Discovering Group Skylines with Constraints by Early Candidate Pruning

Ming-Yen Lin[1], Yueh-Lin Lin[1], and Sue-Chen Hsueh[2(✉)]

[1] Feng Chia University, Taichung 40724, Taiwan
[2] Chaoyang University of Technology, Taichung 41349, Taiwan
schsueh@cyut.edu.tw

Abstract. Skyline query has been an important issue in the database community. Many applications nowadays request the skyline after grouping tuples, such as fantasy sports, so that the group skyline problem becomes the research focus. Most previous algorithms intended to quickly sift through the numerous combinations but fail to address the problem of constraints. In practice, nearly all groupings are specified with constrains, which demand solutions of constrained group skyline. In this paper, we propose an algorithm called CGSky to efficiently solve the problem. CGSky utilizes a pre-processing method to exclude the unnecessary tuples and generate candidate groups incrementally. A pruning mechanism is devised in the algorithm to prevent non-qualifying candidates from the skyline computation. Our experimental results show that CGSky improves an order of magnitude over previous algorithms in average. It also shows that CGSky has good scale-up capability on different data distributions.

Keywords: Skyline query · Group skyline · Constraint · Constrained skyline

1 Introduction

Skyline is a query operator that helps users to retrieve interesting tuples from databases [5, 8, 9]. These tuples are not dominated by any other tuples. A typical example is that in a hotel relation having price and distance-to-beach attributes. A user is likely to ask for hotels that are both cheap and close to the beach. If the price of a hotel h is higher than that of a hotel h′ and the distance-to-beach of h is not shorter than that of h′, h is *dominated* by h′, written as $h' \succ h$, and h can be excluded from the query result. Skyline queries return the sets of tuples that are not *dominated* by any other tuples. Typically, a user may specify some constraints on certain attributes, such as price, so that constrained skyline [2, 13, 15, 21] might be more common in practice. For example, the skyline query for best web-portal advertising plan might be constrained by 'cost $\leq$ 6000'. Skyline query and constrained skyline query thus are used in many multi-criteria decision support applications.

Although (constrained) skyline query may retrieve non-dominated tuples, interesting groups generated from combinations of single tuples can be more desirable in some applications. For example, to increase visibility of new products, a company may advertise on more than one web portal so that the skyline result contains combinations of portals, which combinations are not dominated by other ones, considering the total

G. Cong et al. (Eds.): ADMA 2017, LNAI 10604, pp. 49–62, 2017.
https://doi.org/10.1007/978-3-319-69179-4_4

'cost' and the total 'number-of-visitors' (#visitors) attributes. Given a number of portals, a user may specify that a group is composed of 2 portals. The problem of group skyline [17] is to find the groups, considering all the combinations of 2 portals, which are not dominated by other groups. The attribute-values of the group thus are aggregate-values (e.g. sum) of the 2 portals. Applications querying the skylines of groups are commonplace, such as baseball teams of 9 persons as a group, fantasy basketball of 5 persons as a group, Hackathon of 3 persons as a group, etc. The cardinality of a group (9 in baseball, 5 in basketball, etc.) is specified by users, which is called the group cardinality, denoted by k. In addition, the aggregate-function is also specified by users, which can be sum(), min(), max(), and so on. Because group skyline has to consider all the combinations of certain cardinality, the problem is more complicated than traditional skyline problems.

Nevertheless, no previous studies have discussed group skylines with constraints. A common scenario is that not all combinations are acceptable due to certain constraints. For example, advertising web-portals may subject to a limited total budget, which means the total cost (by summing cost of the portals in the combination) cannot exceed certain amount. Given a query of investment portfolios for stocks, the investors may have a budget limit of 10000 for the non-dominated portfolio of 3 stocks ($k = 3$). Also, NBA league rules that every NBA team cannot exceed a certain amount of team salary, which means the total salary is constrained. The result of the group skyline would be groups of players having good (aggregated) scoring/defense capability while satisfying the total salary constraint.

Table 1. An example of web portals datasets.

Tuple	Web Portal	Cost	#Visitors	Skyline
t_1	Facebook	3	4	v
t_2	Google	10	4	
t_3	Yahoo	5	1	
t_4	Apple Daily	7	10	v
t_5	Mobile01	8	6	
t_6	PChome	6	7	v

However, finding group skylines with constraints is complicated. In Table 1, tuples t_1, t_4, and t_6 are skyline tuples since $t_1 \succ t_2$, $t_1 \succ t_3$, and $t_6 \succ t_5$, when smaller 'cost' and larger '#visitors' are preferred in web-portal advertising plans. Using the 6 tuples for groups of cardinality $k = 3$, there are total 20 groups as listed in Table 2. Although t_3 is not a skyline tuple in Table 1, group G_5 having t_1, t_3 and t_4 is a group skyline tuple. Similarly, tuple t_5, dominated by t_6 in Table 1, becomes the member of the group skyline tuple G_{20}. That is, both combinations of skyline tuples and that of non-skyline tuples have to be considered in the group skyline finding process. The number of combinations can be very huge. For example, assume that a dataset contains 500 tuples and group cardinality $k = 5$, about $C(500, 5) = 2.6 \times 10^{11}$ candidate groups will be generated in total. To compute such a huge amount of combinations is a serious challenge. Furthermore, the constraint can only be considered after groups are formed

because the constraint is specified against the aggregated attribute-value. This restricts effective pruning of candidate groups and is more time-consuming, comparing to specifying constraint on single tuples of a typical skyline query. Constrained skyline may exclude tuples that do not satisfy the constraint a priori to reduce the computation but constrained group skyline needs to generate all candidate groups to exclude the groups unsatisfying the constraint. Therefore, the discovery of the group skyline with constraints is much more difficult than that of both skyline with constraints and group skyline without constraints.

Table 2. Enumerated groups of cardinality 3 for the dataset in Table 1.

Group	Members	Cost	#Visitors	Skyline
G_1	t_1, t_2, t_3	18	9	
G_2	t_1, t_2, t_4	20	18	
G_3	t_1, t_2, t_5	21	14	
G_4	t_1, t_2, t_6	19	15	
G_5	t_1, t_3, t_4	15	15	v
G_6	t_1, t_3, t_5	16	11	
G_7	t_1, t_3, t_6	14	12	v
G_8	t_1, t_4, t_5	18	20	
G_9	t_1, t_4, t_6	16	21	v
G_{10}	t_1, t_5, t_6	17	17	
G_{11}	t_2, t_3, t_4	22	15	
G_{12}	t_2, t_3, t_5	23	11	
G_{13}	t_2, t_3, t_6	21	12	
G_{14}	t_2, t_4, t_5	25	20	
G_{15}	t_2, t_4, t_6	23	21	
G_{16}	t_2, t_5, t_6	24	17	
G_{17}	t_3, t_4, t_5	20	17	
G_{18}	t_3, t_4, t_6	18	18	
G_{19}	t_3, t_5, t_6	19	14	
G_{20}	t_4, t_5, t_6	21	23	v

The problem of finding group skyline with constraint is defined as follows. Given a database of n tuples $D = \{t_1, t_2, \ldots, t_n\}$ of m numeric attributes, user-specified group cardinality k ($k > 1$), and a constraint c, the objective is to find the set of group skyline tuples satisfying c. A tuple t_i is represented as $(t_i[A_1], t_i[A_2], \ldots, t_i[A_m])$. Tuple t_x *dominates* tuple t_y, denoted by $t_x \succ t_y$, if $\forall\, i\, (1 \leq i \leq m)$, $t_x[A_i] \geq t_y[A_i]$, and $\exists\, j\, (1 \leq j \leq m)$, $t_x[A_j] > t_y[A_j]$. The operators '$>$' and '$\geq$' can be replaced by '$<$' and '$\leq$' when smaller values are preferred, such as the smaller 'cost' and the larger 'number-of-visitors' the example of Table 1. A group tuple $Gx = \{t_1', t_2', \ldots, t_k'\}$ is a combination of k distinct tuples in D, where $t_p' \in D\ \forall 1 \leq p \leq k$. Gx also has m attributes and the attribute value of $Gx[A_i] = \sum_{p=1}^{k} t_p'[A_i]$ $(1 \leq i \leq m)$ if sum() is the

preferred aggregate-function. The *Gx* is also called a *k-tuple* group since it has *k* tuples. A group tuple *Gx* satisfying constraint *c* is denoted by G_x^c. A group G_x^c is said to dominate group G_y^c, denoted by $G_x^c \succ_G G_y^c$, if and only if $\forall\, i\ (1 \leq i \leq m)$, $G_x^c[A_i] \geq G_y^c[A_i]$, and $\exists\, j\ (1 \leq j \leq m)$, $G_x^c[A_j] > G_y^c[A_j]$. G_x^c is a group skyline tuple satisfying *c* if there exists no G_y^c such that $G_y^c \succ_G G_x^c$. The objective is to find the set of all the group skyline tuples satisfying *c*.

For example, given dataset D in Table 1, group cardinality k = 3, and constraint c = 'Cost < 19', all the 20 combinations are listed in Table 2. When constraint c is not considered, the group skyline tuples are G5, G7, G9, and G20. The group skyline tuples satisfying c are G_5^c, G_7^c, and G_9^c because G20 does not satisfy 'Cost < 19' constraint. Alternatively, we said that the three group tuples are constrained group skyline tuples.

In this paper, we present a novel algorithm called CGSky (*Constrained Group Skyline*), for solving troblem of computing group skyline with constraints. In the following context, the group skyline tuples satisfying the constraint are simply called the group skyline tuples, or collectively named the group skyline.

The rest of the paper is as follows. Section 2 briefly reviews the related work. The proposed CGSky algorithm is presented in Sect. 3. Section 4 describes the experimental results. Finally, Sect. 5 concludes the paper.

2 Related Work

The skyline operator was first introduced in [1] and many algorithms have been proposed. These algorithms can be categorized into generic and index-based types. Generic skyline algorithms do not need pre-computation, such as BNL [1], D&C [1], SFS [4], SSPL [10], etc. Index-based skyline algorithms utilize the pre-processing data structures to avoid scanning the entire dataset, such as NN [12], BBS [18], Bitmap [20], etc. Generic skyline algorithms usually incur high I/O cost, while the efficiency of index-based ones will decrease as the number of attributes increases.

Constrained skyline was proposed on the extension of the BBS algorithm [6]. The main idea of constrained skyline is to compute the results satisfying user preferences. Two types of constraint problems were described. The constrained skyline problem [2] uses the constraint to filter out tuples first, then computes the results using the remaining tuples. The skyline with constraints problem is to computing the skyline first, then using the constraint to filter out the results dis-qualifying the constraint.

Computing the group skyline is a complicated task. GDynamic [11] algorithm utilizes an incremental method to overcome the bottleneck, which is the number of candidate groups. There are $\binom{n}{c}$ possible combinations for n tuples and group cardinality of c. The Improved Decomposition Algorithm (IDA) [6] is a combinatorial skyline algorithm. The dynamic programing algorithm based on order-specific property (OSM) [22] is also a group skyline algorithm. While the G-Skyline [14] is a group skyline algorithm without using aggregate function as the foundation of group dominance relation. Among these algorithms, the IDA algorithm utilizes a pre-processing

method to compute the number of dominating numbers for each tuple. Moreover, the pre-processing result will output a dominance table for speeding up the formation of groups. However, all these algorithms ignore that the group skyline might incorporate the need of certain constrains. The focus of the study is to push constraints into the computation process so that the skyline finding process can be greatly accelerated. As indicated in the experimental results, the proposed algorithm successfully improve the discovering process.

3 Proposed Algorithm

The proposed CGSky algorithm, inspired by the IDA algorithm [6], computes the constrained group skyline in three phases: early tuple-pruning, candidate-group generation, and group-dominance checking. Figure 1 is an overview of the CGSky algorithm. The dominance relationships among tuples are computed and tuples cannot become members of group skyline tuples are pruned in the phase of early tuple-pruning. The remaining tuples are sorted in ascending order of the constraint attribute in this phase. The sorted tuples are used to generate candidate groups recursively in the phase of candidate-group generation. Finally, the resulting group skyline tuples satisfying the constraint are found by applying any (single-tuple) skyline algorithm, such as SFS [4], on the candidate groups in the phase of group-dominance checking.

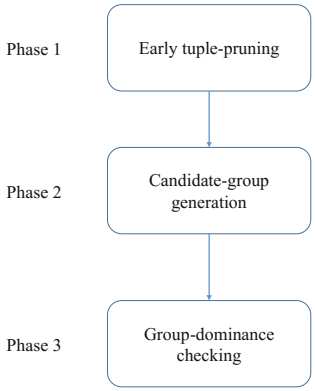

Fig. 1. An overview of the CGSky algorithm.

3.1 Phase One: Dominance-Table Computation

First, Phase one of the CGSky algorithm reduces the number of candidate groups by pruning tuples that cannot become members of the final group skyline tuples. In this phase, the CGSky algorithm first computes the dominance relationships among tuples in D. For each tuple t, the number of tuples dominating t is accumulated. The number is referred to as the *dominating number* of t, denoted by *dom*(t). Once a tuple's dominating number is larger than the group cardinality *k*, the tuple is pruned and no

candidate group will include this tuple as a member. Any group formed by these eliminated tuples is impossible to be a skyline group tuple, as proved in Theorem 1. That is, if $dom(t) \geq k$, then tuple t is eliminated from the generation of candidate groups of cardinality k. For convenience, we use '$t \oplus^{k-1} G_x$' to represent the group tuple formed by a (k-1)-tuple group G_x and tuple t. CGSky then sorts tuples by ascending order of the constraint attribute. Sorting tuples by the constraint attribute is beneficial to the generation of candidate groups, as presented in Sect. 3.2

Theorem 1. Tuple t with $dom(t) \geq k$ cannot form a skyline group tuple for group cardinality of k.

Proof. Tuple t is dominated by at least k tuples in dataset D since $dom(t) \geq k$. Assume t is included in a group tuple $G = t \oplus^{k-1} G_x$, we will show that there exists a group tuple G′, which is formed by $t' \oplus^{k-1} G_x$ and $G' \succ_G G$. (i) Let the set of tuples dominating t be $H = \{t_1', t_2', \ldots, t_k'\}$ if $dom(t) = k$. When $H \cap {}^{k-1}G_x = \phi$, there exists $t' \in H$, $G' = t' \oplus^{k-1} G_x \succ_G G = t \oplus^{k-1} G_x$ since $t' \succ t$. Let $H \cap {}^{k-1}G_x = {}^{p}G_y$ ($1 \leq p \leq k\text{-}1$) and $H' = H - {}^{p}G_y$, thus there are (k-p) tuples in H′ dominating t. Then there exists $t' \in H'$, $G' = t' \oplus ({}^{p}G_y \oplus^{k-p-1} G_z) \succ_G t \oplus ({}^{p}G_y \oplus^{k-p-1} G_z)$ since $t' \succ t$. (ii) If $dom(t) > k$ then we just pick k tuples from the set dominating t to constitute $H = \{t_1', t_2', \ldots, t_k'\}$, the rest of the proof is the same as (i). Thus, any group having t must be dominated by some other group. Tuple t with $dom(t) \geq k$ cannot form a skyline group tuple and can be safely eliminated from the generation of candidate groups. □

For example, tuples in Table 1 are processed in this phase and the dominating number of each tuple is obtained, as shown in Table 3. In addition, tuples are sorted in ascending order of Cost, which is the constraint attribute. Tuple t_2 is not engaged in the generation of candidate groups in phase two since its $dom(t_2) > k$ for group cardinality $k = 3$. The exclusion can be illustrated as follows. A group G formed by including t_2 will be dominated by a group G′, by replacing t_2 with any one of the four dominating tuples $\{t_1, t_6, t_4, t_5\}$. Only two tuples at most will be used to form G for k = 3 so that there are always two remaining tuples to be picked to form G′. Obviously, G′ dominates G. Consequently, t_2 cannot produce any 'potential' group skyline tuple and can be eliminated from the candidate generation.

Table 3. Dominating numbers for Table 1 (sorted by Cost).

Tuple	Cost	#Visitors	dom(t_i)
t_1	3	4	0
t_3	5	1	1
t_6	6	7	0
t_4	7	10	0
t_5	8	6	2
t_2	10	4	4

3.2 Phase Two: Candidate-Group Generation

Figure 2 presents the pseudo-code of the candidate-group generation. The CGSky algorithm invokes GenCandidate with parameters (D[1, n], k, limit), where D[1, n] is the dataset after the early pruning in phase one, k is the group cardinality, and limit is the upper bound of the constraint value. The subroutine generates all the candidate groups of cardinality k satisfying the constraint (limit). The principle of this phase is as follows.

```
Subroutine GenCandidate
Input: D[p,q] – list of tuples t_p, t_{p+1}, …, t_q
       k – group cardinality
       limit – constraint limit // constraint attribute A_c
Output: A = set of candidate group tuples
1.  A = ϕ ;
2.  if (k=1)
3.    for i = p to q do
4.      if (t_i[Ac] ≥ limit) break ;
5.      add {t_i} to A ;
6.    endfor
7.    return A ;
8.  endif
9.  for i = p to q-(k-1)
10.   if (t_i[A_c] ≥ limit) break ;
11.   A' = GenCandidate(D[i+1,q], k-1, limit-t_i[A_c]) ;
12.   if (A'= ϕ) break ;
13.   for each set S in A'
14.     Add t_i to S ;
15.     Add S to A ;
16.   endfor
17. endfor
18. return A;
```

Fig. 2. Pseudo-code of the candidate-group generation.

Given the sorted list of tuples $[t_p, t_{p+1},\ldots, t_i, t_{i+1},\ldots, t_q]$, if tuple t_i with constraint-attribute value t_i[Ac] will constitute a k-tuple group with a (k-1)-tuple group Gx, then the aggregate-value of Gx[Ac] must be less than (limit′ = limit-t_i[Ac]). In addition, the Gx is constructed from potential combinations of (k-1) tuples from list $[t_{i+1},\ldots, t_q]$. Recursively, if tuple t_{i+1} with constraint-attribute value t_{i+1}[Ac] will constitute a (k-1)-tuple group with a (k-2)-tuple group Gx′, then the aggregate-value of Gx′[Ac] must be less than (limit′-t_{i+1}[Ac]). The Gx′ is constructed from potential combinations of (k-2) tuples from list $[t_{i+2},\ldots, t_q]$. The recursion eventually would reach the formation of 1-tuple group from list $[t_p', t_{p'+1},\ldots, t_q]$, constrained by certain upper bound $limit^z$. When some tuple t_s in the list having t_s[Ac] $\geq$ $limit^z$ is unqualified, all the

rest of tuples are impossible to satisfy the bound since tuples are sorted in ascending order of constraint value (lines 2–8). Therefore, we may prevent a large number of "unqualified" candidate groups from generation. Furthermore, when no combination is generated during the construction of certain group tuple, assume that such combinations are to be used with t_h, then no tuples after t_h in the list may generate a qualified group (lines 9–17).

For example, let the CGSky algorithm invoke GenCandidate(D[***t1***, t_3, t_6, t_4, t_5], k = 3, limit = **19**) in Table 3. Tuple t_1 can only form qualified candidates with 2-tuple groups, generated from D[t_3, t_6, t_4, t_5] and constrained by (2-tuple group) limit of 19-t_1[Cost] = 16. This will invoke GenCandidate(D[***t3***, t_6, t_4, t_5], k = 2, limit = **16**). The call with t_3 can only form qualified candidates with 1-tuple groups, generated from D [t_6, t_4, t_5] and constrained by limit of 16-t_3[Cost] = 11. GenCandidate(D[t_6, t_4, t_5], k = 1, limit = **11**) returns {t_6}, {t_4} and {t_5} so that {t_3, t_6}, {t_3, t_4} and {t_3, t_5} are returned. The three will be collected for t_1 to form 3-tuple groups {t_1, t_3, t_6}, {t_1, t_3, t_4} and {t_1, t_3, t_5} later. The GenCandidate(D[t_3, ***t6***, t_4, t_5], k = 2, limit = **16**) continues with t_6. This call with t_6 can only form qualified candidates with 1-tuple groups, generated from D[t_4, t_5] and constrained by limit of 16-t_6[Cost] = 10. GenCandidate (D[t_4, t_5], k = 1, limit = **10**) returns {t_4} and {t_5} so that {t_6, t_4} and {t_6, t_5} are returned. The two will be collected for t_1 to form 3-tuple groups {t_1, t_6, t_4} and {t_1, t_6, t_5} later. The GenCandidate(D[t_3, t_6, ***t4***, t_5], k = 2, limit = **16**) continues with t_4. This call with t_4 can only form qualified candidates with 1-tuple groups, generated from D [t_5] and constrained by limit of 16-t_4[Cost] = 9. GenCandidate(D[t_5], k = 1, limit = **9**) returns {t_5} so that {t_4, t_5} is returned. This one will be collected for t_1 to form a 3-tuple group {t_1, t_4, t_5} later. The call with t_1 now stops.

Next, GenCandidate(D[t_1, ***t3***, t_6, t_4, t_5], k = 3, limit = **19**) continues with tuple t_3. Tuple t_3 can only form qualified candidates with 2-tuple groups, generated from D[t_6, t_4, t_5] and constrained by (2-tuple group) limit of 19-t_3[Cost] = 14. This will invoke GenCandidate(D[***t6***, t_4, t_5], k = 2, limit = **14**). The call with t_6 can only form qualified candidates with 1-tuple groups, generated from D[t_4, t_5] and constrained by limit of 14-t_6[Cost] = 8. GenCandidate(D[t_4, t_5], k = 1, limit = **8**) returns {t_4} so that {t_6, t_4} is returned. The {t_6, t_4} will be used with t_3 to form 3-tuple groups {t_3, t_6, t_4} later.

Subsequently, GenCandidate(D[t_1, t_3, ***t6***, t_4, t_5], k = 3, limit = **19**) continues with tuple t_6. Tuple t_6 can only form qualified candidates with 2-tuple groups, generated from D[t_4, t_5] and constrained by (2-tuple group) limit of 19-t_6[Cost] = 13. This will invoke GenCandidate(D[t_4, t_5], k = 2, limit = **13**). The call with t_4 can only form qualified candidates with 1-tuple groups, generated from D[t_5] and constrained by limit of 13-t_4[Cost] = 6. GenCandidate(D[t_5], k = 1, limit = **6**) returns empty so that the call with t_4, invoked by the call with t_6 are stopped. Any invocation after t_6 cannot generate valid 3-tuple groups so the recursion ends. The candidate 3-tuple groups are {t_1, t_3, t_6}, {t_1, t_3, t_4}, {t_1, t_3, t_5}, {t_1, t_6, t_4}, {t_1, t_6, t_5}, {t_1, t_4, t_5}, and {t_3, t_6, t_4}.

3.3 Phase Three: Group-Dominance Checking

The resulting group skyline tuples satisfying the constraint are found by applying any common skyline algorithm, such as the BNL algorithm [1], on the candidate groups. The final group skyline with constraints includes {t_1, t_3, t_6}, {t_1, t_3, t_4}, and {t_1, t_6, t_4}.

Note that only 8 candidates, rather than 20 candidates, are generated by the CGSky algorithm. The number of candidates required for dominance checking using common skyline algorithms is greatly reduced.

4 Experimental Results

Comprehensive experiments were executed to assess the performance of the proposed algorithm. All the algorithms were executed on a Windows 7 PC, with Intel(R) Core (TM) i5-4460 3.2 GHz, 16 GB RAM and 1 TB hard disk. A modified IDA algorithm [6], called IDA*, and CGSKY were compared in the experiments. Both were implemented in Java. The IDA algorithm [6] is one of the representative group skyline algorithms up-to-date. The IDA* algorithm extends IDA with an additional phase of selecting the groups satisfying the constraints. Both synthetic datasets and a real dataset were used in the experiments. Here, we report the results on anti-correlated datasets, the results on independent and correlated datasets were similar.

Similar to most skyline algorithms, the synthetic datasets include distributions of correlated, anti-correlated and independent data. Distinct datasets were generated using different parameters including data size n, group cardinality k, number of attributes m, and constraint (limit) c. The used parameters are summarized in Table 4. The default setting was n = 500, k = 3, m = 3, and c < 120. The range of values for each attribute was uniformly distributed from 0 to 100. All the attribute values were independently generated in the independent dataset. For a correlated dataset, if the value of the first attribute is x, the values of the rest attributes range from x*0.95 to x*1.05. For an anti-correlated dataset, if the value of the first attribute is x, the values of the rest attributes range from 100-x*0.95 to 100-x*1.05.

Table 4. Parameter values used in the synthetic datasets.

Parameter	Used value	Default
Data Size n	100, 300, 500, 700, 900	500
Group cardinality k	2, 3, 4, 5	3
Attribute m	2, 3, 4, 5	3
Constraint c	80, 120, 160, 200, 240	120

Figure 3 shows the results of executions on anti-correlated datasets of k = 3, m = 3, and c < 120, by varying the data size n from 100 to 900. The results on independent datasets and correlated datasets were similar. As the data size increases, both the number of candidate groups and the number of constrained group skyline tuples increase. In average, CGSky runs four times faster than IDA*. Table 5 lists the number of candidate tuples, that of candidate groups, and that of group skyline tuples with respect to the algorithms. For example, in Table 5 with n = 100, both algorithms eliminated 60 tuples having dominating number larger than the group cardinality. This leaves C(40, 3) = 4980 candidate groups generated for IDA* while CGS generated

only 2013 candidates by incorporating the constraint during candidate generation. The number of (constrained) group skyline tuples was the same for both algorithms.

Figure 4 shows the results of executions on anti-correlated datasets of n = 500, k = 3, and c < 120, by varying the number of attributes m from 2 to 5. In average, CGSky runs 11.1 times faster than IDA*. IDA* spent 1400 s to compute the answer when m = 5. As listed in Table 6, the number of candidate groups for IDA* is 5.6 times for CGSky when m = 4. The increase in the number of attributes has a great impact on the total execution time since the number of "incomparable" tuples increases exponentially.

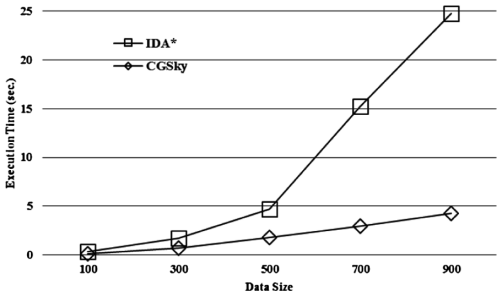

Fig. 3. Results on varying data size *n*.

Table 5. Number of candidate tuples and candidate-groups w.r.t. *n*.

n	Candidate tuples	IDA*	CGSky
100	40	9880	2013
300	66	45760	15693
500	90	117480	41971
700	119	273819	94130
900	133	383306	124480

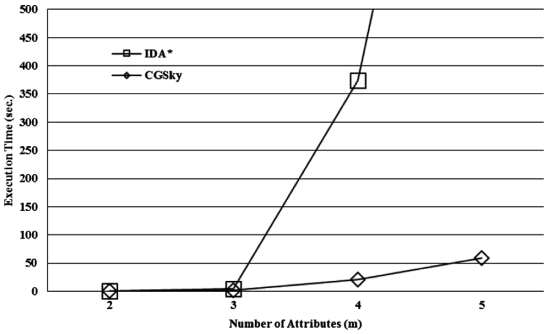

Fig. 4. Results on varying number of attributes *m*.

Table 6. Number of candidates, candidate-groups, and group-skyline tuples w.r.t. *m*.

m	Candidate tuples	IDA*	CGSky
2	31	4495	3121
3	90	117480	41971
4	182	988260	174493
5	272	3317040	421015

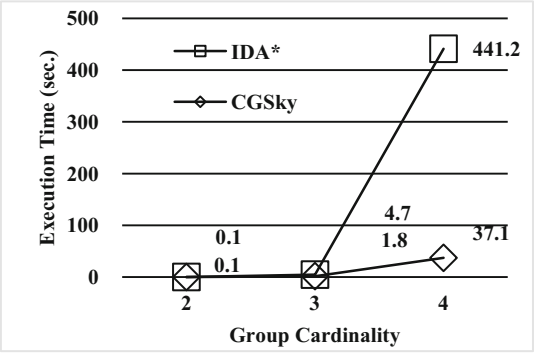

Fig. 5. Results of varying group cardinality *k*.

Table 7. Number of candidates, candidate-groups, and group-skyline tuples w.r.t. *k*.

k	Candidate tuples	IDA*	CGSky
2	71	2485	1452
3	91	121485	30918
4	111	5989005	1675519
5	127	254231775	21450113

Next, we investigated the results of varying group cardinality, which was varied from 2 to 5, with n = 500, m = 3, and c < 120. In average, CGSky runs 5.2 times faster than IDA*, as shown in Fig. 5. When k = 5, IDA* spent 8000 s and CGSky spent 3196 s for the computation. Table 7 indicates that CGSky effectively eliminated a large number of candidate groups. IDA* had to process 3.6 times of candidate groups than CGSky did. That is why CGSky outperforms IDA* for about 11 faster with group cardinality k = 4.

The next experiment was varying constraint c from 80 to 240 on anti-correlated datasets, with n = 500, m = 3, and k = 3. The execution time of the IDA* algorithm stayed nearly constant of 4.7 s since the time-consuming process of finding group skyline tuples took the same time, and the constraint is used only to retrieve groups passing the threshold. The execution time of the CGSky algorithm increased as the constraint value increased, because the number of candidates increased. The CGSky algorithm finished less than 1 s for 'c < 80', increased to 1.7 s for 'c < 120', but kept

less than 4 s for 'c < 240'. Note that the constraint can be useless when the constraint value is close to 300 for k = 3.

The experiments continued with the read-world dataset, from http://tw.global.nba.com/statistics/ with the NBA 2015-2016 regular season data. This data contains 412 players with five attributes: salary, points, rebounds, assists, and steals. The salary attribute is the constraint attribute. The salary constraint of 70 million is set for 12 players by NBA league so the default salary constraint used in the experiment was about 30 million for group cardinality k = 5. The number of attributes was varied from 2 to 5 and the result is shown in Fig. 6. The experimental results of varying k and varying c are similar.

Figure 6 shows the effect of varying the number of attributes m from 2 to 5. Again, the CGSky algorithm runs 3 times faster than the IDA* algorithm. This confirms that the early pruning and the candidate group generation are very effective in reducing the number of candidate groups.

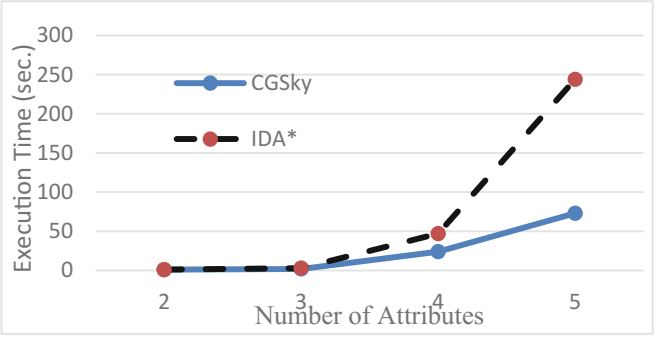

Fig. 6. Results on NBA real-datasets by varying *m*.

5 Conclusion

In this paper, we propose the CGSky algorithm for discovering constrained group skylines. The CGSky algorithm features in the reduction of candidate groups and pruning impossible candidate combinations from applying the constraints. The comprehensive experiments comprising synthetic and real datasets confirm that the CGSky algorithm outperform the well-known IDA algorithm by an order of magnitude faster in average. Future extension of the study could be finding the group skyline where group members are formed with specified characteristics, or finding group skylines with constraints in distributed computing platforms [3, 7, 16, 19].

Acknowledgements. The authors appreciate the valuable comments from the reviewers. This research was supported partly by the Ministry of Science and Technology, R.O.C. under grant MOST 105-2634-E-004-001.

References

1. Borzsonyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proceedings of the 17th International Conference on Data Engineering, pp. 421–430 (2001)
2. Chen, L., Cui, B., Lu, H.: Constrained skyline query processing against distributed data sites. IEEE Trans. Knowl. Data Eng. (TKDE) **23**(2), 204–217 (2011)
3. Chen, L., Hwang, K., Wu, J.: MapReduce skyline query processing with a new angular partitioning approach. In: 26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, pp. 2262–2270 (2012)
4. Chomicki, J., Godfrey, P., Gryz, J., Liang, D.: Skyline with presorting. In: Proceedings of the 19th International Conference on Data Engineering, pp. 717–719 (2003)
5. Chomicki, J., Ciaccia, P., Meneghetti, N.: Skyline queries, front and back. SIGMOD Rec. **42**(3), 6–18 (2013)
6. Chung, Y.C., Su, I.F., Lee, C.: Efficient computation of combinatorial skyline queries. Inf. Syst. **38**(3), 369–387 (2013)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: 6th Symposium on Operating System Design and Implementation (OSDI), pp. 137–150 (2004)
8. Dellis, E., Seeger, B.: Efficient computation of reverse skyline queries. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 291–302 (2007)
9. Endres, M., Roocks, P., Kießling, W.: Scalagon: An Efficient Skyline Algorithm for All Seasons. In: Renz, M., Shahabi, C., Zhou, X., Cheema, M.A. (eds.) DASFAA 2015. LNCS, vol. 9050, pp. 292–308. Springer, Cham (2015). doi:10.1007/978-3-319-18123-3_18
10. Han, X., Li, J., Yang, D., Wang, J.: Efficient skyline computation on big data. IEEE Trans. Knowl. Data Eng. (TKDE) **25**(11), 2521–2535 (2013)
11. Im, H., Park, S.: Group skyline computation. Inf. Syst. **188**, 151–169 (2012)
12. Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: an online algorithm for skyline queries. In: Proceedings of 28th International Conference on Very Large Data Bases, pp. 275–286 (2002)
13. Lee, J., Hwang, S.W.: Toward efficient multidimensional subspace skyline computation. VLDB J. **23**(1), 129–145 (2014)
14. Liu, J., Xiong, L., Pei, J., Luo, J., Zhang, H.: Finding pareto optimal groups: group-based skyline. Proc. VLDB Endowment **8**(13), 2086–2097 (2015)
15. Mortensen, M.L., Chester, S., Assent, I., Magnani, M.: Efficient caching for constrained skyline queries. In: Proceedings of the 18th International Conference on Extending Database Technology (EDBT), pp. 337–348 (2015)
16. Mullesgaard, K., Pedersen, J.L., Lu, H., Zhou, Y.: Efficient skyline computation in MapReduce. In: Proceedings of the 17th International Conference on Extending Database Technology (EDBT), pp. 37–48 (2014)
17. Magnani, M., Assent, I.: From stars to galaxies: skyline queries on aggregate data. In: Proceedings of the 16th International Conference on Extending Database Technology (EDBT), pp. 477–488 (2013)
18. Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 467–478 (2003)
19. Park, Y., Min, J.K., Shim, K.: Parallel computation of skyline and reverse skyline queries using MapReduce. Proceedings of the VLDB Endowment **6**(14), 2002–2013 (2013)

20. Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient progressive skyline computation. In: Proceedings of 27th International Conference on Very Large Data Bases, pp. 301–310 (2001)
21. Zhang, M., Alhajj, R.: Skyline queries with constraints: integrating skyline and traditional query operators. Data Knowl. Eng. **69**(1), 153–168 (2010)
22. Zhang, N., Li, C., Hassan, N., Rajasekaran, S., Das, G.: On skyline groups. IEEE Trans. Knowl. Data Eng. (TKDE) **26**(4), 942–956 (2014)