributes: taxi ID, pick-up time,

Table 1. Parameter Settings

Parameter	Value
mean of n	100K 200K 500K **1M** 2M 5M 10M
mean of k	10 20 50 **100** 200 500 1000
mean of s	0.01% 0.05% 0.1% 0.5% **1%** 5% 10% ($\times n$)
$\|\mathcal{Q}\|$	10 20 50 **100** 200 500 1000
f	10 20 50 **100** 200 500 1000

drop-off time, and trip distance, and similarly, we sorted the cleaned records in ascending order based on the pick-up time. The preference function is defined as $dist - tp$. Additionally, both the normal and uniform datasets contain 1 GB of real numbers, with the data following a normal distribution and a uniform distribution, respectively.

Query Workload. The query workload $\mathcal{Q}$ consists of 100 different top-k queries. In this workload, the values of n, k and s follow a normal distribution, with their means specified in Table 2, and each parameter has a default standard deviation (SD) of 100.

Parameter Settings. For the query parameters n, s, k, $|\mathcal{Q}|$ and f, we evaluate the performance of different algorithms. Here, n represents the number of objects in the query window, s denotes the sliding rate of the window, $|\mathcal{Q}|$ represents the number of queries, and f represents the query update frequency. To evaluate the impact of n on algorithm performance, we designed seven groups of query workloads. For each group, parameters k and s follow a normal distribution, with means of 100 and $1\% \times n$, respectively. The parameter n also follows a normal distribution, with specific means provided in Table 2, ranging from 100K to 10M. When analyzing the impact of k on algorithm performance, we similarly constructed seven groups of query workloads. In each group, the parameters n and s follow a normal distribution, with means of 1M and $1\% \times n$, respectively, while k follows a normal distribution, with its specific means given in Table 2. To evaluate the impact of s on algorithm performance, we again designed seven groups of query workloads. In these workloads, the parameters n and k follow a normal distribution, with means of 1M and 100, respectively, while s also following a normal distribution, with its specific means referenced in Table 2. For evaluating the impact of f on algorithm performance, we similarly designed seven sets of query loads. In these loads, the parameters n, k and s all follow a normal distribution with means of 1M, 100 and $1\% \times n$, respectively, while the parameter f also follows a normal distribution with specific means referenced in Table 2.

Competitors. In addition to PH-Tree, we have implemented M-SAP, MTopList, MTopBand and PH-Base as competing algorithms. All algorithms were implemented in C++, and all experiments were conducted on a 6226R CPU with 256GB of memory, running Microsoft Windows 10.

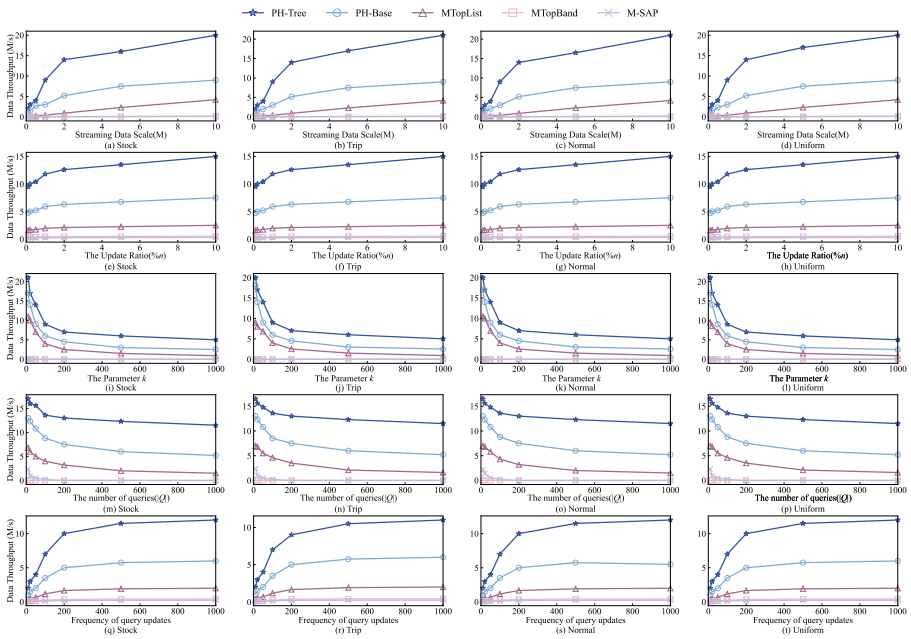

Fig. 4. Data throughput comparison of different algorithms.

4.2 Experimental Evaluation

Comparison of data throughput under different algorithms. First, we compared the performance of these five algorithms under different window sizes, as shown in Fig. 4(a)-(d). Our main findings are as follows: PH-Tree consistently demonstrated outstanding performance across all datasets, achieving significant improvements compared to all other algorithms. For example, on the Stock dataset, PH-Tree's average data throughput was approximately 2 times faster than PH-Base, 6 times faster than MTopList, 28 times faster than M-SAP, and 50 times faster than MTopBand. This is mainly attributed to PH-Tree's ability to adaptively select an appropriate scan window range based on query parameters and the distribution of object scores. This design minimizes the scanning range, thereby significantly reducing overall computation costs while maintaining result accuracy. In contrast, PH-Base retains more candidate objects within smaller partitions, may maintain predicted query result objects across multiple future query windows, M-SAP has to maintain a large candidate set to meet query requirements, and MTopBand incurs high overall computation costs due to maintaining a large number of redundant candidate objects.

Next, we evaluated the impact of different values of parameter s on algorithm performance, with other parameters set to their default values. Fig. 4(e)-(h) shows that as s increases, the data throughput of these algorithms changes

minimally. However, PH-Tree remains the best, with average data throughput approximately 2 times faster than PH-Base, 6 times faster than MTopList, 28 times faster than M-SAP, and 50 times faster than MTopBand.

Thirdly, we evaluated the impact of parameter k (ranging from 10 to 1000) on the algorithms' performance while keeping other parameters at their default values. As shown in Fig. 4(i)-(l), the trend was similar to the previous observations, with PH-Tree consistently performing the best. Although data throughput decreases significantly for all algorithms as k increases, PH-Tree maintains relatively high data throughput. This is because new candidate objects are not inserted into the PH-Tree until their partition is fully populated, which significantly reduces update frequency and, thus, operational costs.

We then compared the data throughput of PH-Tree, PH-Base, MTopList, M-SAP, and MTopBand under different $|\mathcal{Q}|$ values (ranging from 10 to 1000), with other parameters set to their default values. As shown in Fig. 4(m)-(p), PH-Tree performs the best. As $|\mathcal{Q}|$ increases, the performance of PH-Tree, PH-Base, and MTopList remains stable because they efficiently maintain candidate sets that support a large number of top-k queries. In contrast, M-SAP and MTopBand need to maintain more candidate sets, leading to a decrease in throughput.

Finally, we evaluated the query update frequency of each algorithm across the datasets, as shown in Fig. 4(q)-(t). It can be observed that PH-Tree consistently performed well across all datasets. The main reason is that PH-Tree has the smallest memory consumption in all datasets, thanks to the properties of its index structure, which only requires partial node maintenance from top to bottom. In contrast, MTopList needs to maintain more candidate objects because it must store all significant objects, resulting in higher computation costs during range queries.

Space cost comparison of different algorithms. We evaluated the space cost of each algorithm on the dataset, with the results shown in Fig. 5(a)-(d). It can be observed that MTopList requires maintaining more candidate objects than PH-Base and PH-Tree, as it must store all relevant objects. In contrast, PH-Base maintains fewer candidate objects, while PH-Tree maintains the fewest candidate objects.

Query time comparison of different algorithms. As shown in Fig. 5(e)-(h), PH-Tree exhibits the lowest running costs. This advantage stems from PH-Tree's organization of candidate objects using a max-heap, which allows for quick retrieval of query results by accessing a small number of high-level objects within the PH-Tree in most cases. Additionally, the candidate selection rules based on PH-Tree ensure that we can leverage candidates within the PH-Tree to support query processing in the majority of cases, thereby improving query efficiency.

Running time comparison of algorithms under different data dimensions. We will compare the running time of the PH-Tree, PH-Base, MTopList, M-SAP, and MTopBand algorithms under different dimensions d (ranging from 2 to 8), with other parameters set to default values. As shown in Table 3, the running time of PH-Tree is lower than the other four algorithms at each dimen-

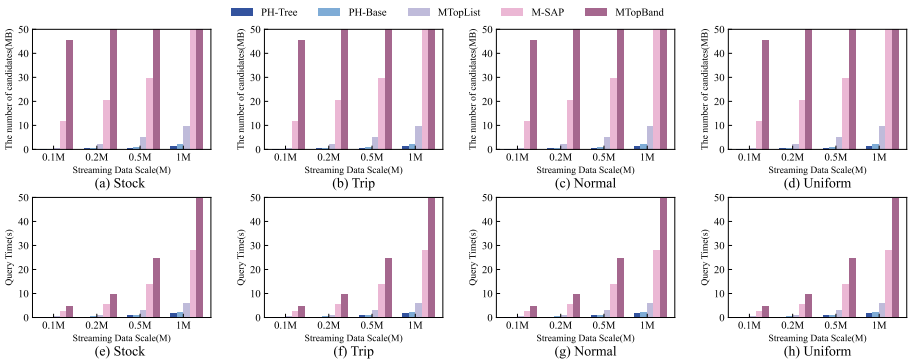

Fig. 5. Comparison of different algorithms under other metrics.

Table 2. Comparing time across different dimensions(UNIT:s)

Algorithm	Normal				Uniform			
	2	4	6	8	2	4	6	8
PH-Tree	1.783	8.279	25.398	35.983	1.932	8.970	27.520	38.989
PH-Base	4.418	13.130	54.905	74.082	5.782	17.183	71.856	96.953
MTopList	10.792	25.996	127.814	201.766	11.201	26.981	132.657	209.412
M-SAP	50.337	233.729	596.161	844.065	50.908	236.365	602.923	853.639
MTopBand	88.352	410.244	1258.532	21103.442	88.732	412.008	1263.944	21194.207

sion. This means that our proposed algorithm is stable and insensitive to data dimensions.

The experimental results show that PH-Tree excels in handling large-scale data and complex queries due to its adaptive partitioning and efficient candidate management, which enhance memory utilization and query efficiency. In contrast, M-SAP performs well on small datasets but struggles with larger queries, MTopBand ensures high accuracy but incurs high maintenance costs in multi-query scenarios, and MTopList provides comprehensive results at the expense of increased memory and computational resource consumption.

5 Conclusion

This paper introduces STKQF, a novel framework for supporting DMCTQ in a data stream environment. The framework employs an index structure called PH-Tree to maintain objects under different queries. Extensive experiments on various datasets confirm the outstanding performance of the proposed algorithms.

ACKNOWLEDGMENT

The work is partially supported by the National Natural Science Foundation of China (Nos. 62472293,U23A20309, U22A2025, 62232007,62102271), Rui Zhu and Xiaochun Yang are corresponding authors of this paper.

References

1. Chen, L., Gao, Y., Li, X., Jensen, C.S., Chen, G.: Efficient metric indexing for similarity search and similarity joins. IEEE Transactions on Knowledge and Data Engineering **29**(3), 556–571 (2017)
2. Chen, L., Zhong, Q., Xiao, X., Gao, Y., Jin, P., Jensen, C.S.: Price-and-time-aware dynamic ridesharing. In: 2018 IEEE 34th international conference on data engineering (ICDE). pp. 1061–1072. IEEE (2018)
3. Lin, Y., Lee, B.S., Lustgarten, D.: Continuous detection of abnormal heartbeats from ecg using online outlier detection. In: Information Management and Big Data: 5th International Conference, SIMBig 2018, Lima, Peru, September 3–5, 2018, Proceedings 5. pp. 349–366. Springer (2019)
4. Ma, Y., Zhao, X., Zhang, C., Zhang, J., Qin, X.: Outlier detection from multiple data sources. Information Sciences **580**, 819–837 (2021)
5. Mirzaie, S., Bushehrian, O.: A new outlier detection method for anomaly detection in iot-enabled distribution networks. Adhoc & Sensor Wireless Networks **55** (2023)
6. Mouratidis, K., Bakiras, S., Papadias, D.: Continuous monitoring of top-k queries over sliding windows. In: Proceedings of the 2006 ACM SIGMOD international conference on Management of data. pp. 635–646 (2006)
7. Shastri, A., Di, Y., Rundensteiner, E.A., Ward, M.O.: Mtops: scalable processing of continuous top-k multi-query workloads. In: Proceedings of the 20th ACM international conference on Information and knowledge management. pp. 1107–1116 (2011)
8. Toliopoulos, T., Gounaris, A.: Multi-parameter streaming outlier detection. In: IEEE/WIC/ACM International Conference on Web Intelligence. pp. 208–216 (2019)
9. Yang, D., Shastri, A., Rundensteiner, E.A., Ward, M.O.: An optimal strategy for monitoring top-k queries in streaming windows. In: Proceedings of the 14th International Conference on Extending Database Technology. pp. 57–68 (2011)
10. Zhu, R., Jia, Y., Yang, X., Zheng, B., Wang, B., Zong, C.: Multiple continuous top-k queries over data stream. In: 2024 IEEE 40th International Conference on Data Engineering (ICDE). pp. 1575–1588. IEEE (2024)
11. Zhu, R., Wang, B., Yang, X., Zheng, B., Wang, G.: Sap: Improving continuous top-k queries over streaming data. IEEE Transactions on Knowledge and Data Engineering **29**(6), 1310–1328 (2017)
12. Zhu, R., Wang, B., Yang, X., Zheng, B., Wang, G.: Sap: Improving continuous top-k queries over streaming data. In: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018. pp. 1819–1820. IEEE Computer Society (2018)