Efficient Processing of Skyline Group Queries over a Data Stream

Xi Guo, Hailing Li, Aziguli Wulamu*, Yonghong Xie, and Yajing Fu

Abstract: In this paper, we study the skyline group problem over a data stream. An object can dominate another object if it is not worse than the other object on all attributes and is better than the other object on at least one attribute. If an object cannot be dominated by any other object, it is a skyline object. The skyline group problem involves finding \( k \)-item groups that cannot be dominated by any other \( k \)-item group. Existing algorithms designed to find skyline groups can only process static data. However, data changes as a stream with time in many applications, and algorithms should be designed to support skyline group queries on dynamic data. In this paper, we propose new algorithms to find skyline groups over a data stream. We use data structures, namely a hash table, dominance graph, and matrix, to store dominance information and update results incrementally. We conduct experiments on synthetic datasets to evaluate the performance of the proposed algorithms. The experimental results show that our algorithms can efficiently find skyline groups over a data stream.

Key words: skyline; skyline group; data streams; query processing

1 Introduction

Given objects with multiple attributes, a skyline query finds objects not dominated by any other object. An object can dominate another object if it is not worse than the other object on all attributes and is better than the other object on at least one attribute. Skyline queries are widely used in many fields, for example, city navigation applications and user preference queries\(^{[1]}\). Consider a scenario in which a tourist going on holiday wants to find a hotel that is both cheap and close to the beach. In this case each hotel has two attributes, namely price and distance to the beach, as shown in Fig. 1. The \( X \)-axis represents distance to beach and the \( Y \)-axis represents cost of staying in a hotel. A hotel dominates another hotel if it is cheaper and closer to the beach. A skyline query returned seven hotels lying on the broken line. Each returned hotel could not be dominated by any other hotel. These hotels are considered skyline objects and are recommended to the tourist.

The skyline group problem\(^{[2,3]}\) is to find groups of \( k \)-items, that is, groups consisting of \( k \) objects, not dominated by any other group. Over a data stream, objects enter and expire with time. In this paper, we analyze the skyline group problem over a data stream.
stream and design algorithms to solve the problem. For example, Fig. 2 shows a stream of objects having two attributes. Three objects A, B, and C exist at moment 1 s. At moment 2 s, an object D is added. Therefore, the group of active objects is now \(\{A, B, C, D\}\). At moment 3 s, object A expires and the group of active objects changes to \(\{B, C, D\}\).

A two-item group consists of two currently active objects. For example, at 1 s, the following two-item groups exist: \(\{AB, AC, BC\}\). The attribute value of a group is the sum of attribute values of its group members. For example, the attribute values of group AB is \((3, 7)\). Given that new objects enter and old objects expire at each moment, the groups change with time. For example, at 2 s, three new groups \(\{AD, BD, CD\}\) are formed owing to the entry of object D. At 3 s, three groups \(\{AB, AC, AD\}\) are removed because of the expiry of object A.

Let us consider the skyline groups at each moment. (In this paper, we consider a smaller value is better.) At 1 s, the skyline groups are \(\{AB, AC\}\) because BC can be dominated by AC, while AB and AC cannot be dominated by other groups. At 2 s, the skyline groups change to \(\{AB, AC, CD\}\) owing to the entry of object D. At 3 s, the skyline groups change to \(\{BC, BD, CD\}\) because of the removal of object A. Skyline groups change with time. Our problem is to find and update skyline groups with time over a data stream.

The skyline problem has been studied extensively recent years\(^{[3–6]}\). However, studies on skyline groups are limited. A few researchers\(^{[2, 3]}\) have proposed algorithms to find skyline groups from a static dataset. In this study, we attempt to find skyline groups over a data stream, which to our knowledge is the first such attempt. Studies have attempted to find skyline objects from data streams (for example, Ref. \(^{[7]}\)). However, these studies have focused on individual objects, whereas our paper focuses on object groups. We cannot answer skyline group queries over a data stream by simply slightly modifying the existing algorithms.

The naïve algorithm to find and update the skyline groups over a data stream uses the algorithms proposed in Refs. \(^{[2, 3]}\) at every moment. However, it is not sufficiently efficient. A few skyline groups do not change at every moment. For example, in Fig. 2, groups \(\{AB, AC\}\) are skyline groups at 1 s as well as at 2 s. In this paper, we propose an efficient algorithm to find skyline groups incrementally based on the algorithms proposed in Refs. \(^{[2, 3]}\). The basic idea is to store dominance information that could be reused. We use a hash table to store candidate objects that can be used to construct skyline groups. We use a matrix to store the skyline sub groups. In addition, we propose the use of a dominance graph to store candidate objects. The use of a dominance graph can eliminate the dominance checks required for determining the candidate objects. Our experiments show that the proposed algorithms can efficiently find skyline groups over a data stream.

We summarize our contributions below.

- We propose the problem of finding and updating \(k\)-item skyline groups over a data stream.
- We design efficient algorithms to update skyline groups incrementally by reusing dominance information.
- We design an algorithm to store reusable candidate objects using a dominance graph.
- We conduct extensive experiments to evaluate the proposed algorithms.

The remainder of this paper is organized as follows. Section 2 reviews previous work related to our problem. Section 3 reviews the skyline group problem and defines our problem formally. Section 4 proposes the algorithms to find and update skyline groups over a data stream. Section 5 reports the experimental results and Section 6 summarizes the findings of our work.

### 2 Related Work

The skyline query problem has been studied in recent years\(^{[4, 5, 8–13]}\). Skyline objects can not be dominated by any other object. An object dominates another object if it is not worse than the other object on all attributes and is better than the other object on at least one attribute. Skyline query is an important tool for data mining and data analysis. It is mainly used in scenarios requiring
optimization of multiple attributes.

With the development of information technology, data is continuously generated at a high speed. Data streams are encountered in telecommunication systems, financial trading systems, real-time sensor signal analysis systems, and others \cite{14, 15}. A data stream is a data sequence \( S \) comprising data records. These data records arrive continuously and change dynamically with time. A query on data streams is different from traditional queries on static data because it is a continuous computation \cite{16}. Various categories of data streams include continuous, real-time arrival, and one-time processing. Considering the characteristics of data streams, it is not possible to process continuous high-speed data streams using limited computation resources. Many data stream models have been proposed in recent years. The most widely used model is the sliding window model. This model has two endpoints, namely the start and end positions of the window. If the window size is \( W \), a data record is active during the time interval \([t, t + W]\), where \( t \) is the arrival time of the data record.

A skyline query on a data stream involves finding skyline objects when objects are added and removed with time. Such a skyline query is a continuous query. Query results are returned in the form of a data stream \cite{17, 18}.

3 Preliminaries

3.1 Processing skyline groups in static data

Given a \( d \)-dimensional data set \( S = \{p_1, p_2, \ldots, p_n\} \), we can construct \( k \)-item groups using the objects in \( S \), that is, each group consists of \( k \) objects from \( S \). The attribute value of a group is the sum of the attribute values of its members. References \cite{2, 3} define the dominance relationship between groups.

Definition 1 A \( k \)-item group dominates another group if it is not worse than the other group on all attributes and it is better than the other group on at least one attribute.

The skyline group is defined according to the dominance relationship.

Definition 2 A \( k \)-item group is on the skyline, that is, a skyline group, if it cannot be dominated by any other \( k \)-item group.

A \( k \)-item skyline group query involves finding the \( k \)-item groups that are on the skyline. For example, as can be seen in Table 1, there are five objects A–E. Each object has two attributes \( d_1 \) and \( d_2 \). Let us consider the two-item groups. The attribute values of group AB are \((3, 3)\) that are the sums of A(2, 1) and B(1, 2). Similarly, the attribute values of group BD are \((4, 4)\). Group AB can dominate group BD since it is better than BD on all attributes. Compared with the other two-item groups, AB cannot be dominated. Therefore, group AB is on the skyline. References \cite{2, 3} propose algorithms to find all such skyline groups.

A naïve method to find skyline groups is enumerating and checking all \( k \)-item groups. The groups that cannot be dominated are the skyline groups. Enumerating all \( k \)-item groups is quite time consuming. Hence, a property was proposed to filter objects that cannot be used to construct a skyline group \cite{2, 3}. For an object to be included in a \( k \)-item skyline group, it must be dominated by fewer than \( k \) objects, that is, it should have fewer than \( k \) dominators. Such objects are the candidates for constructing a skyline group. For example, in Table 1, the five objects A, B, C, D, and E have 0, 0, 1, 2, and 2 dominators, respectively. Therefore, the candidates suitable for inclusion in two-item skyline groups are \{A, B, C\} because they have fewer than two dominators.

Furthermore, References \cite{2, 3} proposed a dynamic programming method to construct skyline groups gradually. If a group is a subgroup of a \( k \)-item skyline group, it should be a skyline group too. Based on this observation, we can construct skyline groups \( \text{GSky}_k^m \) by considering sub-problems \( \text{GSky}_{k-1}^{m-1} \) and \( \text{GSky}_{k-1}^{m-1} \). The symbol \( \text{GSky}_k^m \) denotes \( k \)-item skyline groups constructed from \( m \) objects. When checking object \( p_m \) to find \( \text{GSky}_k^m \), we consider skyline groups \( \text{GSky}_{k-1}^{m-1} \) and skyline groups \( \text{sky}_{k-1}^{m-1} \) by adding \( p_m \) as candidates. The skyline groups \( \text{GSky}_k^m \) are selected from the candidates. Let us continue our example. Since we have three candidate objects \{A, B, C\}, we denote the problem as \( \text{GSky}_2^3 \), which can be divided into \( \text{GSky}_2^2 \) and \( \text{GSky}_1^2 \), as shown in Fig. 3. The results of \( \text{GSky}_2^2 \) and \( \text{GSky}_1^2 \) are \{AB\} and \{A, B\}, respectively. By combining C with each group in \( \text{GSky}_1^2 \), we construct

| Object | \( d_1 \) | \( d_2 \) | Dominator number |
|--------|--------|--------|-----------------|
| A      | 2      | 1      | 0               |
| B      | 1      | 2      | 0               |
| C      | 3      | 1      | 1               |
| D      | 3      | 2      | 2               |
| E      | 4      | 3      | 2               |

\begin{table}[h]
\centering
\begin{tabular}{|l|c|c|c|}
\hline
Object & \( d_1 \) & \( d_2 \) & Dominator number \\
\hline
A & 2 & 1 & 0 \\
B & 1 & 2 & 0 \\
C & 3 & 1 & 1 \\
D & 3 & 2 & 2 \\
E & 4 & 3 & 2 \\
\hline
\end{tabular}
\caption{Two-dimensional objects with dominator number.}
\end{table}
two candidate groups \{AC, BC\}. Finally, we find the skyline groups GSky$_2^3$ = \{AB, AC\} from \{AB, AC, BC\}.

### 3.2 Our problem

In a data stream, objects are added and removed with time. In this paper, we study the sliding window model of data streams.

**Definition 3** Given window size $W$, if an object $p$ arrives at moment $t$, it is active (or alive) during the time interval $[t, t + W]$.

In other words, object $p$ is added to the set at $t$ and is removed from the set at $t + W$.

**Problem 1** Find the $k$-item skyline groups consisting of active objects over a data stream.

Given that objects are added and removed with time, the $k$-item skyline groups should be updated when changes occur. However, Refs. [2, 3] do not cover the processing of skyline groups over data streams. To our knowledge, the problem considered herein is novel and cannot be solved directly by using the existing method\cite{2, 3} for static data.

### 4 Processing Skyline Groups over a Data Stream

We propose an algorithm to find $k$-item skyline groups over a data stream based on the method proposed in Refs. [2, 3], which was reviewed in Section 3.1. A naïve method would be to update the results at every moment an object is added or removed by using the method in Refs. [2, 3] directly. In other words, for every change event, we must recompute the candidate objects and then run the dynamic programming algorithm from scratch. However, the naïve method can be improved by saving the candidate objects’ computations and executing the dynamic programming algorithm incrementally.

#### 4.1 Computing candidate objects incrementally

For every change event, we need to compute candidate objects. We propose an efficient algorithm to compute them incrementally. For each active object $p$, we maintain (1) the number of dominators, denoted by $p$.num and (2) objects that could be dominated by $p$, denoted by $p$.dominatee. When an object is added or removed, we update $p$.num and $p$.dominatee of each object influenced by $p$. Objects having fewer than $k$ dominators are reported as candidates. Thus, we can save the dominance checks of active objects that come before. Next, we describe the process of updating $p$.num and $p$.dominatee when a new object is added and an old object expires.

When an object $p$ arrives at moment $t$, we compute its dominance relationships with the objects existing in $P$. A straightforward way is to check the dominance relationship between $p$ and every object $p_i \in P$. If the objects in $P$ are organized by a spatial index, for example, R-tree, we can retrieve the dominators of $p$ and the objects that are dominated by $p$ (i.e., dominatees) more quickly.

According to the dominance relationship, the set $P$ can be divided into three groups: (1) objects dominated by $p$, denoted as Dominatee$_p$, (2) objects that dominate $p$, denoted as Dominator$_p$, (3) and other objects $P – \text{Dominatee}_p – \text{Dominator}_p$. We process the three groups in different ways.

1. If an object $p_i \in \text{Dominatee}_p$, the number of dominators $p_i$.num should increase by one. The object $p_i$ should be added to the dominatee list $p$.dominatee of $p$.
2. The number of dominators $p$.num of $p$ should be the number of objects in the group Dominator$_p$. The object $p$ should be added to the dominatee list $p_i$.dominatee, where $p_i \in $ Dominator$_p$.
3. Because the objects in this group do not have a dominance relationship with $p$, we do nothing.

After updating the number of dominators for each current active object, we report objects having fewer than $k$ dominators as the candidates.

When an object $p$ expires, that is, $p$ exists more than $W$, it should be removed from $P$. The remove operation may cause the list of candidate objects to change because a few non-candidate objects may become
candidate objects. The removal of $p$ may decrease the number of dominators of a few objects. According to the dominance relationship with $p$, we process objects $p_i \in P$ in different ways.

1. If an object $p_i \in \text{Dominatee}_p$, the number of dominators $p_i \text{.num}$ should be decreased by one.
2. If $p$ is in the set of candidate objects, it should be removed.
3. Objects that share no dominance relationship with $p$ are not influenced by the removal of $p$.

Hence, we do nothing.

After updating the influenced objects, we report objects having fewer than $k$ dominators as the candidates.

We use the example in Fig. 2 to illustrate the algorithm for computing and updating candidate objects. There are three active objects at moment 1 s, as shown in Fig. 4. We compute and store two pieces of information for each object: the number of dominators of the object, $p_i \text{.num}$, and the number of objects dominated by it, $p_i \text{.dominante}$. At moment 1 s, the number of objects that can dominate A is 0 and the objects dominated by A is B. Similarly, we store information about B and C. The candidate objects are {A, B, C}.

At the next moment, 2 s, object D arrives. We determine the dominance relationships of D with other objects. Object D cannot dominate any other object. The number of dominator is one because object C can dominate object D. Simultaneously, D should be added to the dominatee list of C. The candidate objects are {A, B, C, D}.

At moment 3 s, assume that A expires. Object A can dominate object B, so we decrease the number of dominators of B. Then, object A is removed. The candidate objects are {B, C, D}.

We summarize the processes of initializing candidate objects at the starting moment (Algorithm 1), updating candidate objects when a new object arrives (Algorithm 2), and updating candidate objects when an object expires (Algorithm 3). Algorithm 1 finds the candidate objects Candid from the set of objects $P$ at the starting moment. For each object $p_i \in P$, we compute the number of dominators and dominatees (lines 3 to 11). Line 5 calls the function GetDomRel($p_i$, $p_j$) to determine the dominance

---

**Algorithm 1 Initialize candidate objects**

1. procedure INITCANDID($P$, $k$)  
   \(\triangleright\) Invoked at the start moment
2. Candid <- $\emptyset$
3. for all $p_i \in P$ do
4.   for all $p_j \in P - \{p_i\}$ do
5.     dom <- GetDomRel($p_i$, $p_j$);
6.     if dom = 1 then \(\triangleright\) $p_i$ dominates $p_j$
7.       $p_i$.dominante <- $p_i$.dominante $\cup \{p_j\}$;
8.     else if dom = -1 then \(\triangleright\) $p_i$ is dominated by $p_j$
9.       $p_i$.num <- $p_i$.num + 1;
10.    end if
11.   end for
12. if $p_i$.num < $k$ then
13.   Candid <- Candid $\cup \{p_i\}$;
14. end if
15. end for
16. end procedure

**Algorithm 2 Updating candidates when a new object arrives**

1. procedure UPDATECANDIDFORNEW($P$, $k$, Candid)
2. \(\triangleright\) Invoked for each incoming object $p$
3. $p$.num <- 0;
4. $p$.dominante <- $\emptyset$;
5. for all $p_j \in P$ do
6.   dom <- GetDomRel($p$, $p_j$);
7.   if dom = 1 then \(\triangleright\) $p$ dominates $p_j$
8.     $p$.dominante <- $p$.dominante $\cup \{p_j\}$;
9.     $p_j$.num <- $p_j$.num + 1;
10.    if $p_j$.num $\geq$ $k$ then
11.       Candid <- Candid $\cup \{p_j\}$;
12.    end if
13. else if dom = -1 then \(\triangleright\) $p$ is dominated by $p_j$
14.     $p$.num <- $p$.num + 1;
15.     $p_j$.dominante <- $p_j$.dominante $\cup \{p\}$;
16.    end if
17. end for
18. if $p$.num < $k$ then
19.   Candid <- Candid $\cup \{p\}$;
20. $P$ <- $P$ $\cup \{p\}$;
21. end if
22. end procedure
relationship between \( p_i \) and \( p_j \). If the returned value \( \text{dom} \) equals one, \( p_i \) dominates \( p_j \) (line 6). We add \( p_j \) to the dominatee list of \( p_i \) (line 7). If the returned value \( \text{dom} \) equals \(-1\), \( p_i \) is dominated by \( p_j \) (line 8). We increment the number of dominators by one (line 9). After checking for dominance relationships with other objects in \( P - \{p_j\} \), we determine whether \( p_i \) is a candidate object. If the number of dominators is fewer than \( k \), it is a candidate object and we add it to the candidate set \( \text{Candid} \).

Algorithm 2 summarizes the procedure of updating candidate objects when a new object \( p \) arrives. First, we initialize the dominator number of \( p \) (line 2) and the dominatee list of \( p \) (line 3). Lines 4 to 16 update the dominator number and the dominatee list of \( p \), in addition to updating the domination information of the objects influenced by the incoming object \( p \). If \( p \) dominates an object \( p_j \in P \), we update the dominatee list of \( p \) (line 7) and update the dominator number of \( p_j \) (line 8). If the dominator number of \( p_j \) is equal to or greater than \( k \), we remove \( p_j \) from the set of candidates \( \text{Candid} \) (lines 9 to 11). If \( p \) is dominated by \( p_j \), we increment the dominator number of \( p \) by one (line 13) and add \( p \) to the dominatee list of \( p_j \) (line 14). Finally, if the dominator number of \( p \) is lower than \( k \), we add it to \( \text{Candid} \) (lines 17 to 20).

Algorithm 3 summarizes the procedure of updating the candidate objects when an object \( p \) expires. First, we process the dominatees of \( p \) (lines 2 to 7). For each dominatee \( p_j \), we decrement its dominator number by one (line 3). If the dominator number is lower than \( k \), we add \( p_j \) to \( \text{Candid} \) (lines 4 to 6). If \( p \) belongs to list \( \text{Candid} \), we remove it from \( \text{Candid} \) (line 8 to line 10). Finally, we remove \( p \) from the set of active objects \( P \) (line 11).

4.2 Updating candidate objects using dominance graph

To update candidate objects incrementally, we propose the use of Algorithms 2 and 3. In these algorithms, we use hash tables to store the domination information of active objects. When the domination information of an object is influenced by incoming objects or expired objects, we can locate and change the domination information directly. We can improve the aforementioned algorithms by using a dominance graph. Algorithm 2 can be improved by using the transitive property of the domination relationship.

In the dominance graph \( G \) of a set of objects \( P \), the vertices denote the objects and the directed edges denote the domination relationships. If object \( p_i \) dominates object \( p_j \), the edge between \( p_i \) and \( p_j \) is from \( p_i \) to \( p_j \). For example, in Fig. 5, object \( A \) dominates object \( B \) and there is an edge from \( A \) to \( B \). The numbers in parentheses are dominator numbers of objects. For example, \( B(1) \) means \( B \) has one dominator.

Figure 5 shows the dominance graphs of the objects at moments \( t = 1 \text{ s}, t = 2 \text{ s}, \) and \( t = 3 \text{ s} \).

In Algorithm 2, line 5 checks the dominance relationships between \( p \) and each active object \( p_j \in P \) by using the function \( \text{GetDomRel}(p, p_j) \). Using the dominance graph, we can avoid a few dominance checks. If object \( p \) dominates another object \( p_j \), an edge from \( p \) to \( p_j \) exists. We can infer that \( p \) dominates \( p_j \) too if \( p_j \) dominates \( p \). In the dominance graph \( G \), the descendants of \( p \) could be dominated by \( p \). In contrast, the ancestors of \( p \) could dominate \( p \).

For example, in Fig. 5, at \( t = 2 \text{ s} \), \( D \) is the incoming object. \( D \) does not belong to the graph \( G \) at \( 1 \text{ s} \). We compute the dominance relationships between \( D \) and other active objects \{A, B, C\}. The first object that has a dominance relationship with \( D \) is \( C \). \( C \) dominates \( D \).

We get the edge \( CD \) that connects \( D \) with the objects in the graph. The ancestors of \( C \) could be the dominators of \( D \). Assuming that there is an object dominated by \( D \),
we can also use this edge to connect D with the graph. The descendants of D could be the dominatrees of D. After retrieving the ancestors and descendants of D, we check for dominance relationships between D and other objects. Thus, we can avoid performing dominance checks by using the transitive property of dominance relationships.

Algorithm 4 summarizes the procedure for updating the candidate objects pertaining to an incoming object p when using a dominance graph G. First, we add p as a vertex to G (line 1) and initialize its dominator number as zero (line 2). When there exists an object p_j that has not been checked, we execute the algorithm. If p dominates p_j (line 6), we add the edge p_p_j to the graph (line 7), update the dominator number of p_j (line 8 to line 11), and update the dominator number of the objects that could be dominated by p_j by using the recursive function UpdateDominator (line 12). If p is dominated by p_j (line 13), we add the edge p_j p to the graph (line 14), increment the dominator number of p by one (line 15), and retrieve the dominators of p_j that could also dominate p by using the recursive function UpdateDominator (line 16). Thereafter, we mark p_j as a checked object.

Algorithm 4 Update candidates for an incoming object by using dominance graph

1: procedure UpdateCandidatesForNewGraph(p, P, G, k, Candid)
   2:   G.V = G.V ∪ {p};  \textcolor{red}{\triangleright} Invoked for each incoming object p
   3:   p.num ← 0;
   4:   while There is a p_j ∈ P unchecked. do
   5:       dom ← GetDomRel(p, p_j);
   6:       if dom = 1 then \textcolor{red}{\triangleright} p dominates p_j,
              G.E ← G.E ∪ {p_p_j};
   7:              p_j.num = p_j.num + 1;
   8:       if p_j.num >= k then
   9:           Candid ← Candid − {p_j};
  10:      end if
  11:  UpdateDominator(p, p_j, G, Candid);
  12:  else if dom = −1 then \textcolor{red}{\triangleright} p is dominated by p_j,
               G.E ← G.E ∪ {p_j.p};
  13:          p.num = p.num + 1;
  14:  UpdateDominator(p, p_j, G, Candid);
  15:  end if
  16:  end while
  17:  if p.num < k then
  18:     Candid ← Candid ∪ {p};
  19:     P ← P ∪ {p};
  20:  end if
  21: end procedure

Algorithm 5 summarizes the procedure of updating the objects that could be dominated by p_j by using the dominance graph. The neighbors p_t of p_j that have edges from p_j are processed using the function GetDomedNB. If p_t is not checked (line 5), its dominator number is incremented by one (line 4). If its dominator number is equal to or greater than k, we remove it from the candidate set (lines 5 to 7). Then, the function UpdateDominatee is called recursively to process the objects that could be dominated by p_t (line 8). Finally, we mark p_t as checked (line 9).

Algorithm 6 summarizes the procedure of updating the dominators p_t of p_j. We retrieve neighbors having edges pointing toward p_j by using the function UpdateDominatee (line 2). If p_t is not checked (line 3), the dominator number of p is incremented by one (line 4). Then, we recursively process the dominators of p_t by calling the function UpdateDominator (line 5). Finally, we mark p_t as checked (line 6).

4.3 Computing skyline groups

After updating the candidate objects, we can construct the skyline groups incrementally based on the dynamic programming algorithm\cite{3}. The basic idea of the dynamic programming algorithm is that a skyline group cannot contain any non-skyline subgroups. We use

Algorithm 5 Update dominatees of object p

1: procedure UpdateDominator(p, p_j, G, Candid)
2:   for all p_t ∈ GetDomedNBs(p_j) do
3:       if p_t.checked = false then
4:         p_t.num = p_t.num + 1;
5:       if p_t.num >= k then
6:         Candid ← Candid − {p_t};
7:       end if
8:       UpdateDominatee(p, p_t, G, Candid);
9:   end if
10: end for
11: end procedure

Algorithm 6 Update dominators of object p

1: procedure UpdateDominator(p, p_j, G, Candid)
2:   for all p_t ∈ GetDomNBs(p) do
3:       if p_t.checked = false then
4:         p.num = p.num + 1;
5:       UpdateDominator(p, p_t, G, Candid);
6:       p_t.checked = true;
7:       end if
8:   end for
9: end procedure
to denote the \( k \)-item skyline groups from \( m \) candidate objects. The sub-problems associated with computing \( \text{GSky}_m^k \) are the computation of \( \text{GSky}_{m-1}^k \) and the computation of the skyline groups from \( \text{GSky}_{m-1}^{m-1} \) by adding the object \( \text{candid}_m \). Sub-problem \( \text{GSky}_{m-1}^k \) involves finding the \( k \)-item skyline groups from the first \( (m-1) \) candidate objects, i.e., \{candid_1, \ldots, candid_{m-1}\}. \( \text{GSky}_{m-1}^{m-1} \) denotes the \((k-1)\)-item skyline groups from the first \((m-1)\) candidate objects. The sub-problem is to find the \( k \)-item skyline groups from the groups \( g_i \cup \{\text{candid}_m\} \), where \( g_i \in \text{GSky}_{m-1}^{m-1} \).

In this paper, to avoid execution of the dynamic programming algorithm from scratch, we maintain a \( \text{GSky} \) matrix to record the skyline groups generated by the dynamic programming algorithm, as shown in Fig. 6. For example, at 1 s, cell \( (2,2) \) contains the skyline groups \( \text{GSky}_2^2 \). The row number denotes \( k \) value, and the column number denotes \( m \) value. In this example, \( m = 2 \) implies we process \( \text{candid}_2 = B \).

When an object arrives and is a candidate, we add a column to the matrix. For example, in Fig. 6, at 2 s, object \( D \) arrives and it is a candidate; therefore, we add a column to the matrix. The column contains two cells \( \text{GSky}_1^1 = \{A,C\} \) and \( \text{GSky}_2^2 = \{AB, AC, CD\} \). We construct the two cells based on cells \((1,3)\) and \((2,3)\).

When a candidate object expires, we delete a column and update the cells influenced. For example, in Fig. 6, at 3 s, candidate \( A \) expires. Hence, we remove the first column of the matrix and update the cells influenced.

---

**Fig. 6  \( \text{GSky} \) matrix. The symbol \( G(k,m) \) is used to denote \( \text{GSky}_m^k \).**
5.3 Performance with respect to different attribute numbers

In the second experiment, we evaluated the performance of the proposed Algorithm 4, in dealing with objects having different attributes. We conducted experiments using the datasets $D_1$, $D_2$, and $D_3$, which have two, three, and four attributes, respectively. We recorded the time required for processing the datasets. The window size ranged from 100 to 500.

Figure 8 shows the time needed for finding three-item skyline groups and four-item skyline groups, respectively. In the figures, the elapsed time increases as the attribute number increases, because the number of skyline groups increases as the attribute number increases. Moreover, we can observe that the elapsed time increases with window size.

5.4 Number of results with respect to different window sizes

In the third experiment, we used dataset $D_1$, which contains objects with two attributes. The window size was 100 to 500. With respect to different window sizes, we reported the number of candidate objects and the number of skyline groups.

Figure 9a shows the number of candidate objects and the number of skyline groups when finding three-item skyline groups. Figure 9b shows the same numbers when finding four-item skyline groups. As shown in Fig. 9, algorithm performance varies little with window size.

5.5 Number of skyline groups varying with time

In the fourth experiment, we determined the number of
skyline groups and the number of the candidate objects with time. The window size was 100 and we used dataset $D_1$, in which all objects have two attributes. We observed the numbers of candidate objects and skyline groups at each moment.

Figure 10 shows the variation in the numbers when determining three- and four-item skyline groups, respectively. The numbers do not vary with time.

6 Conclusions and Future Work

In this paper, we proposed algorithms to find and update skyline groups over a data stream. The algorithm selects candidate objects incrementally and constructs a dynamic programming matrix incrementally. Moreover, we studied how to use a dominance graph for selecting candidate objects for a data stream. We evaluated the performance of the proposed algorithms experimentally. In the future, we will consider how to process skyline groups over distributed data streams.

Acknowledgments

This work was supported by the Fundamental Research Funds for the Central Universities (Nos. FRF-TP-14-025A1 and FRF-TP-15-025A2). This work was also supported by the Key Technologies Research and Development Program of 12th Five-Year Plan of China (No. 2013BAI13B06). The authors would like to thank Jiayu Du for revising the language errors.

References

[1] C. Xu and Y. Gao, A novel approach for selecting the top skyline under users’ references, in Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on, 2010, pp. 708–712.
[2] H. Im and S. Park, Group skyline computation, Information Sciences, vol. 188, pp. 151–169, 2011.
[3] C. Li, S. Rajasekaran, N. Zhang, N. Hassan, and G. Das, On skyline groups, IEEE Transactions on Knowledge and Data Engineering (TKDE), vol. 26, no. 4, pp. 942–956, 2014.
[4] S. Börzsönyi and D. K. K. Stocker, The skyline operator, in Proceedings of the International Conference on Data Engineering (ICDE), 2001, pp. 421–430.
[5] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, Skyline with presorting, in Proceedings of the International Conference on Data Engineering (ICDE), 2003, pp. 717–719.
[6] S. Chaudhuri, N. Dalvi, and R. Kaushik, Robust cardinality and cost estimation for skyline operator, in Proceedings of the International Conference on Data Engineering (ICDE), 2006, p. 64.
Xi Guo et al.: Efficient Processing of Skyline Group Queries over a Data Stream

[7] Y. Tao and D. Papadias, Maintaining sliding window skylines on data streams, IEEE Transactions on Knowledge and Data Engineering (TKDE), vol. 18, no. 3, pp. 377–391, 2006.

[8] X. Liu, D. Yang, M. Ye, and W. Lee, U-skyline: A new skyline query for uncertain databases, IEEE Transactions on Knowledge and Data Engineering (TKDE), vol. 25, no. 4, pp. 945–960, 2013.

[9] Y. Li, W. Qu, Z. Li, Y. Xu, C. Ji, and J. Wu, Sky line query based on user preference with mapreduce, in 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing, 2014, pp. 153–158.

[10] P. Godfrey, R. Shipley, and J. Gryz, Maximal vector computation in large data sets, in Proceedings of International Conference on Very Large Data Bases (VLDB), 2005, pp. 229–240.

[11] K. C. K. Lee, W. Lee, B. Zheng, H. Li, and Y. Tian, Z-sky: An efficient skyline query processing framework based on z-order, The VLDB Journal, vol. 19, no. 3, pp. 333–362, 2010.

[12] D. Kossemann, F. Ramsak, and S. Rost, Shooting stars in the sky: An online algorithm for skyline queries, in Proceedings of International Conference on Very Large Data Bases (VLDB), 2002, pp. 275–286.

[13] D. Papadias, Y. Tao, G. Fu, and B. Seeger, An optimal and progressive algorithm for skyline queries, in ACM SIGMOD 2003, 2003, pp. 467–478.

[14] A. Ercha, W. Huang, S. Liu, L. Shi, J. Gong, Y. Chen, and H. Shen, A regional ionospheric tec mapping technique over China and adjacent areas: Gnsd data processing and dineof analysis, SCIENCE CHINA Information Sciences, vol. 58, no. 10, p. 107201, 2015.

[15] X. Li, H. Wang, B. Ding, and X. Li, Mabpan optimal resource allocation approach in data center networks, SCIENCE CHINA Information Sciences, vol. 57, no. 16, p. 102801, 2014.

[16] C. C. Aggarwal, ed., Data Streams: Models and Algorithms. Springer, 2007.

[17] M. Morse, J. M. Patel, and W. I. Grosky, Efficient continuous skyline computation, in Proceedings of the International Conference on Data Engineering (ICDE), 2006, p. 108.

[18] Y. Lee, K. Lee, and M. H. Kim, Efficient processing of multiple continuous skyline queries over a data stream, Information Sciences, vol. 221, pp. 316–337, 2013.

Xi Guo received the BS degree in software engineering from Northeastern University, China, in 2006, the MS degree in computer science from Northeastern University, China, in 2008, and the PhD degree in information science from Nagoya University, Japan, in 2012. Her research interests include spatial databases, data privacy, and machine learning. Currently, she is a lecturer at the School of Computer and Communication Engineering at University of Science and Technology Beijing, China. She is a member of the Beijing Key Laboratory of Knowledge Engineering for Materials Science Beijing, China.

Hailing Li is currently a postgraduate student at University of Science and Technology Beijing, China. She received her bachelor degree from University of Science and Technology of Hebei in 2013. Her principal research interests include data analysis, data mining, and big data processing. During the period of learning, she has obtained scholarship many times.

Aziguli Wulamu, professor, is currently an associate supervisor of the Beijing Key Laboratory of Knowledge Engineering for Materials Science Beijing, China. She received her PhD degree from University of Science and Technology Beijing, China, in 2004. She has been engaged in knowledge engineering, natural language processing, and innovation theory research and has get a number of achievements. In recent years, she has successively hosted and completed a number of research programs of Natural Science Foundation of Beijing, 863 projects, and Key Projects of National Science and Technology 10th 5-year.

Yonghong Xie is currently an associate professor of the Department of Computer Science at University of Science and Technology Beijing. She also served as a science and technology project evaluation expert of the Haidian District Scientific and Technical Committee, Beijing. She received her PhD degree from University of Science and Technology Beijing, China, in 2005. She has been engaged in knowledge discovery, intelligent system, database technology and has get a number of achievements. In recent years, she successively presided over a number of research programs of National Natural Science Foundation of China, Ministry of Education Science and Technology key projects, and some other transverse projects.

Yajing Fu is currently a postgraduate student at University of Science and Technology Beijing, China. She is majored in software engineering. She received her bachelor degree from Chengdu University of Information and Technology in 2013. Her principal research interests include data mining and hierarchical temporal memory.