Information Sciences 587 (2022) 300–322

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Efficient computation of G-Skyline groups on massive data

Xixian Han [a,*], Jinbao Wang [a], Jianzhong Li [a], Hong Gao [a]

[a] School of Computer Science and Technology, Harbin Institute of Technology, China

ARTICLE INFO

Article history:
Received 29 April 2021
Received in revised form 27 October 2021
Accepted 11 December 2021
Available online 18 December 2021

Keywords:
G-Skyline
Sublinear-I/O method
Reuse principle
Massive data

ABSTRACT

In many practical applications, G-Skyline query is an important operation to return the best tuple groups, which are not g-dominated by other tuple groups of the same size, from a potentially huge data space. It is found that the existing G-Skyline algorithms cannot deal well with massive data due to high I/O cost and high computation cost. This paper proposes a novel GPR algorithm, which is based on presorting and reuse principle, to compute G-Skyline groups on massive data efficiently. The execution of GPR consists of two phases: acquisition of the candidate tuples and computation of G-Skyline groups. The sublinear-I/O method is proposed in phase 1 to scan the presorted table, which is proved to hold early termination property. This paper devises the basic framework of phase 2 and analyzes its execution cost. The SR strategy is utilized to reuse the subset computation results effectively and reduce the execution cost of phase 2 considerably. The extensive experimental results, conducted on synthetic and real-life data sets, show that GPR outperforms the existing algorithms significantly.

1. Introduction

Skyline is an important operation in many practical applications to find the pareto optimal tuples from a potentially huge data space [4,5,7,8,20,25,28,37,42]. It specifies a subset of attributes as *skyline criteria*, and discovers all tuples which are not dominated by other tuples. Specifically, given two tuples t_1 and t_2, t_1 dominates t_2 if t_1 is no worse than t_2 in all attributes of skyline criteria and strictly better than t_2 in at least one attribute.

The skyline actually returns a candidate set for the optimum, which helps the decision-makers narrow down the exploration space effectively. No matter how the weights are specified among the skyline criteria, the best (or pareto optimal) solution is always contained in the skyline results. For example, considering the set of hotels, whose prices and distances to the beach are depicted in Fig. 1. Suppose that users prefer the hotels with the lower prices and the smaller distances, the skyline of the hotels returns $\{p_1, p_2, p_3, p_4\}$ and the other hotels are dominated by at least one of them. The best hotel can always be found among p_1, p_2, p_3 and p_4 regardless of the different trade-offs between the two aspects.

But then, users often require more than just best individual tuple in many real-life cases, as stated in the following example.

Example 1.1. Given the hotels depicted in Fig. 1, just one hotel cannot provide enough accommodation for every attendee in a large conference. Thus the conference organizer needs to select the best 3 hotels. One simple method to address this issue is to enumerate all the combinations of size 3 from the skyline results. However, this simple method does not cover all the

* Corresponding author.
E-mail addresses: hanxx@hit.edu.cn (X. Han), wangjinbao@hit.edu.cn (J. Wang), lijzh@hit.edu.cn (J. Li), honggao@hit.edu.cn (H. Gao).

https://doi.org/10.1016/j.ins.2021.12.028
0020-0255/

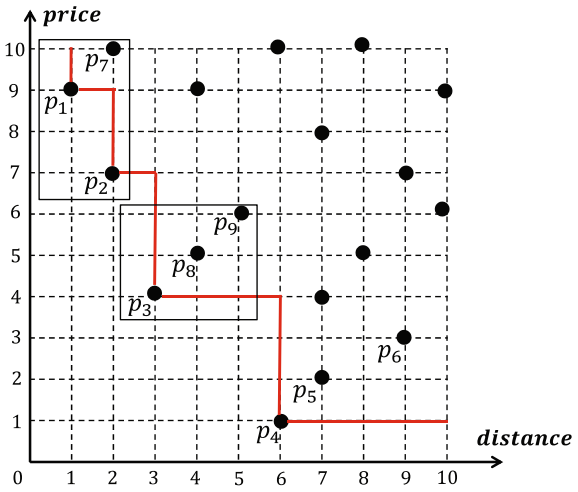


Fig. 1. The illustration of a skyline example.

possible choices. For example, if the conference organizer concerns the shorter distance more than lower price, they should prefer $\{p_1, p_2, p_7\}$ with the shortest distances, where p_1 and p_2 are skyline tuples, and p_7 is dominated only by them. $\{p_4, p_5, p_6\}$ should be a proper choice if the organizer concerns the lower price more. $\{p_3, p_8, p_9\}$ can be another choice if the organizer treats the two attributes with relatively equal consideration. Among the latter two cases, p_5, p_6, p_8 and p_9 are not skyline tuples.

In the above example, many users require the best group of the tuples instead of the best individual tuple, and the direct combination of the skyline cannot cover all the possible options. In order to meet the actual requirement, the traditional skyline is generalized to group-based skyline (G-Skyline), which is formally defined in [17]. In this paper, the group of tuples of size k is called k-tuple group. Analogous to the definition of the traditional skyline, G-Skyline aims to find the k-tuple groups which are not *g-dominated* by any other k-tuple groups. To be specific, given two k-tuple groups G_1 and G_2, G_1 g-dominates G_2, if, $\forall t_2 \in G_2$, there is a distinct tuple $t_1 \in G_1$ such that t_1 is no worse than t_2, and for at least one corresponding pair of tuples, t_1 is strictly better than t_2.

Besides the group dominance defined in G-Skyline, [10,14,38] propose another group dominance relation, which compares aggregate vectors of the k-tuple groups. Just as pointed out in [17], this kind of group dominance definition has two problems.

- It is difficult to find a good aggregate function to calculate the aggregate values.
- It does not generate all the optimal groups.

Consequently, we only consider the group dominance relation of G-Skyline in this paper.

Most of the existing G-Skyline algorithms [17,18,34,36,41] are based on some index structures (DSG or MDG), which depend on the dominance relation between tuples. Since the specified skyline criteria is normally a subset of the total attributes, the index structures whose numbers are exponential to total attribute number have to be built in order to cover all possible skyline criteria. Therefore, the required index structures have to be built in an ad hoc way in practical applications. On massive data, the construction cost of the index structures is relatively high. Besides, the generation of G-Skyline groups by the index structures is also executed with a high computation cost. For the data that cannot be indexed, an index-independent algorithm SP is devised in [41]. Apart from the external sort that has to be executed first, SP performs multiple scans to calculate the dominance information of the candidates, and requires the expensive computation in memory. To sum up, the existing G-Skyline algorithms cannot deal well with massive data due to high I/O cost and high computation cost.

This paper proposes a novel algorithm *GPR* (*G*-Skyline based on **P**resorting and **R**euse principle) to compute G-Skyline groups on massive data efficiently. GPR does not require the index structures built on the specified attribute subsets. Given table T, GPR presorts T to generate PT (**P**resorted **T**able), whose tuples are arranged in the order of round-robin retrieval on the sorted lists of the attributes. The execution of GPR consists of two phases. In phase 1, the sublinear-I/O method is presented to scan PT and obtain the candidate tuples required for computing G-Skyline groups. It is proved that GPR has the property of early termination in phase 1, and only a proportion of the tuples in PT need to be retrieved. In this paper, GPR first proposes a basic framework of phase 2 to compute G-Skyline groups by the candidate tuples. The cost analysis of the basic framework is developed, which determines the performance bottleneck. In order to solve the bottleneck, the SR (**S**ubset **R**euse) strategy is devised in this paper to improve the basic framework significantly by reusing the subset generation results. The extensive experiments are conducted on synthetic and real-life data sets. The experimental results show that, GPR can achieve two orders of magnitude speedup compared with the existing algorithms.

The contributions of this paper are listed as follows:

- This paper proposes a novel algorithm GPR, which adopts presorting and reuse principle, to compute G-Skyline groups on massive data efficiently.
- The sublinear-I/O method is presented for the processing in phase 1. It is proved that GPR has the property of early termination in phase 1, accompanied by the analysis of scan depth and candidate number.
- The basic framework of phase 2 is devised in this paper, along with its performance analysis. The SR strategy is presented to reduce the computation cost of phase 2 considerably by reuse principle.
- The extensive experimental results show that GPR outperforms the existing algorithms significantly.

The rest of the paper is organized as follows. Section 2 reviews related works, followed by preliminaries in Section 3. Section 4 introduces the generation of the presorted table. Section 5 proposes GPR algorithm. Section 6 evaluates the performance of GPR and Section 7 concludes the paper.

2. Related works

The skyline algorithms, aiming to find the best individual tuples, can be classified into three types: indexing-based algorithms [11,13,19,25,32], generic algorithms [1,7,8,39] and lattice-based algorithms [12,24]. Due to its practical importance, skyline is widely used in many applications, such as incomplete data [22], probabilistic data [23], encrypted data [3,40], road networks [21], social networks [27,30], service composition [15,35], feature selection [9] and cohort intelligence [26], just to name a few.

Instead of the best individual tuples, the group-based skyline algorithms return the optimal tuple groups. By the different definitions of group dominance, the existing group-based skyline algorithms can be divided into two categories: aggregate-vector-based algorithms and G-Skyline algorithms, which are surveyed in Section 2.1 and Section 2.2, respectively.

2.1. Aggregate-vector-based algorithms

The problem of computing skyline groups is formulated and investigated in [14,38]. Each k-tuple group can be represented as an aggregate vector, where the ith component of the vector is calculated by utilizing the aggregate function (SUM, MAX, or MIN) on the corresponding attributes of the k tuples. The tuple groups are compared by the corresponding aggregate vectors according to the tuple dominance. Some novel algorithmic techniques (*output compression*, *input pruning* and *search space pruning*) are developed to compute k-tuple skyline groups quickly.

Im et al. [10] study the problem of group skyline computation. As a preliminary step, an intuitive group skyline algorithm GIncremental is investigated. It is based on the intuition that every k-skyline group is likely to contain at least one $(k-1)$-skyline group as its subset. GIncremental first removes all k-dominated tuples, and then sorts the remaining tuples according to an entropy-based monotone preference function. The candidate groups are produced in a progressive manner, and a candidate i-skyline group is generated by extending an $(i-1)$-skyline group with a remaining tuple. GIncremental can be regarded as a practical and approximate group skyline algorithm, since it may miss some k-skyline groups if k is extremely large. Another group skyline algorithm GDynamic is proposed to improve GIncremental. The execution of GDynamic is equivalent to a dynamic algorithm filling a table of group skylines. It also computes group skylines in a progressive way, but generates the set of all candidate groups including a specific tuple at once.

Discussion. The kind of algorithms determines the dominance relation between the tuple groups by the aggregate vectors of the tuple groups. We do not consider it in this paper, because it is difficult to specify a proper aggregate function and aggregate-vector-based algorithms do not generate all the best groups.

2.2. G-Skyline algorithms

Liu et al. [17,18] formally define a novel group-based skyline, G-Skyline, to find pareto optimal groups. G-Skyline returns the tuple groups that are not g-dominated by any other tuple groups of the same size. It is proved that the tuples in the G-Skyline groups of size k all belong to the first k skyline layers. The efficient algorithms are designed to compute the first k skyline layers, on which a novel structure DSG (directed skyline graph) is developed to represent their dominance relation. Given the constructed DSG, G-Skyline can be formulated as the search problem in a set enumeration tree. Two algorithms (point-wise algorithm and unit-group-wise algorithm) are devised to compute G-Skyline groups. In point-wise algorithm, the set enumeration tree of the tuples in the first k skyline layers is generated in level-wise way. Each node corresponds to a candidate group. The current node can be expanded to a set of child nodes in the next level by adding the new points. The subtree pruning and tail set pruning are developed in point-wise algorithm to prune the non-G-Skyline candidates. The unit-group-wise algorithm adopts a similar execution as point-wise algorithm, but adds one unit group at a time when expanding current node.

Yu et al. [36] define the first k skyline layers as MSL (multiple skyline layers), and propose a novel algorithm to construct MSL efficiently. It is found that the information of points in each layer can be obtained by the information of each point's

layer. Accordingly, each skyline layer can be constructed by determining the layers that the points belong to. The construction of MSL is to search concurrently in each dimension, find the layer for each point visited and store the point into the corresponding layer. Based on the observation that the first layer contributes most to the group skylines, two fast algorithms (F_PWise and F_UWise), utilizing a combination queue, are devised to improve the algorithms in [17,18]. Given DSG built from MSL, F_PWise constructs G-Skyline by the combination tree and adopts edge pruning for DSG to speed up the execution. F_UWise improves UWise + algorithm by adding combination queue, and gives primary groups and second groups independently.

Wang et al. [34] present a new g-skyline support structure MDG (minimum dominance graph) to replace DSG. MDG only includes points which are dominated by less than k points. It is proved that MDG is a minimum g-skyline support structure and has no redundancy. A new two-step approach MDS is proposed to compute the G-Skyline groups: constructing MDG and finding G-Skyline groups based on MDG. In the first step, two construction algorithms are introduced to construct MDG after sorting the points in the ascending order of the sum on all dimensions. In the second step, P-MDS and G-MDS are developed to find all the G-Skyline groups. P-MDS first sorts the points of MDG in the ascending order of the numbers of their parents to avoid the duplicate generation. After adding each skyline point into the empty current group, P-MDS starts a depth-first search (DFS) to add proper single points in the current group. Parent pruning strategy is utilized in P-MDS. More precisely, for each candidate point, it can be added to the current group if all its parents are contained in the current group. DFS continues until the current group contains k points or has more than k points. For G-MDS, it first sorts the points of MDG in reverse order of that used in P-MDS, then it employs DFS to enumerate the candidate groups as in P-MDS. But G-MDS generates new candidate groups by adding the parent group of candidate point, rather than single point in P-MDS. The property of skyline-combination is employed to optimize P-MDS and G-MDS.

Zhou et al. [41] consider the problem of computing pareto optimal groups with the improvement in two aspects: progressiveness and efficiency. A new index-based algorithm LU is developed by adopting the layered optimum strategy. The execution of LU is based on some new lemmas. It is proved that, each G-Skyline group of size k contains at most $(k-i+1)$ points within the ith skyline layer $SL_i (1 \leqslant i \leqslant k)$, and any group containing at least one point in SL_i only can be g-dominated by the groups within $\bigcup_{j=1}^{i-1} SL_j$. Given DSG, LU algorithm generates G-Skyline groups containing j points within SL_i for $1 \leqslant i \leqslant k$ and $1 \leqslant j \leqslant k-i+1$ respectively and directly. An index-independent algorithm SP is also proposed for the data that cannot be indexed. The data points are sorted first by a monotone function to ensure that any point is only dominated by the points before it. Then SP performs a loop over the points. *GPruSet* is used to keep some points with no less than $(k-1)$ parents and identify other unqualified points. For the point p currently processed, if it is dominated by one point in *GPruSet*, it can be pruned directly. Otherwise, the unit group $u(p)$ of p is calculated. If $|u_p| \geqslant k$, SP inserts p into *GPruSet* and removes the points in *GPruSet* dominated by p. If $|u_p| = k, u_p$ is reported as a G-Skyline group. If $|u_p| < k, u_p$ is inserted to the candidate set *CG* and combined with the candidate groups in *CG*. SP reports the candidate groups whose sizes are equal to k as G-Skyline groups and removes those whose sizes are larger than k.

Discussion. The most of the existing G-Skyline algorithms are based on the specified index structures (DSG or MDG), which depend on the dominance relation between tuples. However, the skyline criteria is usually a subset of total attributes. The specified index structures whose numbers are exponential to total attribute number have to be built to cover all the possible skyline criteria. Of course, this is prohibitively expensive. Consequently, the algorithms have to build DSG or MDG in ad hoc way, i.e. constructing the index structure at the beginning of the actual execution. This usually involves external sorting of the data set first, which incurs a high I/O cost on massive data. Besides, after obtaining the index structures, the computation of G-Skyline groups is expensive also since it involves the generation of combinations from a relatively large candidate set. For the index-independent algorithm, besides the external sorting, it also needs multiple scans on the table to compute the dominance information of the tuples. Moreover, the index-independent algorithm also requires the complicated computation. Its performance is very poor on massive data.

3. Preliminaries

3.1. Problem definition

Given a table $T(A_1, \ldots, A_M)$ with n tuples, $\forall t \in T$, let $t.A_i$ be the value of attribute A_i of t. Without loss of generality, let $AS_{sky} = \{A_1, \ldots, A_m\}$ be the specified skyline criteria, on which dominance relation between tuples is defined.

Definition 3.1 (*Dominance*). Given a table T and skyline criteria $AS_{sky}, \forall t_1, t_2 \in T, t_1$ dominates t_2 (denoted by $t_1 \prec t_2$) if $\forall i (1 \leqslant i \leqslant m), t_1.A_i \leqslant t_2.A_i$, and $\exists i (1 \leqslant i \leqslant m), t_1.A_i < t_2.A_i$

Definition 3.2 (*Skyline query*). Given a table T and skyline criteria AS_{sky}, skyline query returns a subset of T, each of which is not dominated by other tuples in T.

Given skyline criteria AS_{sky}, skyline returns the best individual tuples in T. Given another parameter k, G-Skyline returns candidates of the best tuple groups of size k in T, which are not *g-dominated* by any other k-tuple groups. Let $\mathscr{P}_k(T)$ denote the set of all the k-subsets of T.

Definition 3.3 (*G-dominance*). Given a table T, skyline criteria AS_{sky} and size parameter k, $\forall G_1 = (t_1, \ldots, t_k), G_2 = (r_1, \ldots, r_k) \in \mathscr{P}_k(T), G_1$ g-dominates G_2 (denoted by $G_1 \prec_g G_2$), if there are two permutations of G_1 and $G_2, G_1 = (t_{f(1)}, \ldots, t_{f(k)})$ and $G_2 = (r_{y(1)}, \ldots, r_{y(k)}), \forall j(1 \leqslant j \leqslant k), t_{f(j)} \prec r_{y(j)}$ or $t_{f(j)} = r_{y(j)}$, and $\exists j(1 \leqslant j \leqslant k), t_{f(j)} \prec r_{y(j)}$.

Definition 3.4 (*G-Skyline query*). Given a table T, skyline criteria AS_{sky} and size parameter k, G-Skyline query returns a subset of $\mathscr{P}_k(T)$, each of which is not g-dominated by other k-tuple groups in $\mathscr{P}_k(T)$.

$\forall t \in T$, its positional index (PI) is a if t is the ath tuple in T, denoted by $T(a)$ [8]. We denote by $T(a, \ldots, b)(a \leqslant b)$ the tuples in T whose PIs are between a and b. The frequently used symbols in this paper are shown in Table 1.

3.2. The overview of GPR algorithm

As analyzed in Section 2.2, the existing algorithms normally have to build the required index structures first. This splits the execution into two parts: index construction and computation of the G-Skyline groups. In part 1, the external sorting is invoked, which incurs a high I/O cost on massive data. In part 2, the computation cost by the built index can be very high. Therefore, we aim to devise an efficient G-Skyline algorithm on massive data, which can effectively reduce the involved I/O cost and computation cost of the existing algorithms.

The processing in part 1 is to find the required tuples and their dominance relation. On massive data, just one full scan on the table requires a relatively high I/O cost, let alone the external sorting which needs multiple scans. Correspondingly, the *sublinear-I/O method*, which retrieves only a part of the tuples, is obviously a better choice. For achieving the goal, this paper utilizes the transitivity of the dominance relation between tuples. This is the reason why we first presort the table (Section 4). The sortedness of the attribute values are exploited to discard the unqualified tuples before retrieving them actually (Section 5.1).

The processing in part 2 is to discover the G-Skyline groups. The computation cost can be very high since this can be treated as the combinations of size k selected from the tuples obtained in part 1. It is observed in this paper that the different types of tuples affect the computation cost differently. Therefore, the execution cost should be analyzed in detail to find the performance bottleneck in part 2. Inspired by the processing for overlapping subproblems in dynamic programming [2], it is best to reuse the expensive immediate computation results to lower the overall computation cost effectively (Section 5.2).

4. The construction of presorted table

This section introduces the construction of presorted table.

The table T can be kept as a set of sorted lists $L_1, \ldots, L_M$. The schema of $L_i(1 \leqslant i \leqslant M)$ is $L_i(PI_T, A_i)$, where PI_T is the positional index of the tuple in T and A_i is the corresponding attribute value of $T(PI_T)$. L_i is sorted in ascending order of A_i. The presorted table PT is built by sorting the tuples of T in the order of round-robin retrieval on the sorted lists. Specifically, $\forall t \in T$, let $PI_i(1 \leqslant i \leqslant M)$ be the positional index of $(t.PI_T, t.A_i)$ in L_i, the tuples in PT are arranged in the ascending order of $\min_{1 \leqslant i \leqslant M} PI_i$.

Implementation detail. Given $L_i(PI_T, A_i)(1 \leqslant i \leqslant M)$, presorting operation first performs sorting on $L_i(PI_T, A_i)$ to generate $PL_i(PI_i, PI_T)$ in the ascending order of PI_T, where PI_i is an implicit attribute in L_i. Obviously, $\forall a(1 \leqslant a \leqslant n), PL_i(a)$ represents the position information of $T(a).A_i$ in L_i. Let $MPI_L(a) = \min_{1 \leqslant i \leqslant M} PL_i(a).PI_i$, the presorted operation merges the column-wise files $PL_1, \ldots, PL_M$ to a single row-wise file, and then sorts the row-wise file in ascending order of MPI_L. The schema of the presorted table is $PT(MPI_L, PI_T, PI_1, \ldots, PI_M)$.

Example 4.1. In this paper, a running example is used to illustrate the execution of the proposed algorithm. As depicted in Fig. 2, the table in running example contains twelve tuples with three attributes. Let $AS_{sky} = \{A_1, A_2\}$ and $k = 3$. The corresponding sorted lists and the presorted table PT are also depicted in Fig. 2. Given $T(4) = \{961, 406, 380\}, L_1(12) = (4, 961), L_2(6) = (4, 406)$ and $L_3(3) = (4, 380), PL_1(4).PI_1 = 12, PL_2(4).PI_2 = 6$ and $PL_3(4).PI_3 = 3$, the MPI_L value is 3, i.e. $\min(12, 6, 3)$.

According to the sortedness of the sorted lists, $\forall t_1, t_2 \in PT$, we say that $t_1 \prec t_2$, i.e. $T(pt_1.PI_T) \prec T(pt_2.PI_T)$, if $\forall 1 \leqslant i \leqslant m, t_1.PI_i < t_2.PI_i$[1]. In the rest of this paper, the computation of G-Skyline groups is performed on PT.

5. The GPR algorithm

This paper introduces a novel algorithm GPR (**G**-Skyline based on **P**resorting and **R**euse principle), which adopts the sublinear-I/O method and subset reuse strategy to compute G-Skyline groups on massive data efficiently. The execution

[1] In this paper, the tuples are considered in general position [31].

Table 1
Summary of symbols.

Symbol	Meaning
AS_{sky}	The used skyline criteria
m	Attribute number in AS_{sky}
k	The size of the required tuple group
$pg(t)$	Parent group of t
$ug(t)$	Unit group of t
S_C	Candidate set maintained in phase 1
B_j	Candidate tuples with j parents

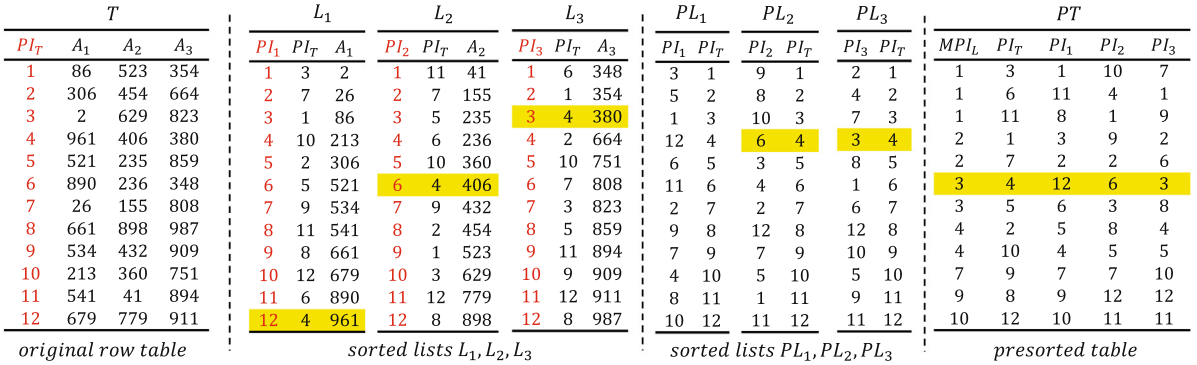


Fig. 2. The illustration of constructing presorted table.

of GPR includes two phases: acquisition of candidate tuples and searching for G-Skyline groups. The two phases are described in Section 5.1 and Section 5.2, respectively.

5.1. Phase 1: acquisition of candidate tuples

In phase 1, GPR acquires the candidate tuples for computing G-Skyline groups.

Firstly, Theorem 5.1 is proposed to present the property of any G-Skyline group. It can be used to verify whether a k-tuple group is a G-Skyline group.

Theorem 5.1. *For any G-Skyline group G, $\forall t \in G$, the tuples which can dominate t are contained in G.*

Proof 5.1. With proof by contradiction, given a G-Skyline group G, assume that there exists at least one tuple r outside G which dominates t. Then we can obtain another k-tuple group G' by replacing t in G with r. Obviously, $G' \prec_g G$, since $r \prec t$ and the other tuples in two k-tuple groups are the same. This contradicts the statement that G is a G-Skyline group.

$\forall t \in PT$, Definition 5.1 defines the parent group of t (denoted by $pg(t)$). Any tuple in $pg(t)$ is called a *parent* of t.

Definition 5.1 (*Parent group*). $\forall t \in PT$, the tuples dominating t constitute the parent group of t.

Theorem 5.2. Given the size parameter k, any tuple t, whose parent group size is no less than k, cannot be contained in any G-Skyline group.

Proof 5.2. With proof by contradiction, given tuple t, assume that $|pg(t)| \geqslant k$ and t is contained in a G-Skyline group G. According to Theorem 5.1, all parents of t must also be contained in G. Therefore, the size of G is more than k, this contradicts the size limit k.

According to Theorem 5.2, given the size parameter k, *the candidate tuples which can be used to generate G-Skyline groups are those having no more than $(k-1)$ parents*. This limits the range of the candidate tuples significantly.

The pseudo-code for processing in phase 1 is listed in Algorithm 1.

Algorithm 1: GPR_Phase1

Input: presorted table PT
Output: the candidate tuples for computing G-Skyline groups
1: S_C initializes to empty set and keeps candidate tuples
2: max heap MH initializes to empty
// sequential scan on PT
3: **for** $a \leftarrow 1$; $a \leqslant n$; a++ **do**
4: let $t \leftarrow PT(a)$ be the currently retrieved tuple in PT
5: $t.sorting\_key \leftarrow \max_{1 \leqslant i \leqslant m} t.PI_i$
// update of max heap
6: **if** MH.size $< k$ **then**
7: insert t into MH
8: **else if**$MH.max > t.sorting\_key$ **then**
9: remove the root node of MH and insert t into MH
10: **end if**
// early termination checking
11: **if** $MH.size = k$ and $MH.max \leqslant t.MPI_L$ **then**
12: **break**;
13: **end if**
// the iteration through S_C for dominance checking
14: **for each** candidate s in S_C **do**
15: **if** $t \prec s$
16: add t to parent group of s
17: **if** $pg(s) = k$ **then**
18: remove s from S_C
19: **end if**
20: **end if**
21: **if** $s \prec t$ **then**
22: add s to parent group of t
23: **if** $pg(t) = k$ **then**
24: **break**
25: **end if**
26: **end if**
27: **end for**
28: **if** $|pg(t)| < k$ **then**
29: append t at the end of S_C
30: **end if**
31: **end for**
32: **return** S_C

In phase 1, GPR aims to find the candidate tuples and their parent groups by a sequential scan on the presorted table PT with sublinear I/Os (line 3 to line 31 in Algorithm 1). During the sequential scan, GPR maintains a set S_C for storing the candidate tuples (initialized in line 1) and a *max heap* MH of size k (initialized in line 2).

Let t be the current tuple retrieved in PT, the sorting key of t used in MH is the maximum value among $PI_1, \ldots, PI_m$ of t, i.e. $\max_{1 \leqslant i \leqslant m} t.PI_i$ (line 4 and line 5). The update of MH is performed as follows (line 6 to line 10).

- If MH contains less than k elements, t is inserted into MH directly.
- Otherwise, if the sorting key of t is less than $MH.max$ (the maximum sorting key of MH), MH removes the element with the maximum sorting key and adds t.

In this way, MH maintains k tuples retrieved so far with *the minimum sorting keys*.

For t retrieved currently, GPR iterates through S_C. Let s be the current candidate tuple visited in S_C. The maintenance of S_C is performed as follows (line 14 to line 27).

- If $t \prec s$, t is added into the parent group of s. If $|pg(s)| = k$, s is removed from S_C.
- If $s \prec t$, s is added into the parent group of t. If $|pg(t)| = k$, t is discarded and the iteration terminates directly.

If $|pg(t)| < k$ at the end of iteration, t is appended into the end of S_C (line 28 to line 30).

Suppose that t has found k parents at the ath tuple in S_C (denoted by $S_C(a)$), Theorem 5.3 proves that t cannot dominate $S_C(a+1),\ldots,S_C(|S_C|)$. Thus, it is safe to discard t and terminate the current iteration, because t cannot contribute to the parent groups of the remaining candidate tuples in S_C.

Theorem 5.3. *During the iteration through S_C, if t finds k parents when visiting $S_C(a)$, t cannot dominate any candidate tuples in $S_C(a+1,\ldots,|S_C|)$.*

Proof 5.3. We prove it by contradiction. Given that t finds k parents when visiting $S_C(a)$, its parent group includes k tuples among $S_C(1,\ldots,a)$. Suppose that t dominates $S_C(b)(a+1\leqslant b\leqslant|S_C|)$, by the transitivity of dominance relation, $S_C(b)$ is also dominated by the tuples in parent group of t. According to the appending way of tuples in S_C, $S_C(b)$ is appended into S_C after $S_C(1),S_C(2),\ldots,S_C(a)$, among which k tuples dominate $S_C(b)$. It is no way to maintain $S_C(b)$ in S_C, which contradicts the assumption.

For the candidate tuple s currently visited in S_C, if $|pg(s)|=k$, s is removed from S_C. Here, there is still some doubt whether s can be removed safely. Theorem 5.4 proves that s cannot be in the parent groups of the candidate tuples not retrieved yet, and GPR can remove s directly.

Theorem 5.4. *If s is removed from S_C, it cannot be in the parent groups of any candidate tuples not retrieved yet.*

Proof 5.4. If s is removed from S_C, it means that $|pg(s)|=k$ and there exists k tuples which dominate s currently. Any tuple s_p in $pg(s)$ either is kept in current S_C or will be removed from S_C afterwards. In the latter case, there are k candidate tuples in S_C dominating s_p, which also dominate s due to the transitivity of dominance relation. In other words, after s is removed from S_C, there exists at least k candidate tuples in S_C dominating s. If s is in the parent group of any tuple v not retrieved yet, $pg(v)$ should have at least k parents in S_C, and v cannot be maintained in S_C.

The existing algorithms involve an external sorting and multiple-pass scan on the table to construct DSG or MDG. On the other hand, GPR can terminate the execution of phase 1 before all tuples are retrieved, which is called *early termination property* in this paper. For the currently retrieved tuple t, as proved in Theorem 5.5, if the early termination condition $t.MPI_L\geqslant MH.max$ is satisfied, phase 1 is over (line 11 to line 13 in Algorithm 1) and GPR acquires the candidate tuples for computing the G-Skyline groups (line 32). Otherwise, GPR retrieves the next tuple in PT with the similar processing until the early termination condition is satisfied or all tuples in PT have been processed.

Theorem 5.5. *Given the currently retrieved tuple t, if $t.MPI_L\geqslant MH.max$, phase 1 ends and GPR maintains the candidate tuples in S_C.*

Proof 5.5. According to the definition of MPI_L, $t.MPI_L=\min_{1\leqslant i\leqslant M}t.PI_i$. If $t.MPI_L\geqslant MH.max,\forall t_x\in MH, t.MPI_L\geqslant\max_{1\leqslant i\leqslant m}t_x.PI_i$. Obviously, $\forall 1\leqslant i\leqslant m, t.PI_i\geqslant t_x.PI_i$, we have $t_x\prec t$. In other words, if $t.MPI_L\geqslant MH.max$, the parent group of t definitely contains k tuples in MH. Besides, the tuples in PT are arranged in the ascending order of MPI_L, all the remaining tuples in PT are dominated by at least k tuples. By Theorem 5.2, they cannot be in any G-Skyline groups. The sequential scan in phase 1 can terminate.

Example 5.1. Given $k=3$, the execution of phase 1 in the running example is illustrated in Fig. 3, where No. is the iteration number, PT Tuple is the currently retrieved PT tuple and Operation represents the specific operations given the current PT tuple, candidate set S_C and max heap MH. Initially, S_C and MH are empty. After the 3*rd* iteration, the first three tuples in PT are retrieved. They are added to S_C and MH, and now $MH.max=11$. Since $PT(3)\prec PT(2)$, $PT(3)$ is a parent of $PT(2)$. The retrieval on PT continues since the MPI_L value of the next PT tuple $PT(4)$ is less than 11. At the 6*th* iteration, $PT(6)$ is dominated by $PT(2),PT(3)$ and $PT(5)$. $PT(6)$ is not maintained in S_C since it cannot be in any G-Skyline group. At the 7*th* iteration, the parent group $\{PT(3),PT(5)\}$ of $PT(2)$ is obtained in the previous iterations. Since $PT(7)\prec PT(2),PT(7)$ is added to $pg(PT(2))$ and $|pg(PT(2))|=3$, $PT(2)$ is removed from S_C. At the 10*th* iteration, since $PT(10).MPI_L=7>MH.max=6$, early termination condition is satisfied and phase 1 terminates.

Next, we analyze the scan depth of phase 1, i.e. the number of tuples retrieved from PT in phase 1.

Assumption 5.1. The attributes $A_1,\ldots,A_M$ are uniformly and independently distributed in $[0,1]$.

Let t_e be the first tuple retrieved in PT with $t_e.MPI_L\geqslant MH.max$. Given the skyline criteria AS_{sky}, this actually means that the m-dimensional hypercube formed by $(1,\ldots,1)$ and $(t_e.MPI_L,\ldots,t_e.MPI_L)$ consists of k tuples in PT, each of which satisfies: $\forall 1\leqslant i\leqslant m, 1\leqslant PI_i\leqslant t_e.MPI_L$. Under Assumption 5.1, $\forall t\in PT$, the probability of $1\leqslant t.PI_i\leqslant t_e.MPI_L$ is $(\frac{t_e.MPI_L}{n})$, and the probability of $1\leqslant t.PI_i\leqslant t_e.MPI_L$ for all $1\leqslant i\leqslant m$ is $(\frac{t_e.MPI_L}{n})^m$. We consider the event to be successful if any tuple t in PT satisfies

no.	PT Tuple	Current candidate set S_C	Current max heap MH	Operation
1	$PT(1)$ = (1,3,1,10,7)	{}	{}:	Add $PT(1)$ to S_C and MH.
2	$PT(2)$ = (1,6,11,4,1)	$\{PT(1): pg()\}$	$\{PT(1): 10\}$:	Add $PT(2)$ to S_C and MH.
3	$PT(3)$ = (1,11,8,1,9)	$\{PT(1): pg(), PT(2): pg()\}$	$\{PT(1): 10, PT(2): 11\}$:	Add $PT(3)$ to S_C, it is a parent of $PT(2)$, add $PT(3)$ to MH and update MH.Max.
4	$PT(4)$ = (2,1,3,9,2)	$\{PT(1): pg(), PT(2): pg(3), PT(3): pg()\}$	$\{PT(1): 10, PT(2): 11, PT(3): 8\}$: 11	Add $PT(4)$ to S_C, rmv $PT(2)$ from MH, add $PT(4)$ and update MH.max.
5	$PT(5)$ = (2,7,2,2,6)	$\{PT(1): pg(), PT(2): pg(3), PT(3): pg(), PT(4): pg()\}$	$\{PT(1): 10, PT(3): 8, PT(4): 9\}$: 10	Add $PT(5)$ to S_C, it is a parent of $PT(2)$ and $PT(4)$, rmv $PT(1)$ from MH, add $PT(5)$ and update MH.max.
6	$PT(6)$ = (3,4,12,6,3)	$\{PT(1): pg(), PT(2): pg(3,5), PT(3): pg(), PT(4): pg(5), PT(5): pg()\}$	$\{PT(3): 8, PT(4): 9, PT(5): 2\}$: 9	$PT(6)$ is dominated by $PT(2)$, $PT(3)$ and $PT(5)$.
7	$PT(7)$ = (3,5,6,3,8)	$\{PT(1): pg(), PT(2): pg(3,5), PT(3): pg(), PT(4): pg(5), PT(5): pg()\}$	$\{PT(3): 8, PT(4): 9, PT(5): 2\}$: 9	Add $PT(7)$ to S_C, rmv $PT(2)$ since $PT(7)$ dominates it, rmv $PT(4)$ from MH, add $PT(7)$ and update MH.max
8	$PT(8)$ = (4,2,5,8,4)	$\{PT(1): pg(), PT(3): pg(), PT(4): pg(5), PT(5): pg(), PT(7): pg(5)\}$	$\{PT(3): 8, PT(5): 2, PT(7): 6\}$: 8	Add $PT(8)$ to S_C, $PT(5)$ dominates it.
9	$PT(9)$ = (4,10,4,5,5)	$\{PT(1): pg(), PT(3): pg(), PT(4): pg(5), PT(5): pg(), PT(7): pg(5), PT(8): pg(5)\}$	$\{PT(3): 8, PT(5): 2, PT(7): 6\}$: 8	Add $PT(9)$ to S_C, it is a parent of $T(8)$, $PT(5)$ dominates $PT(9)$, rmv $PT(3)$ from MH, add $PT(9)$ and update MH.max.
10	$PT(10)$ = (7,9,7,7,10)	$\{PT(1): pg(), PT(3): pg(), PT(4): pg(5), PT(5): pg(), PT(7): pg(5), PT(8): pg(5,9), PT(9): pg(5)\}$	$\{PT(5): 2, PT(7): 6, PT(9): 5\}$: 6	Early termination condition is satisfied.

Fig. 3. The illustration of execution in phase 1.

$1 \leqslant t.PI_i \leqslant t_e.MPI_L$ for all $1 \leqslant i \leqslant m$, and the number of successful events follows Binomial distribution $Bin\left(n, \left(\frac{t_e.MPI_L}{n}\right)^m\right)$, whose expectation is $n \times \left(\frac{t_e.MPI_L}{n}\right)^m$. Let $k = n \times \left(\frac{t_e.MPI_L}{n}\right)^m$, we have $t_e.MPI_L = k^{\frac{1}{m}} \times n^{1-\frac{1}{m}}$. Since the tuples in PT are arranged in the order of round-robin retrieval on the sorted lists $L_1, L_2, \ldots, L_M$, GPR retrieves at most $M \times k^{\frac{1}{m}} \times n^{1-\frac{1}{m}}$ tuples in PT.

Theorem 5.6. *Algorithm 1 is sublinear in I/O.*

Proof 5.6. Since Algorithm 1 retrieves at most $M \times k^{\frac{1}{m}} \times n^{1-\frac{1}{m}}$ PT tuples, it requires $O(n^{1-\frac{1}{m}})$ I/Os. Evidently, Algorithm 1 is sublinear in I/O.

Theorem 5.6 proves that Algorithm 1 is sublinear in I/O. Below we analyze the size of the candidate set S_C by the definition of skyline layer, as stated in Definition 5.2.

Definition 5.2. (*Skyline layer*) Given the table T, the skyline criteria AS_{sky} and the size parameter k, the jth skyline layer $SL_j (1 \leqslant j \leqslant k)$ consists of the skyline tuples of $T - \bigcup_{1 \leqslant i \leqslant j-1} SL_i$.

Given the tuple number n and the size m of skyline criteria, the expected number s of skyline tuples under Assumption 5.1 is $\frac{((\ln n)+\gamma)^{m-1}}{(m-1)!}$ [6,7], where γ is the Euler-Mascheroni constant (approximately 0.5772). By the formula above, $|SL_j| = \frac{\left(\ln(n - \sum_{1 \leqslant i \leqslant j-1} |SL_i|) + \gamma\right)^{m-1}}{(m-1)!}$. Since any tuples, which have no more than j parents, must be in the first $(j+1)$ skyline layers $(0 \leqslant j \leqslant k-1)$, the size of the candidate tuples can be bounded by $\sum_{1 \leqslant j \leqslant k} |SL_j|$.

Time complexity of phase 1. The time complexity of phase 1 is determined by the nested loop in Algorithm 1, the outer loop from line 3 to line 31 and the inner loop from line 14 to line 27. It is proved in Theorem 5.5 that Algorithm 1 retrieves at most $M \times k^{\frac{1}{m}} \times n^{1-\frac{1}{m}}$ PT tuples. The iteration count of the outer loop is $O(n^{1-\frac{1}{m}})$, since time complexity is the amount of time taken by an algorithm to run as a function of the input size [2]. The inner loop involves one sequential scan on S_C, whose size is no more than $\sum_{1 \leqslant j \leqslant k} |SL_j|$. For each iteration in the inner loop, the operations take in constant time. Thus the time complexity of the inner loop is $O((\ln n)^{m-1})$. On the whole, the time complexity of phase 1 is $O(n^{1-\frac{1}{m}} \times (\ln n)^{m-1})$.

5.2. Phase 2: searching for G-Skyline groups

In phase 2, GPR utilizes the candidate tuples in S_C to compute G-Skyline groups. Section 5.2.1 proposes the basic group-wise framework of phase 2, whose execution cost is analyzed in Section 5.2.2. Section 5.2.3 devises the SR strategy to speed up the execution significantly.

5.2.1. Basic framework

First, we give the definition of unit group in Definition 5.3.

Definition 5.3 (*Unit group*). $\forall s \in S_C$, the tuple s and its parents form the unit group of s, i.e. $ug(s) = \{s\} \cup pg(s)$.

Given the candidate tuples in S_C and their corresponding unit groups, $\forall s \in S_C$, if s is contained in a G-Skyline group G, we have $ug(s) \subseteq G$ according to Theorem 5.1. The definition of G-Skyline can be rephrased as a combinatorial problem with the specified restriction. Consequently, given the candidate tuples in S_C, computation of G-Skyline groups can be formally stated in another way, as listed below.

Given the candidate tuples in S_C, a G-Skyline group is a union set of unit groups of a subset in S_C such that the size of the union set is equal to k.

The computation in phase 2 is to efficiently select the subsets $\{s_1, s_2, \ldots, s_j\}$ of S_C such that $|\bigcup_{1 \leqslant b \leqslant j} ug(s_b)| = k$, and return the union sets of the unit groups of the subsets.

Given the size parameter k, a naive method is to enumerate all the j-subsets ($1 \leqslant j \leqslant k$) of S_C, compute the union sets of their unit groups, report the union sets of size k as G-Skyline groups. Obviously, the naive method is expensive due to the time complexity of $O(|S_C|^k)$, especially when $|S_C|$ is relatively great.

In this part, we first devise a basic framework of phase 2, whose pseudo-code is depicted in Algorithm 2.

Algorithm 2: GPR_Phase2

Input: the candidate tuples S_C
Output: the G-Skyline groups
1: sort the tuples of S_C in the descending order of parent sizes.
// skyline optimization
2: direct generation of k-subsets of skyline tuples.
3: initialize root v_{rt} of $\mathscr{E}$ with $v_{rt}.idx \leftarrow 0$ and $v_{rt}.gr \leftarrow \{\}$
// Tte results are outputted directly in the procedure.
4: Depth_First_Explore(v_{rt})

Procedure 3: Depth_First_Explore(Node v)

```
1: if |v.gr| ⩾ k then
2:    output v.gr as a G-Skyline group if |v.gr| = k
3: else
4:    for a ← v.idx + 1 to |S_C| do
5:       if S_C(a) ∉ v.gr and extra_condition then
6:          create a child node v_cld of v with v_cld.idx ← a and v_cld.gr ← v.gr ∪ ug(S_C(a)).
7:          Depth_First_Explore(v_cld)
8:       end if
9:    end for
10: end if
```

At the beginning of phase 2, GPR sorts the tuples in S_C in the descending order of the numbers of their parents (line 1 in Algorithm 2). In this way, the candidate tuples with the maximum parent numbers are put at the front of S_C, while the skyline tuples (no parents) are placed at the end of S_C.

The overall processing of the basic framework is to *explore set enumeration tree $\mathscr{E}$ of the candidate tuples in depth-first fashion*, which practically corresponds to an efficient way to generate the desired subsets.

Each node v in $\mathscr{E}$ has three fields $(idx, gr, tailset)$.

- *idx* the positional index of the tuple, kept by v, in S_C.
- *gr* represents the union set of unit groups of tuples kept by the nodes in the path from the root to v.
- *tailset* corresponds to the potential extension of v in the next level, i.e. the tuples in S_C following $S_C(idx)$.

It should be noted that, *tailset* is not actually materialized for each node, which can be inferred by the value of *idx*.

Let v_{rt} be the root node of $\mathscr{E}$. Initially, the root node v_{rt} is created with $idx = 0, gr = \{\}$ and $tailset = S_C(1, \ldots, |S_C|)$ (line 3 in Algorithm 2). The depth-first exploration of $\mathscr{E}$ starts from v_{rt} (line 4 in Algorithm 2). The pseudo-code of the depth-first exploration is listed in Procedure 3.

Let $v(idx, gr, tailset)$ be the currently explored node.

- If $|v.gr| \geqslant k$, there is no need to expand v to the next level and the depth-first exploration continues to deal with the follow-up nodes (line 1 in Procedure 3). Of course, if $|v.gr| = k$, $v.gr$ is reported as a G-Skyline group (line 2 in Procedure 3).
- If $|v.gr| < k$, v is expanded by $S_C(v.idx+1), \ldots, S_C(|S_C|)$ sequentially in the next level (line 4 in Procedure 3). Let $S_C(a)(v.idx+1 \leqslant a \leqslant |S_C|)$ be the currently considered tuple for expanding v. If $S_C(a)$ is contained in $v.gr$, the expansion by $S_C(a)$ does not increase the size of $v.gr$, so there is no need to expand v by $S_C(a)$ (line 5 in Procedure 3). Otherwise, v is expanded by $S_C(a)$ to generate child node v_{cld} in the next level, where $v_{cld}.idx = a$, $v_{cld}.gr = v.gr \cup ug(S_C(a))$, and $v_{cld}.tailset$ corresponds to $S_C(a+1, \ldots, |S_C|)$ (line 6 in Procedure 3). The processing of v_{cld} is performed as above recursively (line 7 in Procedure 3).

The basic framework is superior to the naive enumeration because a large proportion of the depth-first exploration space can be pruned.

Skyline optimization. As stated in [34,36], the skyline tuples contribute more to the computation of G-Skyline groups, and skyline optimization can be utilized in the basic framework. The G-Skyline groups, consisting of only skyline tuples, can be generated directly without the depth-first exploration (line 2 in Algorithm 2). Because of the skyline optimization, the basic framework does not consider the expansions of the root node by the skyline tuples. At line 5 in Procedure 3, the *extra_condition* "$!((v.idx = 0)\ and\ (|ug(S_C(a))| = 1))$" should be added to avoid duplicate G-Skyline groups.

Example 5.2. Fig. 4 illustrates the execution of basic framework for the running example. The first step is to sort the candidate tuples in S_C in the descending order of numbers of their parents. $PT(8)$ has two parents, which is put in the first position. $PT(4), PT(7)$ and $PT(9)$ have one common parent, and they are put in the middle positions. $PT(5), PT(1)$ and $PT(3)$ are skyline tuples with no parents, and they are put in the last positions. The candidate tuples in S_C can also be referred by their positions $(S_C(1), \ldots, S_C(7))$. The depth-first exploration in set enumeration tree $\mathscr{E}$ is depicted in Fig. 4. Initially, the root node v_{rt} is constructed. It can be expanded by the first four candidate tuples, while the remaining three tuples are not used to expand v_{rt} due to the skyline optimization. When v_{rt} is expanded by $S_C(1)$ to generate the child node v_1, $v_1.gr$ can be reported directly as a G-Skyline group given $k = 3$. When v_{rt} is expanded by $S_C(2)$ to generate the child node v_2, v_2 needs to be expanded further since $|v_2.gr| < 3$. Then v_2 can be expanded by $S_C(3), S_C(4), S_C(6)$ and $S_C(7)$ to generate v_{23}, v_{24}, v_{26} and v_{27} respectively, whose grs are reported as G-Skyline groups. $S_C(5)$ is not used to expand v_2 since it is contained in $v_2.gr$. The processing for v_3 and v_4 can be executed similarly. The basic framework can generate all the G-Skyline groups.

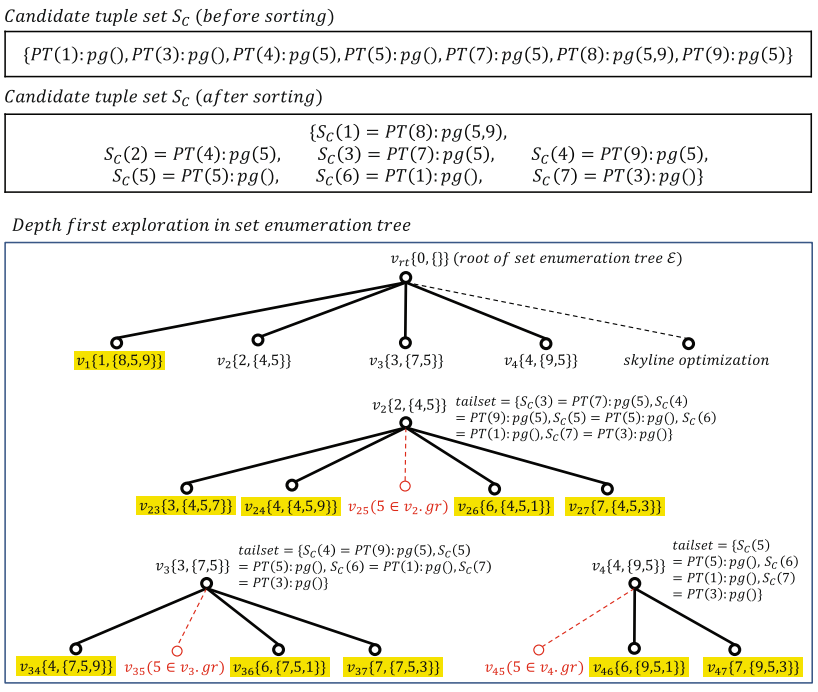


Fig. 4. The illustration of basic framework in phase 2.

5.2.2. The analysis of basic framework

This part analyzes the execution cost of basic framework, which identifies the performance bottleneck of phase 2.

As described in Section 5.2.1, the tuples in S_C are sorted at the beginning of phase 2. Consequently, the tuples in S_C can be split into k continuous sets, $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$, where $B_j (0 \leqslant j \leqslant k-1)$ represents the candidate tuples with j parents. Clearly the tuples in B_0 correspond to the skyline tuples.

In phase 2, GPR expands the root node v_{rt} of $\mathscr{E}$ with the candidate tuples in S_C one by one. For each explored child node of v_{rt}, it can also be expanded with the corresponding tail set recursively.

The number of the leaf nodes in $\mathscr{E}$ is used to measure the execution cost, since it determines the size of the explored tree and *corresponds to the number of the combinations generated from S_C actually*. Note that, when discussing the execution cost in this part, we do not consider the cost of generate k-subsets of the tuples in B_0, because this cost is constant and equal for all the G-Skyline algorithms which adopt skyline optimization.

$\forall s \in B_j (2 \leqslant j \leqslant k-1)$, let v be the child node of v_{rt} expanded by s. Given the potential extension *v.tailset* of v, the processing on v is actually to select the subsets from *v.tailset*, which can add $k-(j+1)$ new candidate tuples into $v.gr$. Let $\mathscr{E}_v$ be the subtree rooted at v. Based on the execution in basic framework, the number of the leaf nodes in $\mathscr{E}_v$ cannot exceed $\binom{\sum_{0 \leqslant b \leqslant j} |B_b|}{k-(j+1)}$. Accordingly, the execution cost $COST_j$ of processing the tuples in B_j is no more than $|B_j| \times \binom{\sum_{0 \leqslant b \leqslant j} |B_b|}{k-(j+1)}$. Obviously, the execution cost of processing B_j is grossly overestimated, since a large proportion of nodes of $\mathscr{E}_v$ are pruned during the depth-first exploration.

$\forall s_1 \in B_1$, let v_1 be the child node of v_{rt} expanded by s_1. Because $|v_1.gr| = 2$, GPR has to find extra $(k-2)$ candidates from $v_1.tailset$. Neglecting the remaining tuples in B_1 after s_1, we only focus on the tuples in B_0. The subtree, rooted at v_1, has at least $\binom{|B_0|-1}{k-2}$ leaves. Except for the parent of s_1, any of $(|B_0|-1)$ tuples in B_0 can be selected to $v_1.gr$. The execution cost $COST_1$ for processing B_1 has a lower bound of $|B_1| \times \binom{|B_0|-1}{k-2}$.

Next, we compare $COST_1$ and $COST_j (2 \leqslant j \leqslant k-1)$ to determine the performance bottleneck of basic framework. According to the description in Section 5.1, we have $\sum_{0 \leqslant b \leqslant j} |B_b| \leqslant \sum_{1 \leqslant b \leqslant j+1} |SL_b|$. Given that $COST_j \leqslant |B_j| \times \binom{\sum_{0 \leqslant b \leqslant j} |B_b|}{k-(j+1)} \leqslant |B_j| \times \binom{\sum_{1 \leqslant b \leqslant j+1} |SL_b|}{k-(j+1)}$ $(2 \leqslant j \leqslant k-1)$, $COST_1 \geqslant |B_1| \times \binom{|B_0|-1}{k-2}$, we also need to compare the relative size of $|B_j|$ and $|B_1|$. Here we provide an intuitive idea to make a comparison. $\forall s \in \bigcup_{1 \leqslant j \leqslant k-1} B_j$, the number of tuples dominating s are those which lie in the left-bottom hypercube, formed by the point $(1, 1, \ldots, 1)$ and $(s.PI_1, s.PI_2, \ldots, s.PI_m)$. For a specified s, the probability of any tuple locating in left-bottom hypercube is $\prod_{1 \leqslant i \leqslant m} \left(\frac{s.PI_i}{n}\right)$ under Assumption 5.1, and the probability $prob_{j'}$ that the left-bottom hypercube contains j' tuples is $\left(\prod_{1 \leqslant i \leqslant m} \left(\frac{s.PI_i}{n}\right)\right)^{j'} (1 \leqslant j' \leqslant k-1)$. It is expected that $prob_1 \geqslant prob_j$ and $|B_1| \geqslant |B_j|$ $(2 \leqslant j \leqslant k-1)$. By the analysis above, $COST_j \leqslant |B_1| \times \binom{\sum_{1 \leqslant b \leqslant j+1} |SL_i|}{k-(j+1)}$ $(2 \leqslant j \leqslant k-1)$ and $COST_1 \geqslant |B_1| \times \binom{|SL_1|-1}{k-2}$. We can compare the values of $COST_1$ and $COST_j$. Let $COST_j^{ub} = |B_1| \times \binom{\sum_{1 \leqslant i \leqslant j+1} |SL_i|}{k-(j+1)}$ and $COST_1^{lb} = |B_1| \times \binom{|SL_1|-1}{k-2}$. Given that $2 \leqslant j \leqslant k-1, 0 \leqslant k-(j+1) \leqslant k-3$, the upper bound of $COST_j$ is significantly lower than the lower bound of $COST_1$. For example, on the data set generated under Assumption 5.1, the values of $\frac{COST_j^{ub}}{COST_1^{lb}}$ are listed in Table 2 and Table 3. It is observed that the overwhelming majority of the execution time of G-Skyline computation is consumed by processing B_1. Note that, the value $\frac{COST_2^{ub}}{COST_1^{lb}}$ at $m=2$ in Table 3 is close to 1 because of the radical overestimation of $COST_2$ and the relatively small number of skyline tuples.

<u>*Time complexity of phase 2*</u>. The execution of phase 2 is described in Algorithm 2. Obviously, the cost of the first statement in Algorithm 2, i.e. sorting candidate tuples, is insignificant compared with the cost of the subsequent operations. We do not discuss the cost of skyline optimization here, since (1) the cost is constant and equal for all the G-Skyline algorithms which adopt skyline optimization, (2) if the skyline cardinality is relatively great, direct subset generation of skyline tuples often dominates the overall execution cost that it is difficult to distinguish the performance of the different algorithms, so the skyline optimization usually needs special treatment. Hence the execution cost of phase 2 is determined by the depth-first exploration. In order to analyze the time complexity of phase 2, we first compute $COST_1$. Likewise, we use the number of the leaf nodes generated in processing the tuples of B_1 to measure $COST_1$. Due to the execution mode, $COST_1 \leqslant |B_1| \times \binom{|B_1|+|B_0|}{k-2} \leqslant |SL_1| \times \binom{|SL_1|+|SL_1|}{k-2}$, where $|SL_1| = \frac{((\ln n)+\gamma)^{m-1}}{(m-1)!}$ [6,7]. Since the upper bound of $COST_j$ $(2 \leqslant j \leqslant k-1)$ is significantly lower than the lower bound of $COST_1$, the time complexity of phase 2 is $O\left(\left[(\ln n)^{m-1}\right]^{k-1}\right)$.

Because the processing of B_1 is the performance bottleneck of the overall processing in phase 2, it is important to find a better method to deal with the case separately.

Table 2
$COST_j/COST_1$ with varying group sizes ($n = 10^8$, m = 4).

Group size	$\frac{COST_5^{ub}}{COST_1^{lb}}$	$\frac{COST_4^{ub}}{COST_1^{lb}}$	$\frac{COST_3^{ub}}{COST_1^{lb}}$	$\frac{COST_2^{ub}}{COST_1^{lb}}$
3	-	-	-	8.76E-4
4	-	-	1.54E-6	0.0035
5	-	4.05E-9	1.39E-5	0.01
6	1.42E-11	6.5E-8	8.35E-5	0.028

Table 3
$COST_j/COST_1$ with varying skyline criteria sizes ($n = 10^8$, k = 5).

Skyline criteria size	$\frac{COST_4^{ub}}{COST_1^{lb}}$	$\frac{COST_3^{ub}}{COST_1^{lb}}$	$\frac{COST_2^{ub}}{COST_1^{lb}}$
2	0.0014	0.079	0.93
3	1.06E-6	5.76E-4	0.069
4	4.05E-9	1.39E-5	0.011
5	3.76E-11	6.12E-7	0.0022
6	6.84E-13	4.23E-8	5.82E-4

5.2.3. The SR strategy

This part optimizes the processing of B_1 to speed up G-Skyline computation considerably.

First of all, the reason why the processing is so expensive is explained as follows. $\forall s \in B_1$, let v be the child node of the root node expanded by s. Because $|v.gr| = 2$, GPR has to find extra $(k-2)$ candidate tuples from $v.tailset$, and the subtree $\mathscr{E}_v$ rooted at v has a height [2] of $(k-2)$. During the depth-first exploration, let v_{cur} be the currently explored node whose $v_{cur}.tailset$ includes only candidates in B_0. The remaining operation for the subtree, rooted at v_{cur}, actually corresponds to generate $(k - |v_{cur}.gr|)$-subsets of B_0. Obviously, the subset generation operation will be invoked repeatedly. Especially if $v_{cur}.gr$ contains only two elements, $(k-2)$-subsets of B_0 have to be generated for $|B_1|$ times. The expensive operation is performed over and over again during the processing of B_1, which incurs the high execution cost.

Example 5.3. As depicted in Fig. 5, the leftmost part of $\mathscr{E}$ includes the subtrees rooted at the tuples in $B_{k-1}, \ldots, B_2$. The other branches of $\mathscr{E}$ represent the expansions of the root node v_{rt} by the tuples $\{s_1^1, s_1^2, \ldots, s_1^{|B_1|}\}$ in B_1, respectively. Evidently, for each of these $|B_1|$ child nodes of v_{rt}, $(k-2)$ subsets of B_0 are generated repeatedly. For the rest parts of $\mathscr{E}_1, \ldots, \mathscr{E}_{|B_1|-1}$, they involve the repeated generation of $(k-3)$ subsets, $(k-4)$ subsets, ..., 1 subsets of B_0.

Inspired by this observation, **S**ubset **R**euse strategy (SR strategy) is developed to process the tuples in B_1 quickly. Rather than the method used in the basic framework, i.e. enumerating the tuples in B_1 first and then combining the subsets of B_0, SR strategy adopts the reverse order. *It enumerates the subsets of B_0 first, and then combines the required tuples in B_1 to acquire the G-Skyline groups directly.*

The SR strategy is illustrated in Fig. 6. In the SR strategy, when the basic framework begins to deal with the tuples in B_1, a sequential scan on B_1 is started. An empty hashtable HT is initialized first. For each $s \in B_1$, whose only parent is r ($r \prec s$), SR strategy checks whether HT has an element with $Key = r$. If there is no such element, a new element ($Key = r, Value = \{s\}$) is put into HT. Otherwise, s is added into $Value$ of the element. When the sequential scan on B_1 is over, HT keeps the information of B_1 in the form of hashtable structure. For the tuples in B_1 which are dominated by the same tuple in B_0, they will be maintained in the same element in HT.

The G-Skyline groups generated in the processing of B_1 can be divided into $(k-1)$ types: the G-Skyline groups including j tuples in B_0 ($1 \leqslant j \leqslant k-1$). The groups can be generated one type at a time.

The SR strategy is executed in $(k-2)$ passes. In the jth pass ($1 \leqslant j \leqslant k-2$), j-subsets of tuples in B_0 are generated first. For each j-subset $(s_0^1, s_0^2, \ldots, s_0^j)$ of B_0, the candidate tuples in B_1, which are dominated by one of the j tuples, can be obtained by j searching operations in HT (each search operation takes O(1) time). Let DS_j be the set maintaining the dominated tuples of B_1 by the j-subset. The $(k-j)$-subsets of DS_j are enumerated, each of which can be combined with $(s_0^1, s_0^2, \ldots, s_0^j)$ to get a G-Skyline group, as proved in Theorem 5.7. The subset generation results of the tuples in B_0 can be reused, and the corresponding G-Skyline groups can be acquired directly by the information obtained from efficient search operations in HT.

Theorem 5.7. *Given the j-subset $(s_0^1, s_0^2, \ldots, s_0^j)$ of B_0 and the tuple set DS_j of B_1 dominated by the j-subset, any $(k-j)$-subset of DS_j can be merged with $(s_0^1, s_0^2, \ldots, s_0^j)$ to generate a G-Skyline group of size k.*

[2] The height of a tree is the number of edges on the longest downward path between the root and a leaf.

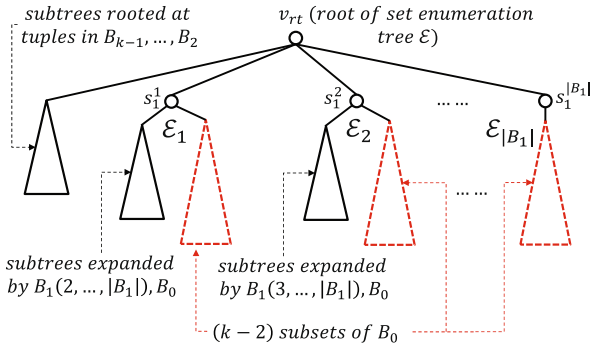


Fig. 5. The illustration of repeated generation of subsets in B_0.

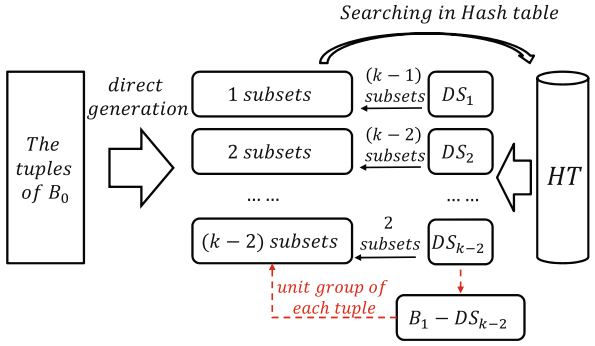


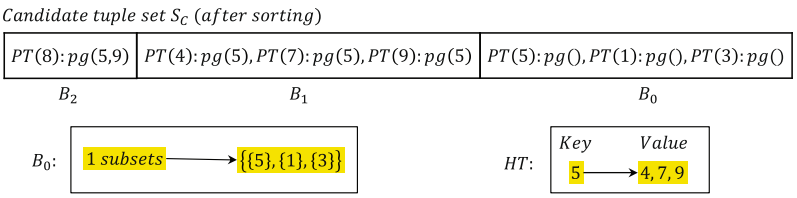
Fig. 6. The illustration of the SR strategy.

Proof 5.7. For each tuple $s_1^a (1 \leqslant a \leqslant k-j)$ in any $(k-j)$-subset of $DS_j, \exists s_0^b (1 \leqslant b \leqslant j), s_0^b$ is the only parent of s_1, and $ug(s_1^a) = \{s_1^a, s_0^b\}$. Consequently, $\left(\bigcup_{1 \leqslant a \leqslant k-j} ug(s_1^a)\right) \cup \{s_0^1, \ldots, s_0^j\} = \{s_1^1, \ldots, s_1^{k-j}, s_0^1, \ldots, s_0^j\}$, which is a G-Skyline group of size k according to Theorem 5.1.

The processing for the (k - 1)-type. For the G-Skyline groups including $(k-1)$ tuples of B_0, if we directly utilize the method mentioned above, the $(k-1)$-subsets of B_0 have to be generated first. The number of the $(k-1)$-subsets of B_0 is $\binom{|B_0|}{k-1}$, which is large. Therefore, the SR strategy utilizes the $(k-2)$-subsets generated in the $(k-2)$th pass directly to generate the G-Skyline groups of $(k-1)$-type. For the $(k-2)$-subset $(s_0^1, s_0^2, \ldots, s_0^{k-2})$ of B_0, we just need to find the tuples of B_1, *which are not dominated by any tuples in the* $(k-2)$*-subset*. Evidently, each of those tuples of B_1 can be combined with the $(k-2)$-subset of B_0 to form a G-Skyline group. Note that, in the $(k-2)$th pass of the SR strategy, DS_{k-2} maintains the tuples of B_1 dominated by one of tuples in the $(k-2)$ subset of B_0, we just scan the tuples of B_1 sequentially to find those not contained in DS_{k-2}. $\forall s \in B_1$, since s is only dominated by one tuple in B_0, the possibility that its parent is not among the $(k-2)$-subset should be relatively high. The majority of the retrieved tuples in B_1 can be combined with the $(k-2)$-subset to generate the G-Skyline groups.

When generating the G-Skyline groups from the tuples of B_1, the SR strategy only requires one-time cost of j-subset generation for tuples in B_0 $(1 \leqslant j \leqslant k-2)$, which reduces the expensive cost of subset generation effectively.

Example 5.4. Fig. 7 illustrates the execution of SR strategy in the running example. The tuples of B_1 are kept in *HT*. Since the three tuples $\{PT(4), PT(7), PT(9)\}$ have the same parent $PT(5)$, there is one element $(Key = 5, Value = \{4,7,9\})$ in *HT*. Given $k = 3$ in the running example, the SR strategy only has one pass. In the first pass, 1-subsets $\{\{5\}, \{1\}, \{3\}\}$ of B_0 are generated. For $\{5\}, (Key = 5, Value = \{4,7,9\})$ in *HT* is returned. We have $DS_1^5 = \{4,7,9\}$, whose 2-subsets are generated and combined with $\{5\}$. The three G-Skyline groups of type 1 are reported. Also for $\{5\}, B_1 - DS_1^5 = \varnothing$, no G-Skyline groups of type 2 can be generated. For $\{1\}$, there is no corresponding element in *HT* and $DS_1^1 = \varnothing$, they cannot be used to generate



GSkyline groups with 1 tuples in B_0

1 subset of B_0	DS_1	2 subset of DS_1	GSkyline Groups
{5}	{4,7,9}	{{4,7},{4,9},{7,9}}	{{5,4,7},{5,4,9},{5,7,9}}
{1}	∅	∅	∅
{3}	∅	∅	∅

GSkyline groups with 2 tuples in B_0

1 subset of B_0	DS_1	$B_1 - DS_1$	1 subset of $(B_1 - DS_1)$	GSkyline Groups
{5}	{4,7,9}	∅	∅	∅
{1}	∅	{4,7,9}	{{4},{7},{9}}	{{1,4,5},{1,7,5},{1,9,5}}
{3}	∅	{4,7,9}	{{4},{7},{9}}	{{3,4,5},{3,7,5},{3,9,5}}

Fig. 7. The illustration of execution of SR strategy in the running example.

G-Skyline groups of type 1. Also for $\{1\}, B_1 - DS_1^1 = \{4,7,9\}$, whose unit groups of 1-subsets are generated and combined with $\{1\}$. The three G-Skyline groups of type 2 are reported. The processing for $\{3\}$ is similar to that for $\{1\}$ and can also return three G-Skyline groups of type 2.

Discussion about the size of DS_j. In the jth pass ($1 \leqslant j \leqslant k-2$), $(k-j)$-subsets of DS_j have to be generated. The size of $|DS_j|$ also needs to be discussed. According to Definition 5.2, $|B_0| = |SL_1| > |SL_2| \geqslant |B_1|$, the last inequality holds since some tuples in SL_2 are dominated by more than one skyline tuples. For each tuple s_0 in B_0, the average number of the tuples in B_1 dominated by s_0 is $\frac{|B_1|}{|B_0|}$, which is less than 1. For example, given the data set of $n = 10^8, m = 4$ and $k = 4$ under Assumption 5.1, the average size of DS_1 is 0.8257 and the average size of DS_2 is 1.6514. Correspondingly, the average size of DS_j should be no more than j ($1 \leqslant j \leqslant k-2$), which is limited by the value of k. The value of k normally is relatively small, since the number of the G-Skyline groups can be much high with the increasing value of k. For example, given the data set of $n = 10^8$ and $m = 4$ under Assumption 5.1, the number of G-Skyline groups increases from 2.59×10^8 at $k = 3$ to 1.72×10^{13} at $k = 5$.

$\forall s \in B_1$, let v be the child node of the root v_{rt} expanded by s, and $\mathscr{E}_v$ be the subtree rooted at v. The process of generating G-Skyline groups in the SR strategy can be seen as the direct generation of the leaf nodes of G-Skyline groups in $\mathscr{E}_v$, which requires only one-time cost of subset generation of the tuples in B_0. The execution cost saved by SR strategy can be computed as below.

- In the jth pass ($1 \leqslant j \leqslant k-2$), the SR strategy generates j-subsets of the tuples in B_0, and then each j-subset is combined with $(k-j)$ subsets of DS_j. The reduced number of the explored leaf nodes in this part is $\sum_{1 \leqslant j \leqslant k-2} \left\{ \binom{|B_0|}{j} \times \left[\binom{|DS_j|}{k-j} - 1 \right] \right\}$.
- In the $(k-2)$th pass, the SR strategy also needs to find the tuples of B_1 that are not dominated by the $(k-2)$ subsets during the sequential scan on B_1. The reduced number of the explored nodes in this part is $\binom{|B_0|}{k-2} \times (|B_1| - |DS_{k-2}| - 1)$.

Therefore, the reduced cost of processing the tuples in B_1 by the SR strategy is $\sum_{1 \leqslant j \leqslant k-2} \left\{ \binom{|B_0|}{j} \times \left[\binom{|DS_j|}{k-j} - 1 \right] \right\} + \binom{|B_0|}{k-2} \times (|B_1| - |DS_{k-2}| - 1)$. Below we discuss the time complexity of the SR strategy. The execution cost of processing tuples in B_1 actually includes two parts: (1) subset generation of tuples in B_0, (2) the generation of G-Skyline groups by combining the subsets of B_0 and the tuples of B_1. Without the SR strategy, the operation of subset generation will be repeated $|B_1|$ times. By the SR strategy, the subset generation is only be executed once, whose execution cost is reduced correspondingly. We focus on the $(k-2)$th pass of the SR strategy. In the $(k-2)$th pass, the SR strategy utilizes each generated $(k-2)$-subsets of tuples in B_0 and scans the tuples of B_1 to discover the G-Skyline groups of the $(k-1)$-type. It is known that the time complexity of big-Oh notation not only throws away low-order terms, but also throws away leading constants. Although the SR strategy does not lower the overall time complexity of phase 2, it improves the

performance of phase 2 effectively (by lowering the leading constants) and reduces the execution cost significantly, which is also verified by the analysis above and the experimental results.

6. Performance evaluation

To evaluate the performance of GPR, we implement it in Java with jdk-8u112-windows-x64. The experiments are executed on DELL Precision T3431 Workstation (Intel(R) Core(TM) i7-9700 CPU @ 3.00 GHz (8 cores) + 32G memory + 64bit windows 10). In the experiments, the performance of GPR is evaluated against the state-of-the-art existing algorithms, GMDS [34], LU [41] and PMDS [34]. Although the index-independent SP algorithm [41] has been implemented, it is not reported in the experiments due to the rather poor performance on massive data.

In the experiments, we evaluate the performance of GPR in terms of several aspects: tuple number (n), skyline criteria size (m), group size (k), table width (M) and correlation coefficient (c). The used parameter settings are listed in Table 4. Except for the tuple number, the other values in the parameter settings are commonly used in evaluating the G-Skyline algorithms [17,18,34,36,41]. The default value of m (or k) is selected because it is a reasonable trade-off point, where the number of skylines (or the number of the G-Skyline groups) is large enough to evaluate the performance of the algorithms, but not too large to make the execution time too long. The independent distribution is a common choice for the synthetic data set. The default table width of 10 is the average of a relatively small table width and a relatively large table width, which will reflect the performance of the algorithms on the table of different widths. The used tuple number here is much larger than that used in the experiments of existing algorithms, because this paper focuses on the performance of G-Skyline on massive data and thus utilizes a much larger scale of data. The real data used is HIGGS Data Set from UCI Machine Learning Repository[3]. HIGGS Data Set contains 11,000,000 tuples with 28 attributes. The first 21 features (columns 2–22) are kinematic properties measured by the particle detectors in the accelerator. The last seven features are functions of the first 21 features; these are high-level features derived by physicists to help discriminate between the two classes. On real data, we evaluate the performance of GPR with varying skyline criteria sizes.

Details of synthetic data set generation. The synthetic data sets are generated by Java's *Random* Class [29], and each attribute is generated as a *long* value. For the independent attributes, the attribute values can be assigned by $Random.nextDouble() \times Long.MAX\_VALUE$, where "*nextDouble*()" returns uniformly distributed double value in [0, 1] and $Long.MAX\_VALUE$ is $(2^{63}-1)$. For correlated distribution, the first two attributes have the specified correlation coefficient, while the left attributes follow the independent distribution. In order to generate two correlated sequences with correlation coefficient c, two uncorrelated Gaussian distributed random sequences X_1 and X_2 are generated first, then $Y_1 = c \times X_1 + \sqrt{1-c^2} \times X_2$ is generated. Two sequences X_1 and Y_1 have the given correlation coefficient c [33]. The sequences X_1 and X_2 are first generated by $Random.nextGaussian()$, which returns the next Gaussian distributed double value with mean 0.0 and standard deviation 1.0. Then X_1 and Y_1 are scaled up by multiplying the factor 2^{60}, the scale factor is selected for avoiding overflow or underflow.

Every result reported in the experiments is the average value of results in three repeated executions. Some experimental results are not included in the experiments since the execution times of these algorithms with the specified parameters are so long that we abort the execution before they terminate. In the experiments, given the total memory space of 32G, we use the command-line option "java -Xmx24G" to allocate 24G for the execution of each algorithm[4], where the flag Xmx specifies the maximum memory allocation pool for a Java Virtual Machine (JVM) [16].

The compression of skyline combination. In the experiments, the time of skyline optimization, i.e. direct generation of k-subsets of the skyline tuples, is not included in the execution time of the evaluated algorithms, although we report it separately (with the legend "SKYOPT"). For one thing, the time is the same for all the algorithms used in the experiments, which adopt skyline optimization. For another, the execution time of skyline optimization can be very high, which can even dominate the overall execution time. In this case, it is difficult to distinguish the performance of different algorithms.

The construction of presorted table. The required presorted table is constructed before the execution of GPR. Given the table T of $n = 10 \times 10^7$ and $M = 10$, the presorting operation takes 2564.296 s to build the presorted table *PT*. For the sorting operation, if the size of the file to sort is larger than allocated memory size, two-stage multiway merge-sort is utilized to sort large file in two stages. Nevertheless, it should be noted that, *PT* only needs to be constructed once, and it can be used for the G-Skyline computation with different values of k and m.

The processing of the new tuples. In typical massive data applications, new data arrives periodically and the update is usually performed in the less busy time (for example, preferably at mid-night, since it is unlikely that the data will be used then). Alternatively, the much larger *PT* remains unchanged while the new tuples are accumulated in a much smaller delta table. When the size of the delta table reaches some threshold (for example 5% of *PT*), it is merged with *PT*. Between two merging operations, phase 1 of GPR is executed on *PT* similarly except that another scan on the delta table is required when scanning on *PT* terminates. Phase 2 of GPR is exactly the same as that in this paper.

[3] https://archive.ics.uci.edu/ml/datasets/HIGGS#

[4] https://docs.oracle.com/javase/8/docs/technotes/tools/windows/java.html

Table 4
Parameter Settings.

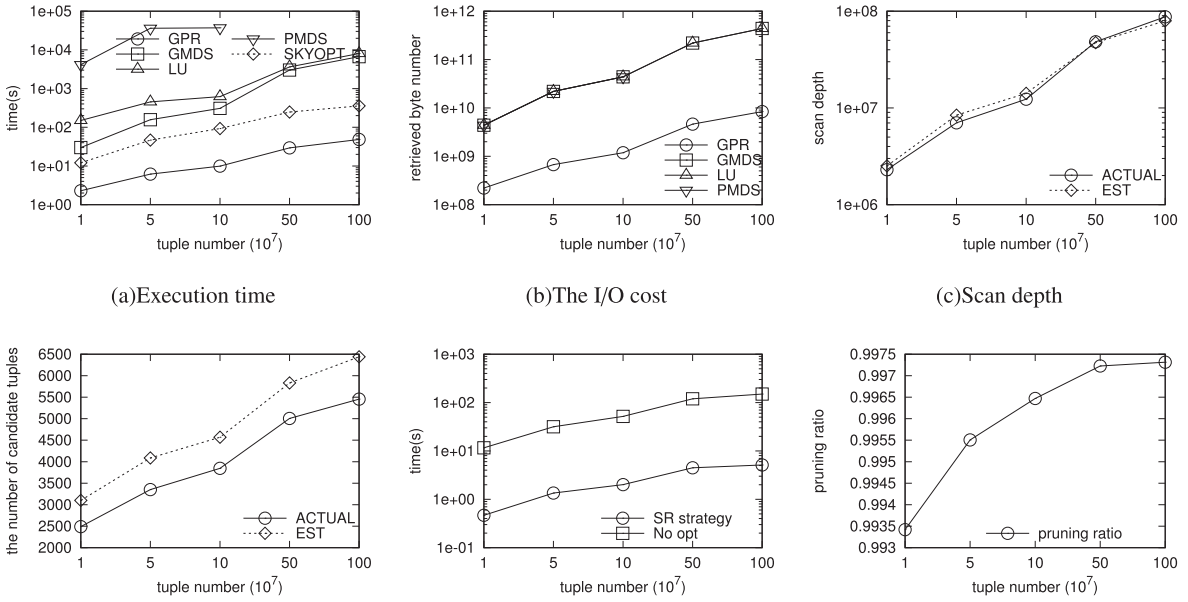
Parameter	Used values	Default value
Tuple number (10^7) (syn)	$1 \sim 100$	10
Skyline criteria size (syn)	$2 \sim 6$	4
Group size (syn)	$2 \sim 5$	4
Table width (syn)	$5 \sim 15$	10
Correlation coefficient (syn)	$-0.6 \sim 0.6$	0
Skyline criteria size (real)	$2 \sim 5$	–

6.1. Exp 1: the effect of tuple number

Given $m = 4, k = 4, M = 10$ and $c = 0$, experiment 1 evaluates the performance of GPR with varying tuple numbers. As illustrated in Fig. 8(a), the execution times of all algorithms increase considerably with a greater value of n, while GPR has the lowest rate of increase. On average, GPR runs 51.714 times faster than GMDS and 81.926 times faster than LU. For PMDS, it runs orders of magnitude slower than the other algorithms in experiment 1. Fig. 8(a) also reports the execution time of the skyline optimization, represented by the dotted line. The significant speedup ratio of GPR comes from two aspects: the much less I/O cost and computation cost. Fig. 8(b) compares the I/O costs of the algorithms. The other algorithms need to perform external sorting first, while GPR holds the early termination property. Consequently, compared with the other algorithms, GPR involves 31.619 times less I/O cost. The scan depth and the number of the maintained candidates for GPR are also depicted in Fig. 8(c) and Fig. 8(d), respectively. The actual execution results conform with the estimated values. The SR strategy helps GPR reduce its computation cost. As shown in Fig. 8(e), when processing the candidate tuples in B_1, the SR strategy needs 21.674 times less computation cost compared with the processing in basic framework. The speedup obtained by SR strategy comes from the effective reuse operation. Fig. 8(f) reports the pruning ratio of the SR strategy. The pruning ratio is measured as: $\frac{num_{removedBySR}}{num_{explored}+num_{removedBySR}}$, where $num_{explored}$ is the number of the nodes actually explored and $num_{removedBySR}$ is the number of the nodes removed by the SR strategy when processing B_1. Evidently, the exploration space is reduced significantly by the SR strategy.

6.2. Exp 2: the effect of skyline criteria size

Given $n = 10 \times 10^7, k = 4, M = 10$ and $c = 0$, experiment 2 evaluates the performance of GPR with varying skyline criteria sizes. In experiment 2, when the value of m increases from 2 to 6, the number of the skyline tuples grows exponentially, from



(a)Execution time (b)The I/O cost (c)Scan depth

(d)Candidate tuple number (e)The time of processing B_1 (f)Pruning ratio of SR strategy

Fig. 8. The effect of tuple number.

22 at $m = 2$ to 21942 at $m = 6$. Certainly, the execution costs of all the algorithms must rise correspondingly. In experiment 2, the time to perform the external sorting is 234.931 s, which remains unchanged given the fixed values of n and M. The sorting time dominates the execution times of PMDS, GMDS and LU at $m \leqslant 3$. After that, the computation costs begin to rise rapidly. As illustrated in Fig. 9(a), on average, GPR runs 296.176 times faster than GMDS and 387.895 times faster than LU. The speedup ratio of two orders of magnitude comes from the much less I/O cost and much lower computation cost. As reported in Fig. 9(b), GPR involves 318.592 times less I/O cost compared with the other algorithms. The I/O cost of GPR is determined by its scan depth, which is illustrated in Fig. 9(c). The number of *PT* tuples retrieved in phase 1 increases quickly with a greater value of m, which also conforms to our analysis. The number of the maintained candidate tuples, depicted in Fig. 9(d), displays the similar variation trend. As shown in Fig. 9(e) and Fig. 9(f), the SR strategy can reduce the exploration space significantly, and save the computation cost of processing B_1 by a order of magnitude.

6.3. Exp 3: the effect of group size

Given $n = 10 \times 10^7, m = 4, M = 10$ and $c = 0$, experiment 3 evaluates the performance of GPR with varying group sizes. As illustrated in Fig. 10(a), the execution times of GPR, GMDS and LU rise at $k \leqslant 4$, but not by much. For GPR, its execution time increases from 4.848 s at $k = 2$ to 9.942 s at $k = 4$, where the time in phase 1 increases from 4.832 s at $k = 2$ to 7.621 s at $k = 4$. For GMDS and LU, the major cost comes from the external sorting in the first step, which remains unchanged in experiment 3. After that ($k > 4$), the execution times of these algorithms grow significantly, since the computation cost of generating G-Skyline groups is much high. On the whole, GPR runs 29.976 times faster than GMDS and 39.486 times faster than LU. The significant performance advantage of GPR is the results of early termination property and SR strategy. As depicted in Fig. 10(b), GPR involves 30.136 times less I/O cost compared with the other algorithms. The scan depth and the number of the maintained candidate tuples conform with our analysis in Section 5.1, which are also shown in Fig. 10(c) and Fig. 10(d), respectively. The time of processing tuples in B_1, with or without SR strategy, is illustrated in Fig. 10(e). The SR strategy can reduce the computation cost of processing tuples in B_1 by one order of magnitude, except for $k = 2$. At $k = 2$, the SR strategy does not actually work, since every tuple in B_1 can generate a G-Skyline group with its parent. As verified in Fig. 10(f), the pruning ratio of the SR strategy is 0 at $k = 2$. However, at $k \geqslant 3$, the SR strategy achieves a very high pruning ratio (over 0.9964).

6.4. Exp 4: the effect of table width

Given $n = 10 \times 10^7, m = 4, k = 4$ and $c = 0$, experiment 4 evaluates the performance of GPR with varying table widths. When the value of M increases from 5 to 15, the table T triples in size. Obviously, for GPR, this increases the execution time in phase 1, but does not affect of the execution in phase 2. The similar observation can be found for GMDS, LU and PMDS. As

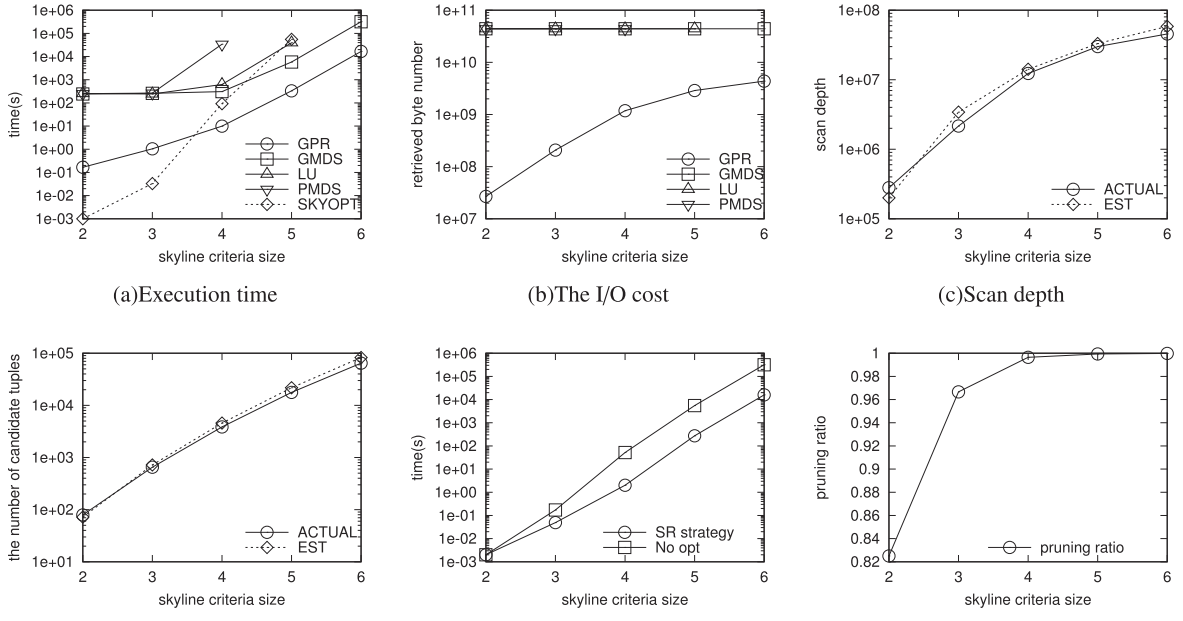
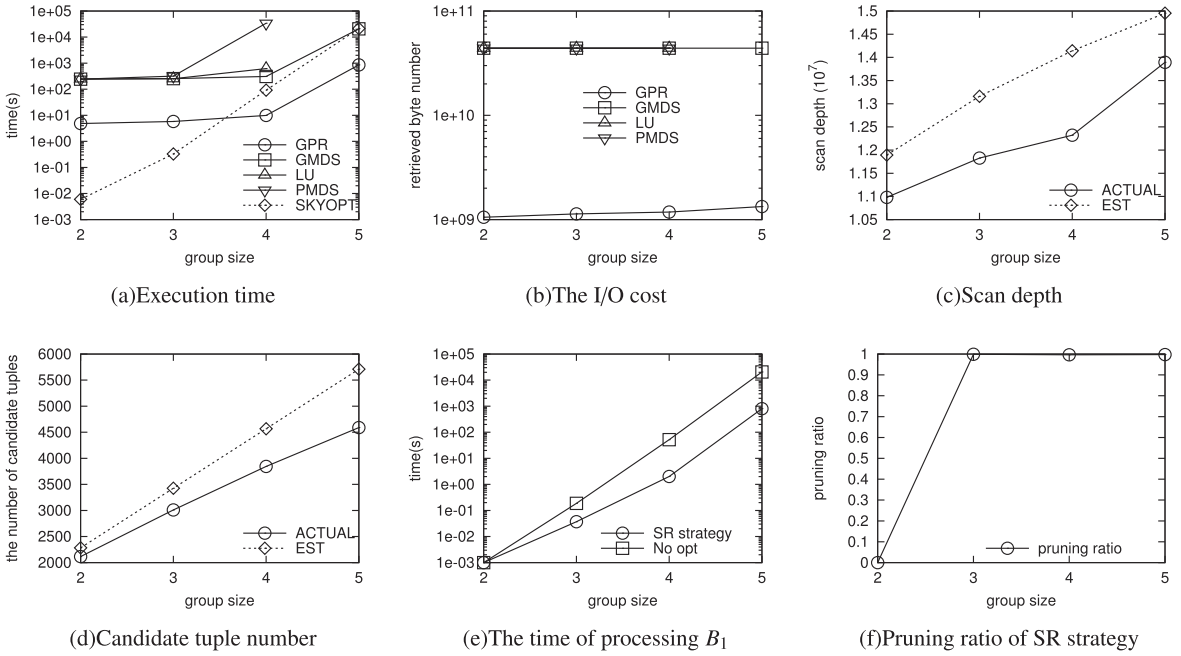


Fig. 9. The effect of skyline criteria size.



(a)Execution time (b)The I/O cost (c)Scan depth

(d)Candidate tuple number (e)The time of processing B_1 (f)Pruning ratio of SR strategy

Fig. 10. The effect of group size.

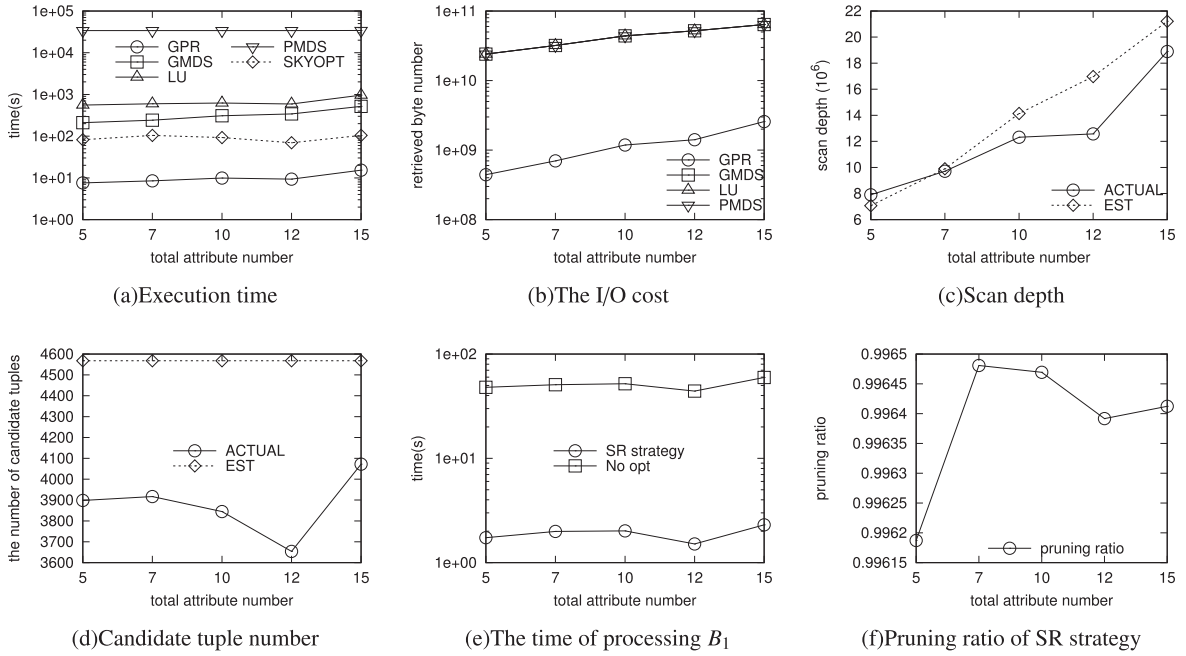
illustrated in Fig. 11(a), GPR runs 26.26 times faster than GMDS and 55.399 times faster than LU, while PMDS is orders of magnitude slower than the other algorithms. As depicted in Fig. 11(b), the involved I/O costs for all the algorithms rise correspondingly since the size of the table is tripled in experiment 4, and the I/O cost of GPR is 33.161 times less than the other algorithms. Due to the early termination property, GPR does not scan all tuples in the presorted table in phase 1. The scan depth of GPR is shown in Fig. 11(c), and the growth trend of the actual scan depth is consistent with the results of our analysis. The number of the maintained candidate tuples is estimated to be constant in experiment 4 given the fixed parameters of n, m and k, which is illustrated in Fig. 11(d). Although there are some fluctuations, the actual number of the candidate tuples conforms with the estimated results in general. Because the different values of M do not affect phase 2 of GPR, the time of processing tuples in B_1 remains unchanged basically. And just as presented in Fig. 11(e), the SR strategy can reduce the computation cost by 22.344 times. The pruning ratio of the SR strategy is also very high (above 0.996), as shown in Fig. 11 (f).

6.5. Exp 5: the effect of correlation coefficient

Given $n = 10 \times 10^7, m = 3, k = 4$ and $M = 10$, experiment 5 evaluates the performance of GPR with varying correlation efficients. In experiment 5, for describing the trends more clearly, the experimental results are depicted from the positive correlation to negative correlation. The correlation of the data set actually affects the number of the skyline tuples considerably. This can be seen from Fig. 12(d). When the value of c changes from 0.6 to -0.6, the number of the skyline tuples increases from 82 (335 candidate tuples) to 21727 (47817 candidate tuples). Evidently, this has a great impact on the execution cost of computing G-Skyline groups. Furthermore, for GPR, the correlation of the data set also influences the scan depth significantly, as shown in Fig. 12(c).

From the description above, the execution times of the algorithms should increase when the data set changes from positive correlation to negative correlation. As illustrated in Fig. 12(a), at $c \geqslant 0$, the growth trends for the algorithms are not significant. On data set of positive correlation, the computation cost is relatively small, and the time to compute the candidate tuples or build the required index, which does not change much at $c \geqslant 0$, dominates the overall execution time. At $c < 0$, on the negatively correlated data, the execution times of the algorithms rise significantly, since the computation cost of generating G-Skyline groups begins to dominate the overall execution cost. As depicted in Fig. 12(a), GPR runs 16.351 times faster than GMDS and 21.182 times faster than LU at $c < 0$, while GPR runs two orders of magnitude faster than GMDS and LU at $c \geqslant 0$.

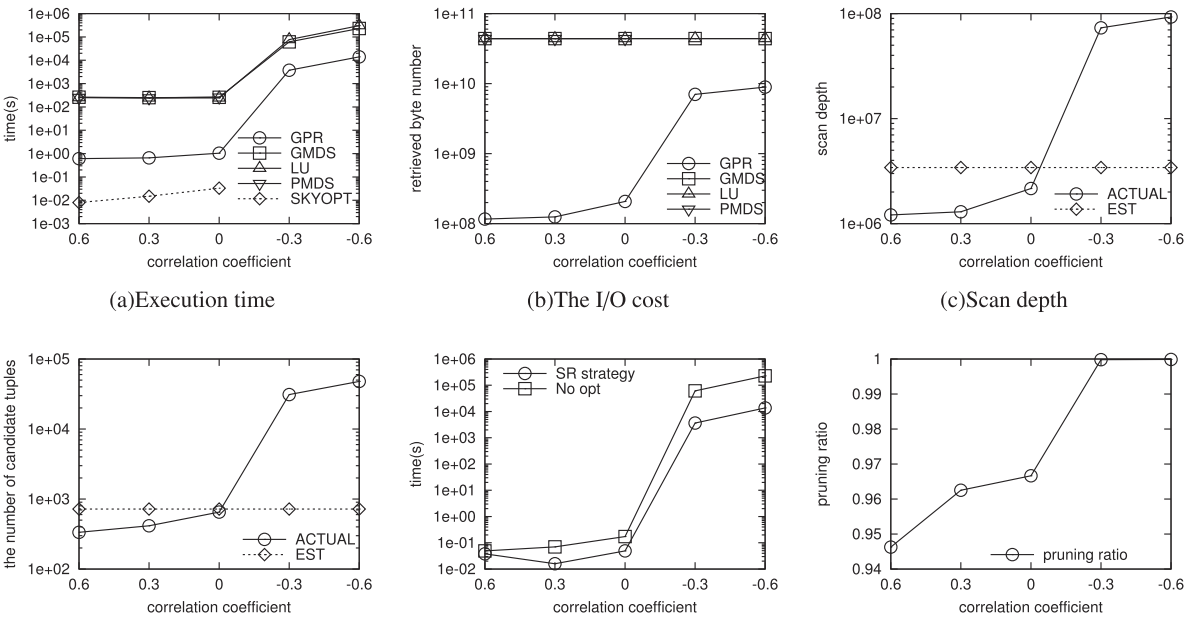
The I/O costs of the algorithms are depicted in Fig. 12(b). It shows that, GPR involves 159.17 times less I/O cost compared with the other algorithms, but the advantage shrinks gradually when the value of c changes from 0.6 to -0.6. The can be explained as follows. The I/O costs of the other three algorithms come from the external sorting on the table, which are fixed in experiment 5, while the scan depth of early termination determines the I/O cost of GPR.



(a)Execution time (b)The I/O cost (c)Scan depth

(d)Candidate tuple number (e)The time of processing B_1 (f)Pruning ratio of SR strategy

Fig. 11. The effect of table width.



(a)Execution time (b)The I/O cost (c)Scan depth

(d)Candidate tuple number (e)The time of processing B_1 (f)Pruning ratio of SR strategy

Fig. 12. The effect of correlation coefficient.

The effect of the SR strategy is illustrated in Fig. 12(e) and Fig. 12(f). The SR strategy helps reduce the computation cost of processing tuples in B_1 by a order of magnitude. The pruning ratio of the SR strategy is above 0.94 in experiment 5. The pruning ratio increases when the value of c changes from 0.6 to -0.6. The much larger size of the candidate tuples in negatively correlated data gives more opportunity to apply the SR strategy.

6.6. Exp 6: real data

The real data used in experiment 6 is HIGGS Data Set, which contains 11,000,000 tuples with 28 attributes. By setting k to 4, experiment 6 evaluates the performance of GPR on varying skyline criteria sizes (using the first several features as skyline criteria). As illustrated in Fig. 13(a), GPR runs 188.334 times faster than GMDS and 219.426 times faster than LU, while PMDS runs the slowest. The variation trend can be explained in the similar way as that in Section 6.2. With the greater skyline criteria size, more skyline tuples will be generated. Evidently, much more G-Skyline groups will be discovered, which increases the execution cost significantly. The I/O costs of the algorithms are depicted in Fig. 13(b). On average, GPR involves 206.049 times less I/O cost than the other algorithms, although the advantage gap becomes smaller with the increasing skyline criteria sizes. The trend of the I/O cost of GPR is determined by its scan depth, as shown in Fig. 13(c). The number of the candidate tuples is illustrated in Fig. 13(d). With a greater skyline criteria size, more skyline tuples and more candidate tuples are maintained by GPR. The effect of the SR strategy is demonstrated in Fig. 13(e) and Fig. 13(f). The SR strategy can reduce the overwhelming majority of exploration space, and save the computation cost by 10.927 times.

6.7. Discussion

This section evaluates the performance of GPR by comparing it with the state-of-the-art algorithms. Meanwhile, we emphasize the novelties of GPR explicitly in this part.

The normal process of the G-Skyline algorithms consists of two phases: acquiring the candidate tuples (or building the index structure) and computing G-Skyline groups. From the experimental results, GPR runs much faster than the existing algorithms. The performance advantage of GPR is due to the two novelties in the two phases.

- *Novelty in phase 1: presorted-based sublinear I/O method*. In order to build DSG or MDG, the existing algorithms require several full-table scans, which involve a high I/O cost on massive data. For GPR, the transitivity of the dominance relation is utilized to skip the unqualified tuples directly without retrieving them actually, namely GPR holds the early termination property. The sublinear I/O method reduces the I/O cost significantly
- *Novelty in phase 2: reusing the subset generation results*. As analyzed in Section 5.2.2, computation of G-Skyline groups in the existing algorithms involves a large number of repeated subset generations, which incur a high computation cost. For GPR, the SR strategy is developed to avoid the repeated subset generation by the reverse combination order of the existing algorithms. GPR first generates the subsets of the skyline tuples once, then the skillful operation is exploited to combine the required tuples to find the G-Skyline groups directly. The SR strategy reduces the computation cost considerably.

When the number of the candidate tuples (or the size of the G-Skyline groups) is relatively small, the execution time in phase 2 is small. Then the overall execution time is dominated by the time in phase 1. In this case, GPR runs much faster than

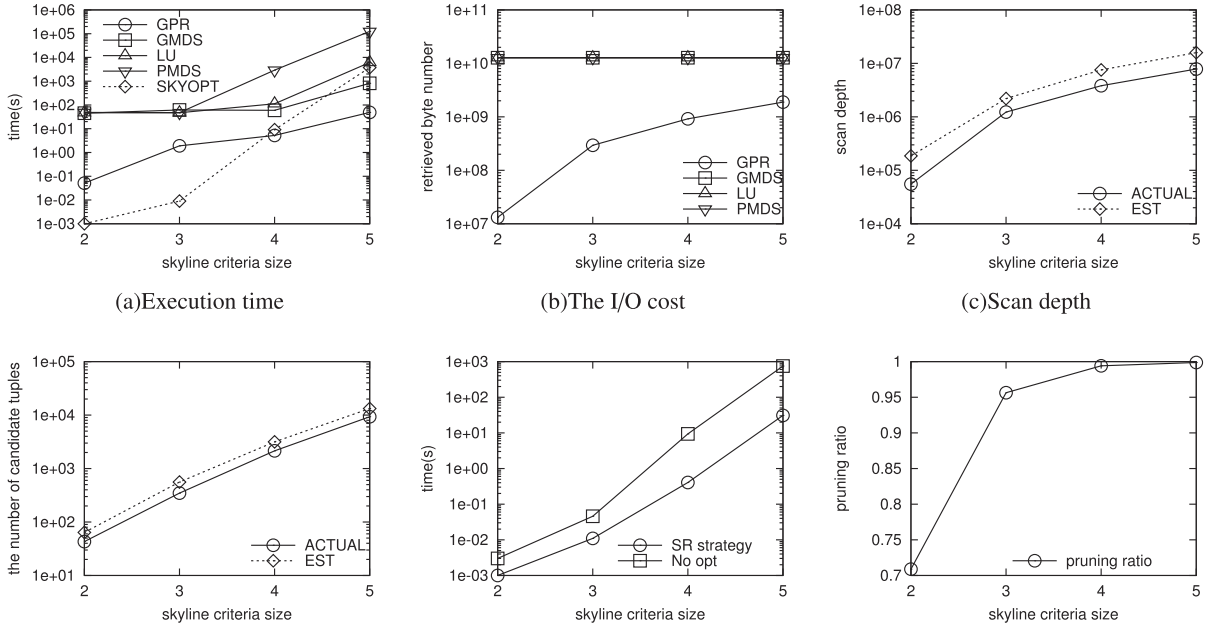


Fig. 13. The effect of real data.

the other algorithms because GPR can terminate earlier with sublinear I/Os and the other algorithms have to perform the external sorting on massive data.

Conversely, when the number of the candidate tuples (or the size of the G-Skyline groups) is relatively great, the cost in phase 2 dominates the overall execution cost. In this case, the time in phase 1 usually is insignificant, and the SR strategy can reduce the computation cost by orders of magnitude.

In a word, as analyzed in this paper and verified in experiments, GPR can compute G-Skyline groups on massive data efficiently by the proposed two novelties.

7. Conclusion

This paper considers the problem of computing G-Skyline groups on massive data. The existing G-Skyline algorithms cannot deal well with massive data. They have to perform the external sorting on massive data and expensive in-memory computation, which incur high I/O cost and high computation cost. For the performance issue of the existing algorithms, this paper devises a novel GPR algorithm to discover the G-Skyline groups on massive data efficiently, targeted at reducing the I/O cost and computation cost effectively. The objective is achieved in the execution of two phases of GPR. In phase 1, the sublinear-I/O method is presented for scanning the presorted table to acquire the candidate tuples, which is proved to hold early termination property. The scan depth and number of the candidate tuples are analyzed also. In phase 2, this paper first devises a basic framework for processing, which actually encompasses the computation mode of G-Skyline group in the existing algorithms. This paper analyzes the execution cost of the basic framework and determines its performance bottleneck. Specifically, GPR develops the SR strategy, which reuses the immediate subset generation results, to speed up the execution of basic framework considerably. The extensive experimental results, conducted on synthetic and real-life data sets, show that GPR obtains up to two orders of magnitude speedup compared with the existing algorithms.

Although G-Skyline returns the candidate set of optimal tuple groups, its major problem is that it normally generates too many G-Skyline groups. This makes it difficult for users to find the useful G-Skyline groups quickly. In the future work, we will consider how to extend GPR to discover a small number of user-preference-based high quality G-Skyline groups on massive data.

CRediT authorship contribution statement

Xixian Han: Conceptualization, Methodology, Software, Investigation, Writing - original draft. **Jinbao Wang:** Software, Visualization, Investigation. **Jianzhong Li:** Writing - review & editing. **Hong Gao:** Writing - review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. This work was supported in part by the NSFC under Grant Nos. 61872106, 61832003, 61632010.

References

[1] Jan Chomicki, Parke Godfrey, Jarek Gryz, Dongming Liang, Skyline with presorting, in: Proceedings of the 19th International Conference on Data Engineering, 2003, pp. 717–719.
[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, 3rd Edition., MIT Press, 2009.
[3] Alfredo Cuzzocrea, Panagiotis Karras, Akrivi Vlachou, Effective and efficient skyline query processing over attribute-order-preserving-free encrypted data in cloud-enabled databases, Future Gener. Comput. Syst. 126 (2022) 237–251.
[4] Fang Dong, Junzhou Luo, Jiahui Jin, Jiyuan Shi, Ye Yang, Jun Shen, Accelerating skycube computation with partial and parallel processing for service selection, IEEE Trans. Serv. Comput. 13 (6) (2020) 969–984.
[5] Weifeng Gao, Yiming Wang, Lingling Liu, Lingling Huang, A gradient-based search method for multi-objective optimization problems, Inf. Sci. 578 (2021) 129–146.
[6] Parke Godfrey, Skyline cardinality for relational processing, in: Foundations of Information and Knowledge Systems, volume 2942, Springer, Berlin/Heidelberg, 2004, pp. 78–97.
[7] Parke Godfrey, Ryan Shipley, Jarek Gryz, Algorithms and analyses for maximal vector computation, VLDB J. 16 (1) (2007) 5–28.
[8] Xixian Han, Jianzhong Li, Donghua Yang, Jinbao Wang, Efficient skyline computation on big data, IEEE Trans. Knowl. Data Eng. 25 (11) (2013) 2521–2535.
[9] Amin Hashemi, Mohammad Bagher Dowlatshahi, Hossein Nezamabadi-pour, An efficient pareto-based feature selection algorithm for multi-label classification, Inf. Sci. 581 (2021) 428–447.
[10] Hyeonseung Im, Sungwoo Park, Group skyline computation, Inf. Sci. 188 (2012) 151–169.
[11] Donald Kossmann, Frank Ramsak, Steffen Rost, Shooting stars in the sky: an online algorithm for skyline queries, Proceedings of VLDB'02 (2002) 275–286.
[12] Jongwuk Lee, Seung-won Hwang, Scalable skyline computation using a balanced pivot selection technique, Inf. Syst. 39 (2014) 1–21.

[13] Ken C.K. Lee, Wang-Chien Lee, Baihua Zheng, Huajing Li, Yuan Tian, Z-SKY: an efficient skyline query processing framework based on z-order, VLDB J. 19 (3) (2010) 333–362.
[14] Chengkai Li, Nan Zhang, Naeemul Hassan, Sundaresan Rajasekaran, and Gautam Das. On skyline groups. In 21st ACM International Conference on Information and Knowledge Management, pages 2119–2123. ACM, 2012..
[15] Helan Liang, Bincheng Ding, Du Yanhua, Fanzhang Li, Parallel optimization of qos-aware big service processes with discovery of skyline services, Future Gener. Comput. Syst. 125 (2021) 496–514.
[16] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, The Java Virtual Machine Specification, Java SE, 8 Edition., Addison-Wesley Professional, 2014.
[17] Jinfei Liu, Li Xiong, Jian Pei, Jun Luo, Haoyu Zhang, Finding pareto optimal groups: Group-based skyline, PVLDB 8 (13) (2015) 2086–2097.
[18] Jinfei Liu, Li Xiong, Jian Pei, Jun Luo, Haoyu Zhang, Yu. Wenhui, Group-based skyline for pareto optimal groups, IEEE Trans. Knowl. Data Eng. 33 (7) (2021) 2914–2929.
[19] Jinfei Liu, Juncheng Yang, Li Xiong, Jian Pei, Jun Luo, Yuzhang Guo, Shuaicheng Ma, Chenglin Fan, Skyline diagram: Efficient space partitioning for skyline queries, IEEE Trans. Knowl. Data Eng. 33 (1) (2021) 271–286.
[20] Rui Liu and Dominique Li. Efficient skyline computation in high-dimensionality domains. In Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, pages 459–462. OpenProceedings.org, 2020..
[21] Xiaoye Miao, Yunjun Gao, Su Guo, and Gang Chen. On efficiently answering why-not range-based skyline queries in road networks. IEEE Trans. Knowl. Data Eng., 30(9):1697–1711, 2018..
[22] Xiaoye Miao, Su. Yunjun Gao, Guo, Lu Chen, Jianwei Yin, Qing Li, Answering skyline queries over incomplete data with crowdsourcing, IEEE Trans. Knowl. Data Eng. 33 (4) (2021) 1360–1374.
[23] Xiaoye Miao, Yunjun Gao, Linlin Zhou, Wei Wang, Qing Li, Optimizing quality for probabilistic skyline computation and probabilistic similarity search, IEEE Trans. Knowl. Data Eng. 30 (9) (2018) 1741–1755.
[24] Michael D. Morse, Jignesh M. Patel, H.V. Jagadish, Efficient skyline computation over low-cardinality domains, in: Proceedings of the 33rd International Conference on Very Large Data Bases, ACM, 2007, pp. 267–278.
[25] Dimitris Papadias, Yufei Tao, Fu. Greg, Bernhard Seeger, Progressive skyline computation in database systems, ACM Trans. Database Syst. 30 (1) (2005) 41–82.
[26] Mukundraj V. Patil, Anand J. Kulkarni, Pareto dominance based multiobjective cohort intelligence algorithm, Inf. Sci. 538 (2020) 69–118.
[27] Zhuo Peng, Chaokun Wang, Member promotion in social networks via skyline, World Wide Web 17 (4) (2014) 457–492.
[28] Weilong Ren, Xiang Lian, Kambiz Ghazinour, Skyline queries over incomplete data streams, VLDB J. 28 (6) (2019) 961–985.
[29] Herbert Schildt, Java: The Complete Reference, Eleventh Edition., McGraw-Hill Education, 2018.
[30] Jung Hyuk Seo and Myoung Ho Kim, Finding influential communities in networks with multiple influence types, Inf. Sci. 548 (2021) 254–274.
[31] Cheng Sheng and Yufei Tao. On finding skylines in external memory. In Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '11, pages 107–116, 2011..
[32] Yufei Tao, Xiaokui Xiao, Jian Pei, Efficient skyline and top-k retrieval in subspaces, IEEE Trans. Knowl. Data Eng. 19 (8) (2007) 1072–1088.
[33] Mathuranathan Viswanathan. Wireless Communication Systems in Matlab (Second edition). Independently published, 2020..
[34] Changping Wang, Chaokun Wang, Gaoyang Guo, Xiaojun Ye, S.Yu. Philip, Efficient computation of g-skyline groups, IEEE Trans. Knowl. Data Eng. 30 (4) (2018) 674–688.
[35] Hongbing Wang, Hu. Xingguo, Yu. Qi, Gu. Mingzhu, Wei Zhao, Jia Yan, Tianjing Hong, Integrating reinforcement learning and skyline computing for adaptive service composition, Inf. Sci. 519 (2020) 141–160.
[36] Wenhui Yu, Zheng Qin, Jinfei Liu, Li Xiong, Xu Chen, and Huidi Zhang. Fast algorithms for pareto optimal group-based skyline. In Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, pages 417–426. ACM, 2017..
[37] Kaiqi Zhang, Hong Gao, Xixian Han, Zhipeng Cai, Jianzhong Li, Modeling and computing probabilistic skyline on incomplete data, IEEE Trans. Knowl. Data Eng. 32 (7) (2020) 1405–1418.
[38] Nan Zhang, Chengkai Li, Naeemul Hassan, Sundaresan Rajasekaran, Gautam Das, On skyline groups, IEEE Trans. Knowl. Data Eng. 26 (4) (2014) 942–956.
[39] Shiming Zhang, Nikos Mamoulis, David W. Cheung, Scalable skyline computation using object-based space partitioning, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, IEEE, 2009, pp. 483–494.
[40] Yandong Zheng, Rongxing Lu, Beibei Li, Jun Shao, Haomiao Yang, and Kim-Kwang Raymond Choo. Efficient privacy-preserving data merging and skyline computation over multi-source encrypted data. Inf. Sci., 498:91–105, 2019..
[41] Xu. Zhou, Kenli Li, ZhiBang Yang, Guoqing Xiao, Keqin Li, Progressive approaches for pareto optimal groups computation, IEEE Trans. Knowl. Data Eng. 31 (3) (2019) 521–534.
[42] Juan Zou, Ruiqing Sun, Shengxiang Yang, Jinhua Zheng, A dual-population algorithm based on alternative evolution and degeneration for solving constrained multi-objective optimization problems, Inf. Sci. 579 (2021) 89–102.