Reverse k-skyband query based on reuse technology

Tao Jiang[#], Bin Zhang[#]
[#]College of Mathematics Physics and Information Engineering
Jiaxing University
Jiaxing, P. R. China
e-mail: jiangtao_guido@yahoo.com.cn

Lu Chen[*], Qing Liu[*], Jianyong Yan[#]
[*]College of Computer Science and Technology
Zhejiang University
Hangzhou, P. R. China
e-mail: {luchen, qingliu}@ zju.edu.cn

Abstract*—In this paper, we introduce a new approach to finish reverse k-skyband (RkSB) query which returns all the points in given dataset P whose dynamic k-skyband contains specific query object q. The main ideas include reuse technology and early stopping. The former save the information of node accesses during R-tree search into an auxiliary heap so that dramatically decreases the I/O cost and improves the CPU efficiency. This is because RkSB needs to execute global k-skyband and window query to finish the computation and the reuse information can be used in the phase of window query. The latter improves the efficiency of RkSB computation. The experimental results conducted on real datasets and synthetic datasets show that our proposed algorithm is effective and has a better performance both I/O cost and CPU efficiency than the algorithm based on reverse BBS (Branch and Bound Skyline) method.

Keywords- algorithm; skyline query; reverse skyline; skyband

I. INTRODUCTION

Recently, many researchers from database research community pay much attention to *skyline queries* [1-8] due to its wide applications related to multi-criteria decision making. Specifically, given a d-dimensional data set D, a *skyline query* [3] retrieves all the data objects which are not *dominated* by other objects in D. A data object p dominates another data object p' if p is not worse than p' in all dimensions and strictly better than p' in at least one dimension.

Papadias et al. [3] relax the restriction of skyline definition and extend skyline queries to k-skyband queries, which return all the data objects that are dynamically dominated by *at most k* data objects. In fact, to identify the influence of query object q, traditional skyline queries can be computed from company perspective. This skyline query is called reverse skyline query [6-7]. In contrast, reverse k-skyband (RkSB) query proposed by Liu et al. [4] is a natural extension of *dynamic k-skyband* query. Although RkSB query can be answered by the algorithms in [4], for example, *branch-bound-based RkSB* (BRkSB), they do not take advantage of reuse technology. In fact, RkSB produces many redundant I/O owing to the window query during the procedure of search.

In this paper, we propose a novel RkSB algorithm which utilizes reuse information technology [8] and early stopping method. We call the algorithm RRkSB (reuse RkSB). Our experimental results show that RRkSB reduces more than 90% I/O cost than BRkSB; meanwhile, RRkSB has a less CPU time than BRkSB in most of cases.

II. PROBLEM FORMULATION

In this section, we illustrate the definitions of dynamic k-skyband, global k-skyband, and reverse k-skyband.

Definition 1 (Dynamic k-Skyband, DkSB). Given a multiple dimensional data set D, a query object q, and a parameter k, if a data object $p \in D$ belongs to the dynamic k-skyband of q, there exist at most k data objects in D so that they dynamically dominate p with respect to q.

Consider the example in Figure 1(a), where the data set is $D = \{p_1, p_2, \ldots, p_8\}$ and p_1' and p_6' are the projections of p_1 and p_6 with regard to p_2. The results of 1-skyband of p_2 are the set $R'=\{p_3, p_5, p_1, p_4, q, p_6\}$. For instance, there no data objects dominating object p_3 and p_4 is only dominated by p_3.

Definition 2 (Reverse k-Skyband, RkSB). Given a multiple dimensional data set D, a query object q, and a parameter k, the reverse k-skyband query of q, denote as RkSB(q), returns all the data objects in D whose dynamic k-skyband contains q.

Essentially, parameter k represents the thickness of skyline. When k becomes zero, the skyline corresponds to original skyline. For example, the results of D0SB(p_2) are $\{p_3, p_5\}$ in Figure 1(a), and R0SB(q) in Figure 1(b) returns $\{p_3, p_6, p_8\}$ as the results; however, when k becomes 1, D1SB(p_2) in Figure 1(a) retrieves the set of $\{p_3, p_5, p_1, p_4, q, p_6\}$ and R1SB(q) in Figure 1(b) returns the set of $\{p_3, p_4, p_2, p_5, p_7, p_9, p_6, p_8\}$ as the results.

Definition 3 (Global k-Skyband, GkSB). Given a query object q, a d-dimensional data set D, and a parameter k, if a data object $p \in D$ is in the global k-skyband of q, there exist at most k data objects in D which *globally dominate* [7] p.

GkSB query runs a window query to judge whether the current object p is in global k-skyband of q. Consider the example in Figure 1(b), the window of p_5 corresponds to the rectangle area, $w(p_5, q)$, which centers on q. Because $w(p_5, q)$ only includes a data object p_2, the object p_5 is among global k-skyband of q. Similarly, p_1 is not in global k-skyband of q because $w(p_1, q)$ contains two other data objects p_3 and p_4. In Figure 1(b), the data objects among the gray areas consists of the results of D1SB(q) among partitions I, II, III, and IV, namely, $\{p_1, p_3, p_4\}$, $\{p_7, p_9\}$, $\{p_2, p_5\}$ and $\{p_6, p_8\}$.

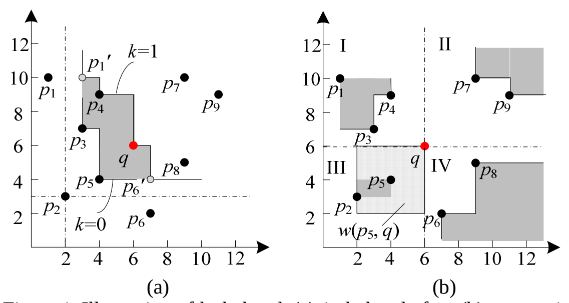


Figure 1. Illustration of k-skyband. (a) 1-skyband of p_2, (b) reverse 1-skyband of q.

III. RKSB QUERY PROCESSING

In this section, we firstly introduce how to reduce redundant I/O cost, and then give the procedure of query processing of our proposed RRkSB algorithm.

A. *Reduce redundant I/O cost*

Clearly, naïve R*k*SB query implemented by definition 2 has $O(|D|\times \mathrm{D}k\mathrm{SB}(q))$ time complexity. Owing to a larger cardinality of dataset D, the algorithm has a lowest efficiency. Recently, Liu et al. [4] introduces a better algorithm, called BR*k*SB. Because BR*k*SB is a customized algorithm of branch and bound skyline [3], it has optimal I/O performance.

However, BR*k*SB still has many redundant I/O accesses as a result of not reusing the node information of heap during the G*k*SB query. Consider the example in Figure 2, if the node information after G1SB query of q has been executed, are saved in a reuse heap H_r, i.e., the nodes of N_5, N_1, and N_6, the execution of G1SB of q can avoid to access *root* node. In fact, each window query of G*k*SB needs to access R-tree [9] one time. Therefore, this will bring many redundant I/O cost. This is because R*k*SB query cannot use the window query based on memory.

Now, let us consider the procedure of G1SB of q. R*k*SB requires executing multiple window queries for data objects, b, c, d, e, f, g, h, and i. For the window query of c, the *root* node is firstly accessed; then the node N_1 is accessed since it is intersected with the window $w(c, q)$; finally, the node N_3 is accessed. In fact, only one access is necessary and two redundant I/O accessed can be avoided because the G1SB of the first object b has saved the node information of *root* and N_1 into the reuse heap. Similarly, the window query of g brings 1 redundant I/O accesses since the *root* node is contained in reuse heap.

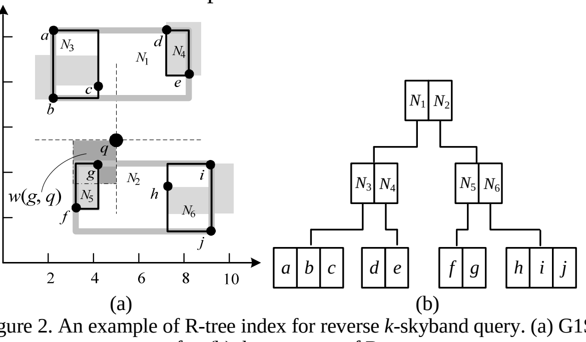


Figure 2. An example of R-tree index for reverse k-skyband query. (a) G1SB of q, (b) the structure of R-tree.

Therefore, a method to reduce redundant I/O is maintaining the node information having been accessed according to the procedure of R*k*SB search. We use a reuse heap to save the node information so that next window query can use them. Thus, much redundant I/O cost is saved. We call the algorithm reuse R*k*SB (RR*k*SB) algorithm. But, note that the mechanism of RR*k*SB is different from cache method because the former is related with R*k*SB procedure and the latter is related with local node information. Specifically, RR*k*SB inserts the node having been accessed into the reuse heap. When the child node of some node needs to be accessed, RR*k*SB extends the node and then deletes it from reuse heap.

B. *Query Processing*

In this subsection, we discuss the detailed RR*k*SB query processing which searches the R-tree [9] index in *best first* (BF) manner [3], and, meanwhile, utilizes the reuse heap to save the node information being accessed. Algorithm 1 illustrates the RR*k*SB query procedure in detail. Specifically, we first initialize the set variable S_r to the query results and the set variable S_c to save the candidate object (i.e., global k-skyband objects) at line 1, and then maintain a minimum heap H with entries in the form (e, *key*) (line 2), where e is an MBR node of index and *key* is defined as the minimum between q and e. Every time we pop out an entry (e, *key*) from H with the minimum key value (line 4). Then, we verify whether or not e is globally dominated by k+1 objects in S_c. If it is true, then e can be safely pruned; otherwise, we process e as follows.

If e is an intermediate node, we check each child e_i of e to judge whether it is globally dominated by k+1 objects in S_c (line 8). If it is true, we discard e_i; otherwise, we insert e_i into auxiliary heap H (line 9). Next, we update the reuse heap H_r at line 10. Note that the line is very important. It guarantees that all nodes accesses are saved in H_r and there are no redundant nodes. In order to shrink the capacity of H_r, we might only insert intermediate node into H_r. In addition, in order to improve the efficiency of search, we adopt the binary search method. If e is a data object, we insert e into S_c which is a candidate set to save global k-skyband objects (line 12). Then, we invoke *k-WindowQuery* procedure at line 13 to judge whether or not the window $w(e, q)$ contains more than k other data objects. If it is true, we mark e as false alarm by line 15; otherwise, e will be returned as query results at line 16.

Algorithm 1. RR*k*SB(T, q, k, S_r)
Input: T–index; q–query object
Output: S_r–result set

```
1. $S_r = \emptyset$, $S_c = \emptyset$, $H = \emptyset$, $H_r = \emptyset$
2. Insert all entries of root into $H$, $H_r$ with ($e$, $mindist(e, q)$)
3. WHILE $H$ is not empty DO
4.   Pop the entry ($e$, $mindist(e, q)$) of $H$
5.   IF $e$ is not globally dominated by ($k$+1) points in $S_c$
6.     IF $e$ is an intermediate entry THEN
7.       FOR each child entry $e_i \in e$ DO
8.         IF $e_i$ isn't globally dominated by ($k$+1) points in $S_c$
9.           Insert ($e_i$, $mindist(e_i, q)$) into $H$
10.        Insert all $e_i \in e$ into $H_r$, Delete $e$ from $H_r$
11.      ELSE //$e$ is a data object
```

```
12.    $S_c = S_c \cup \{e\}$ //insert $e$ into global $k$-skyband
13.    $rflag$ = **k-WindowQuery**($T, e, q, H_r$)
14.    IF $rflag$==TRUE THEN//contain (>$k$+1) other objects
15.        Mark $e$ as false alarm
16. RETURN all not false alarm points in $S_r$
```

The window query procedure, namely, *k-WindowQuery*, is described in detail as algorithm 2. It first initializes a auxiliary heap H and a set variable S_w which is used to save the objects in window $w(p, q)$ at line 1, and then inserts all entries which intersects with $w(p, q)$ into H by line 2. Next, algorithm 2 uses *best first* search policy to obtain all objects in window $w(p, q)$ and inserts them into S_w. When the number of objects in S_w is great than k, algorithm 2 returns FALSE; otherwise, it return TRUE until algorithm 2 finishes the search (line 14). Lines 10-13 extends the intermediate node and inserts all child entries which is inside or overlap with $w(p, q)$ into H. But, note that the *key* in algorithm 2 becomes the distance between p and current e, namely, *minidist*(e, p). Since algorithm 2 can stop the search, it does not has to search all nodes in index and obtain a higher efficiency.

Algorithm 2. k-WindowQuery(p, q, H_r, k)

Input: p–current object; q–query object; H_r–reuse heap, k–user specified parameter

Output: a Boolean value to indicate whether the current window $w(p, q)$ contains more than k objects

```
1. $H = \emptyset$, $S_w = \emptyset$, get query window $wq(p, q)$
2. insert entries of $H_r$ which intersect with $w(p, q)$ into $H$ with the form ($e$, $minidist(e, p)$)
3. WHILE $H$ is not empty DO
4.   de-heap the top entry ($e$, $mindist(e, p)$) from $H$
5.   IF $e$ is a data object THEN
6.     IF $e \neq p$ and $e$ is contained in $wq(p, q)$ THEN
7.       insert $e$ into $S_w$ //$S_w$ saves the objects in $w(p, q)$
8.     IF $|S_w|>k$ THEN RETURN FALSE
9.   ELSE //intermediate entry
10.    IF $wq(p, q)$ is inside or overlap with $e$ THEN
11.      FOR each child entry $e_i \in e$ DO
12.        IF $wq(p, q)$ is inside or overlap with $e_i$ THEN
13.          insert the entry ($e_i$, $minidist(e_i, p)$) into $H$
14. RETURN TRUE
```

By above analysis, now we immediately obtain the following important Theorem 1 and Theorem 2.

Theorem 1. For a given query object q, RRkSB algorithm correctly returns all results of reverse k-skyband, and does not product false alarms or false dismissals.

Proof. According to Definition 2 and Definition 3, we know that the candidate set of global k-skyband, Sc, will correctly save the results of global k-skyband because RRkSB uses the method of global dominating at line 5 and line 8. On the other hand, it is equivalent to obtain the node information by using reuse heap H_r and accessing index. For example, in Figure 2, N_5 corresponds to data objects f and g, N_1 corresponds to data objects a, b, c, d, and e, and N_5 corresponds to data objects h, i, and j when the nodes among H_r are N_5, N_1, and N_6. Thus, the procedure of k-WindowQuery and the window query of k-skyband have the same effect. Therefore, Theorem 1 is correct.

Theorem 2. For the entries in reuse heap H_r, BRkSB need to access multiple times disk, yet RRkSB only accesses disk one time.

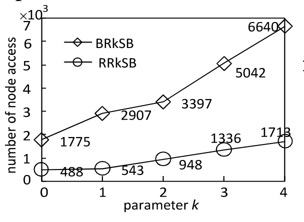
Proof. According to the process of reverse k-skyband, we know that it is divided two stages, namely, global k-skyband and k-window query. Since the nodes among reuse heap H_r are inserted after the first access, RRkSB can directly fetch these nodes from H_r when they are accessed later. Thus, RRkSB only need to access disk one time. However, BRkSB will access these nodes in H_r at both the stage of global k-skyband and the stage of k-window query. Therefore, BRkSB requires at least two disk accesses. For instance, consider the example in Figure 2, H_r becomes g, N_1, N_6, and f when RRkSB obtains the first object of global 1-skyband, g. Then, the window queries of the objects b, c, d and e require to access node N_1 one time, respectively. Consequently, BRkSB produces 5 I/O accesses. So, Theorem 2 is correct.

IV. Experimental Evaluation

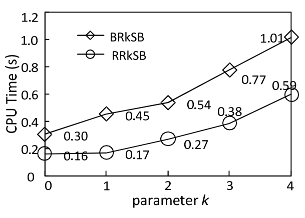
In this section, we experimentally evaluate the performance of BRkSB [4] and RRkSB algorithms for RkSB retrieval over both real and synthetic datasets. The real dataset *Color* (*Col*) includes 3D vectors representing the color histograms of 68K images [5]. We also create a 3D synthetic dataset *Correlated* (*Corr*) used in traditional skyline. The values of the attributes are in the range [1, 10000]. In each experiment we performed 50 reverse skyline queries to the particular data set and reported the overall result. Each dataset is indexed by an R-tree, where the page size is set to 4KB in all cases. All algorithms were implemented in C++, and all experiments were conducted on an Intel Dual Core 2.0 GHz PC with 4GB RAM.

A. The Effect of parameter k

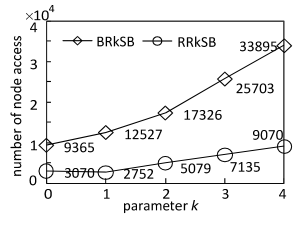
The first set of experiments studies the performance of proposed algorithms as the function of parameter k. Figure 3 shows the results. With the increase of k, the I/O cost quickly grows, but CPU time only slowly increases. This is because there are more query results when the parameter k increases. In most of cases, the I/O cost of RRkSB is 1/5 that of BRkSB and CPU time of RRkSB is about 3/5 that of BRkSB. This indicates that RRkSB has a better performance than BRkSB on parameter k.



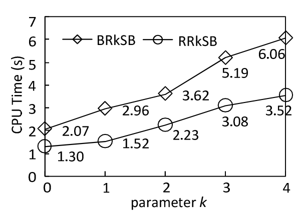
(a)



(b)



(c)



(d)

Figure 3. The comparison of the performance for the algorithms vs. *k*. (a) I/O cost on *Col*, (b) CPU Time on *Col*, (c) I/O cost on *Corr*, (d) CPU Time on *Corr*.

B. *The Effect of dataset size*

The second set of experiments reports the influence of dataset size on the algorithms, varying the cardinality of *Col* from 12K to 60K, the cardinality of *Corr* from 128K to 2048K, respectively. Figure 4 gives the results. Although the cost of the algorithms both grows with the increase the size of dataset, the cost on *Col* has a slower raise than that on *Corr*. The reason lies in the different distribution of two datasets. Whichever dataset is used, RR*k*SB gains better performance than BR*k*SB.

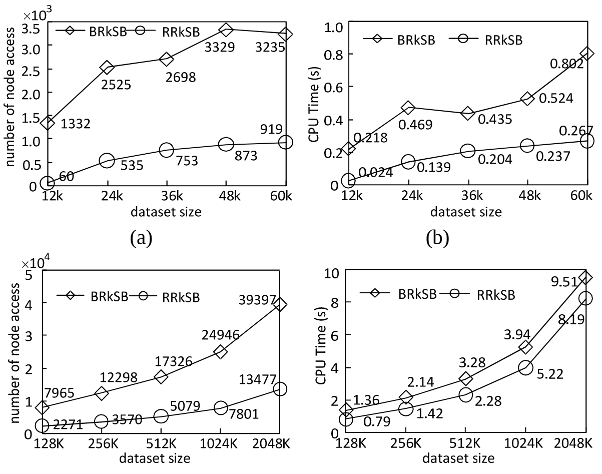
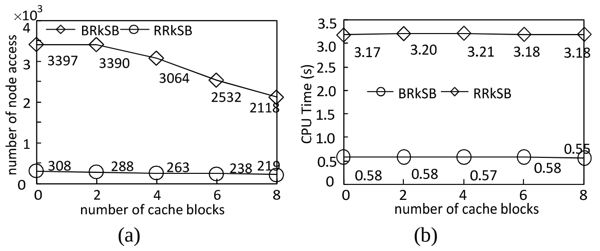


Figure 4. The comparison of the performance for the algorithms vs. *dataset size*. (a) I/O cost on *Col*, (b) CPU Time on *Col*, (c) I/O cost on *Corr*, (d) CPU Time on *Corr*.

C. *The Effect of cache size*

The third serial of experiments aims to evaluate the effect of cache size on the algorithms. We change the number of cache blocks from 0 to 4, where each block contains 4096 bytes. Figure 5 plots the cost of the algorithms w.r.t. the number of cache blocks. Clearly, we find that the I/O cost slowly decreases with the increase of the number of cache blocks; yet there is no change on CPU time over two datasets. This verifies the usefulness of our proposed reuse technology. The reason lies that cache cannot forecast next cache block used by the algorithms, and reuse method save the next block into reuse heap. Comprehensively, RR*k*SB obtains better performance than BR*k*SB.



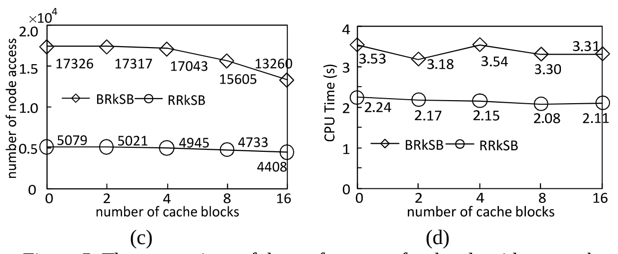


Figure 5. The comparison of the performance for the algorithms vs. *the number of cache blocks*. (a) I/O cost on *Col*, (b) CPU Time on *Col*, (c) I/O cost on *Corr*, (d) CPU Time on *Corr*.

V. CONCLUSION

This paper introduces a novel algorithm to answer the reverse *k*-Skyband query, namely, reuse R*k*SB (RR*k*SB). The main ideas are reuse information technology and early stopping computation. The experiments conducted on real and synthetic datasets demonstrate the efficiency and effectiveness of RR*k*SB, compared with basic R*k*SB algorithm (BR*k*SB) [4].

ACKNOWLEDGMENT

This work was supported by NSFC 61003049, the Research Innovation Team Funds for College Students of Zhejiang Province, ZJNSF LY12F02047 and LY12F02019, 2011AY1005, the Fundamental Research Funds for the Central Universities under grant 2010QNA5051.

REFERENCES

[1] I. Bartolini, Z. Zhang, and D. Papadias, “Collaborative Filtering with Personalized Skylines,” IEEE Trans. Knowl. Data Eng., vol. 23, Feb 2011, pp. 190-203, doi: 10.1109/TKDE.2010.86

[2] K. Hose and A. Vlachou, “A Survey of Skyline Processing in Highly Distributed Environments,” The VLDB Journal, vo. 21, June 2012, pp. 359-384, doi: 10.1007/s00778-011-0246-6.

[3] D. Papadias, Y. Tao, G. Fu, and B. Seeger, “Progressive Skyline Computation in Database Systems,” ACM Trans. on Database Syst., vol. 30, March 2005, pp. 41-82, doi: 10.1145/1061318.1061320.

[4] Q. Liu, Y. Gao, G. Chen, and Q. Li, “On Efficient Reverse *k*-Skyband Query Processing,” Proc. of DASFAA Conf., Springer, April 2012, pp. 544-559, doi: 10.1007/978-3-642-29038-1_39.

[5] Y. Tao, X. Xiao, and J. Pei, “Efficient Skyline and Top-*k* Retrieval in Subspaces,” IEEE Trans. Knowl. Data Eng., vol. 19, Aug. 2007, pp. 1072-1088, doi: 10.1109/TKDE.2007.1051.

[6] E. Dellis and B. Seeger, “Efficient Computation of Reverse Skyline Queries,” Proc. of VLDB Conf. (VLDB07), VLDB Endowment, Sep. 2007. pp. 291-302.

[7] L. Chen and X. Lian, “Monochromatic and Bichromatic Reverse Skyline Search over Uncertain Databases,” Proc. of ACM SIGMOD, ACM Press, June. 2008. pp. 213-226, doi: 10.1145/1376616.1376641.

[8] J. Lee and S. Hwang, “QSkycube: Efficient Skycube Computation using Point-Based Space Partitioning,” Proc. of the VLDB Endowment, vo. 4, Dec. 2010, pp. 185-196.

[9] A. Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching,” ACM SIGMOD Record, vol. 14, June 1984, pp. 47−57, doi:10.1145/971697.602266.