G-Tree: A Novel Index Structure for Subspace Skyline Query

Yi-Chung Chen and Chiang Lee

***Abstract*—The skyline search algorithm has recently emerged as an important technique in database research. Given a set of data points in a multidimensional database, such queries return points that are not "dominated" (detailed in the paper) by any other point. In practice, databases that require a skyline query usually provide numerous candidate dimensions, of which users are interested in only a few. As a result, queries are issued regarding various subsets of the dimensions and such queries are called subspace skyline queries.**

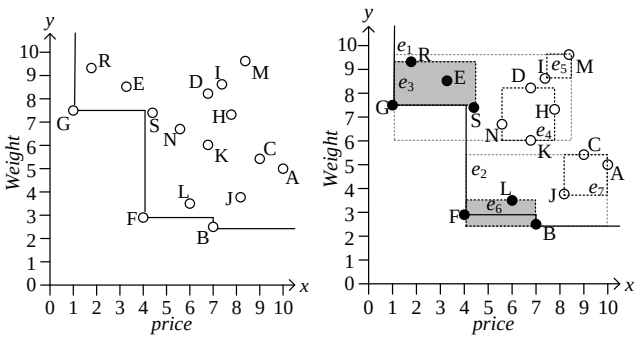
Using the conventional skyline algorithm to process these queries directly can be extremely ineffective. Additional algorithms and architectures have been added to improve search efficiency; however, such modifications can increase computational costs or necessitate an increase in data storage capacity. This paper proposes a novel index model based on a Gaussian function to enhance the performance of subspace skyline queries. Simulation results demonstrate the efficacy of the proposed tree in locating skyline points within a subspace.

***Index Terms*—Database, skyline, tree structure.**

I. Introduction

The skyline search algorithm [1]-[5] has moved to the forefront of database research due to its wide applicability in multi-criteria decision-making environments. Take for example the cell phone dataset in Fig. 1(a), which includes two attributes: the price (*x* axis) and weight (*y* axis) of cell phones. If a consumer were looking for a light, inexpensive cell phone, then cell phone F would be better than cell phone L because F is cheaper and lighter than L (i.e., F *dominates* L). Phone B is lighter but more expensive than phone F; therefore, these cell phones cannot be compared (i.e., *incomparable*). Skyline queries operate by locating all non-dominated data points. In this example, the *skyline points* include B, F, and G because these cell phones are not dominated by any other cell phones. The line connecting all skyline points is called the *skyline*.

A major bottleneck in processing skyline queries is the fact that all of the data points must be loaded into the algorithm at least once, which means that the I/O costs can be astronomical [4]-[6]. The most common method [6], [7] of processing skyline queries is to first employ the *R*-tree [8]-[11] to index data points, as shown in Fig. 1(b). In this figure, similar data points are stored in the same node (i.e., the rectangle in Fig. 1(b)). Through the *R*-tree, the algorithm knows that only the data points in the nodes intersecting the skyline (i.e. within the grey rectangle) are potential skyline answers. None of the other data points could be skyline solutions. In other words, obtaining skyline results requires only that the algorithm load data points from within the grey rectangle, which is far fewer than the total number of data points in the set. This significantly reduces the I/O costs of the overall algorithm.



(a) Original dataset. (b) An index result of the *R*-tree.

Fig. 1. The cell phone dataset.

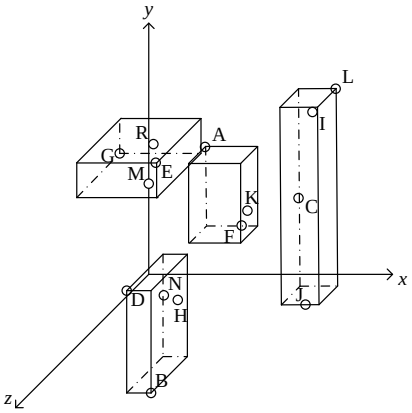
Subspace skyline queries [12]-[17] are an important topic in skyline queries. To define a subspace skyline query, assume that the entire (the full space dataset) has *d* dimensions, and the algorithm obtains skyline results using only *n* ($n<d$) dimensions in the subspace rather than using all *d* dimensions. For example, a hotel database usually has more than five dimensions, such as room prices, room sizes, restaurant prices, hotel ranking, restaurant ranking, distance to the beach, distance to scenic spots, and distance to the airport. However, a user may only be interested in room price and distance to the airport, while another user would want to learn about room price, hotel ranking, and distance to scenic spots. Traditional skyline algorithms consider all dimensions, yet these two cases consider subsets of the dimensions, possibly resulting in entirely different skyline points. As a result, new algorithms must be derived for subspace skyline queries.

The most intuitive method used to process subspace skyline queries is to build an *R*-tree of all data points in the entire space and then use this tree to search for the subspace skyline [12]. However, this method performs poorly because the data points in a single *R*-tree node are similar in the full space but not necessarily similar within the subspace (see Fig. 2). Fig. 2(a) presents an *R*-tree constructed using a three-dimensional data set (*X*, *Y* and *Z*) (where the *R*-tree is indicated by R_a). Fig. 2(b) presents the result of projecting the data points from Fig. 2(a) and R_a onto the surface of *XY*. Fig. 2(c) presents an *R*-tree constructed using the *XY* surface, where the *R*-tree is indicated by R_b. It is clear that in R_a, the data points within a single node are more scattered, compared to those in R_b. When reading the

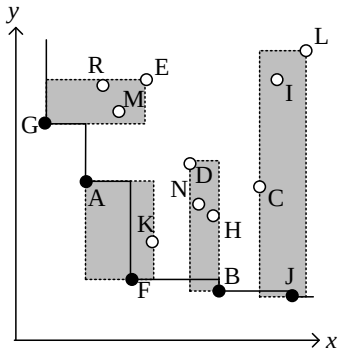
Manuscript received June 17, 2013; revised August 12, 2013.

Y. C. Chen and C. Lee are with the Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan, 701, Taiwan, R.O.C. (e-mail: mitsukoshi901@dblab.csie.ncku.edu.tw, leec@mail.ncku.edu.tw).

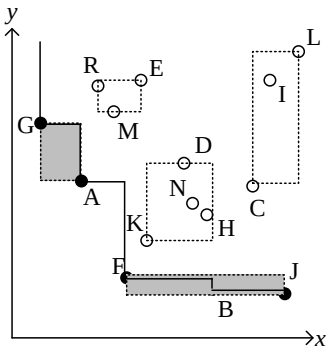
nodes that intersect with the skyline (i.e., the gray rectangles in Figs. 2(b) and 2(c)), the algorithm will read more nodes from R_a than from R_b. Therefore, using R_a to search for skyline results would be much more inefficient than using R_b. Current methods of processing subspace skyline queries use additional algorithms and architectures to accelerate the search process. Tao *et al*. [12] added the anchors algorithm to the conventional skyline algorithm to reduce the number of data points to be read when searching for the subspace skyline, thereby improving search efficiency. However, as this method still generates additional operational costs, the degree to which it actually improves the efficacy of subspace skyline queries is limited. Jin *et al*. [13] first obtained answers to all subspaces and then stored these answers using an additional indexing architecture. This approach allows users to immediately obtain the results they desire after inputting the search; however, it also requires significantly greater storage capacity for the additional index architecture. The current study proposes a novel subspace skyline algorithm capable of overcoming the weaknesses encountered using previous methods.



(a) A three dimensional dataset and its index result of the *R*-tree.



(b) The result of projecting the data points from (a) and R_a onto the surface of *XY*.



(c) An *R*-tree constructed using the data points on *XY* surface.

Fig. 2. An example for explaining the difficulty that *R*-tree meets in the subspace skyline query.

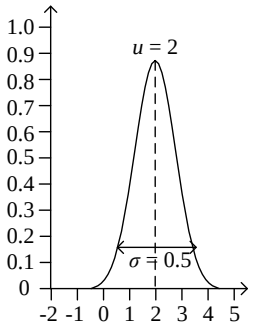
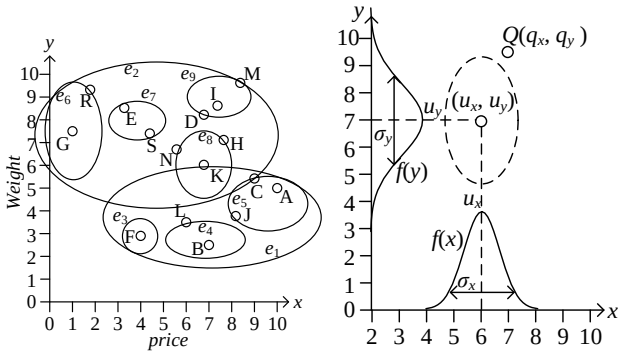


Fig. 3. An example of the Gaussian function.



(a) An index result of the *G*-tree. (b) An example of MBG.

Fig. 4. An example of the *G*-tree.

We designed a new index tree framework called the Gaussian tree (*G*-tree) to improve the efficiency of subspace skyline queries. All data points within a single *G*-tree node are extremely similar, whether in full space or various subspaces. Therefore, the *G*-tree can overcome the problems encountered when using the conventional *R*-tree in processing subspace skylines. When a *G*-tree constructed in the full space is used to search for a subspace skyline, the algorithm reads a limited number of data points, thereby improving search efficiency. Another advantage of using the *G*-tree to process subspace skyline queries is that we need only modify the conventional skyline algorithm rather than incorporating additional algorithms or architectures. This study employed a number of simulations to verify the efficiency of the *G*-tree to process subspace skyline queries.

The remainder of this paper is organized as follows: Section II outlines the design of the *G*-tree, while Section III explains the methods of using the *G*-tree to find the subspace skyline points. Section IV provides experimental results and lastly Section V presents the conclusions of this study.

II. *G*-Tree

Two parts are discussed in this section, which are the extensions of our previous work [18]. These parts introduce the structure and the construction algorithm of the *G*-tree.

A. G-Tree Structure

The *G*-tree was built using the Gaussian function shown in Fig. 3; this function can be described as

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-u)^2/2\sigma^2} \quad (1)$$

Parameter u in (1) is the mean of the function used to

determine the location of the curve. In (1), σ is the standard deviation of the function used to determine the width of the curve.

Fig. 4(a) presents the *G*-tree results built on the scenario in Fig. 1(a). Note that for ease of explanation, we employed only a two-dimensional example to introduce this tree. Structures with more dimensions could easily be extrapolated from this example. The *G*-tree has two types of nodes: internal nodes and leaf nodes. In Fig. 4(a), e_1 is an example of an internal node storing other internal nodes, such as e_3, e_4, and e_5. An example of a leaf node is e_5, which stores data points such as A, C, and J. All nearby data points in Fig. 4(a) are included within the same oval. In this paper, this oval is called the Minimum Bounding Gaussian function (MBG). Every MBG in Fig. 4(b) comprises two Gaussian functions $f(x)$ and $f(y)$, in which the mean and standard deviations are (u_x, σ_x) and (u_y, σ_y). Thus, the correlation between any point $Q(q_x, q_y)$ in the database and this MBG can be calculated as follows:

$$\mathbf{P}q = \frac{1}{2\pi\sigma_x\sigma_y}\left(e^{-\left(\sigma_y^2(q_x-u_x)^2+\sigma_x^2(q_x-u_x)^2\right)\Big/2\sigma_x^2\sigma_y^2}\right) \qquad (2)$$

The dotted line in Fig. 4(b) represents the boundary of the MBG. The correlation between this node and any point on the line should be α, the value of which is defined by the user.

Equation (2) also indicates that data point Q (q_x, q_y) must be very close to the MBG center (u_x, u_y) on both the x and y axes in order to obtain a Pq greater than 0 and be added to the MBG. If there is a significant difference between Q and the MBG center of any dimension, then Pq will approach 0 and Q will not be inserted in this MBG. This design ensures that the dimensional values of data points within a single MBG will closely approach the value of the MBG center. Even if we reflect this MBG in another subspace, the data points will still be close in value to the MBG center. In this manner, the *G*-tree can overcome problems encountered when using *R*-tree to process subspace skyline queries.

B. G-Tree Construction

When building the *G*-tree, the algorithm inserts data points into the tree one at a time. Each inserted data point is processed, as shown in Fig. 5.

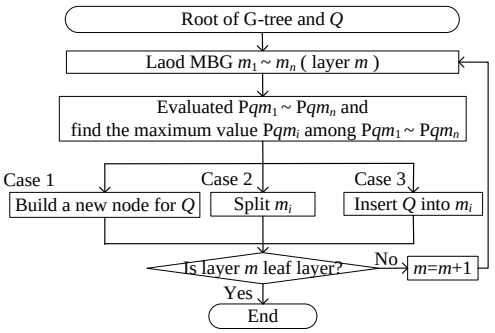


Fig. 5. The flow chat of the *G*-tree construct algorithm.

Assuming that the search point is Q (q_x, q_y), the algorithm starts at the root node and hierarchically searches for the node into which Q should be inserted. This process continues until Q is inserted into a leaf node. Assuming that the level to be searched is m with nodes m_1 to m_n, the algorithm will separately compute the possibility that Q falls into these nodes, indicated as Pqm_1 to Pqm_n. Assuming that Pqm_i is the greatest of these possibilities, the following steps are taken based on Pqm_i and the value α of the user-defined MBG boundary:

Case 1: Pqm_i < α. In this case, Q is not correlated with the nodes of layer m and therefore a Q-centered MBG m_e is constructed at level m. After m_e is constructed, the algorithm searches for the parent node of m_e at level m-1 and establishes the relationship between m_e and its parent node. If a parent node for m_e cannot be found at level m-1, the algorithm constructs an m_e-centered parent node at level m-1.

Case 2: Pqm_i ≥ α, and m_i has not reached maximum capacity. In this case, Q is directly inserted into m_i.

Case 3: Pqm_i ≥ α, and m_i has reached maximum capacity. In this situation, Q is inserted into m_i and then m_i is split into m_{i1} and m_{i2}. The center of m_i is the central point of m_{i1} while the furthest point of m_i is considered as the center of m_{i2}. Next, the data points surrounding m_i are re-allocated to the nodes to which they are most strongly correlated.

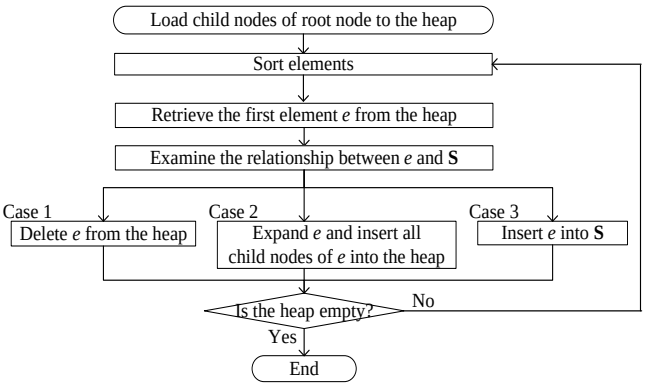


Fig. 6. The flow chat of using *G*-tree to find the subspace skyline points.

III. USING *G*-TREE TO FIND THE SUBSPACE SKYLINE POINTS

Fig. 6 presents a flow chart illustrating the use of the *G*-tree to locate subspace skyline points. This is an extension of the algorithm in [6], employing three data structures: a *G*-tree, a heap, and a list. The heap stores temporary data as points or MBGs. For ease of explanation, we will refer to these as elements. The list stores subspace skyline points; sets of which are designated by **S** in the following. MBGs beneath the root node are first input into the heap and arranged in ascending order according to their summation of coordinates within the subspace. It should be noted that the summation of an MBG is equal to the smallest summation of coordinates related to a point in the subspace. The algorithm then retrieves the first element e from the heap and checks for domination using each point in **S**. The three possible results are listed in the following:

Case 1: If e is dominated by any point in **S**, then e is deleted from the heap.

Case 2: If e cannot be dominated by any point in **S** and is an MBG, then e is expanded and all internal nodes or data points within it are placed into the heap.

TABLE I: EXAMPLE OF USING *G*-TREE TO FIND SUBSPACE SKYLINE POINTS

Step	Heap	Action	Skyline (**S**)
1	Φ	Access root	Φ
2	<e_1, 6.5> <e_2, 8>	Expand e_1	Φ
3	<e_3, 6.8> <e_2, 8> <e_4, 9> <e_5, 11>	Expand e_3	Φ
4	<F, 6.8> <e_2, 8> <e_4, 9> <e_5, 11>	Move F to **S**	{F}
5	<e_2, 8> <e_4, 9> <e_5, 11>	Expand e_2	{F}
6	<e_6, 8> <e_4, 9> <e_5, 11> <e_7, 11.8> <e_8, 12> <e_9, 15>	Expand e_6	{F}
7	<G, 8> <e_4, 9> <e_5, 11> <R, 11.2> <e_7, 11.8> <e_8, 12> <e_9, 15>	Move G to **S**	{F, G}
8	<e_4, 9> <e_5, 11> <R, 11.2> <e_7, 11.8> <e_8, 12> <e_9, 15>	Expand e_4	{F, G}
9	<B, 9> <L, 9.2> <e_5, 11> <R, 11.2> <e_7, 11.8> <e_8, 12> <e_9, 15>	Move B to **S**	{F, G, B}
10	<L, 9.2> <e_5, 11> <R, 11.2> <e_7, 11.8> <e_8, 12> <e_9, 15>	Delete L from heap	{F, G, B}
11	<e_5, 11> <R, 11.2> <e_7, 11.8> <e_8, 12> <e_9, 15>	Delete e_5 from heap	{F, G, B}
12	<R, 11.2> <e_7, 11.8> <e_8, 12> <e_9, 15>	Delete R from heap	{F, G, B}
13	<e_7, 11.8> <e_8, 12> <e_9, 15>	Delete e_7 from heap	{F, G, B}
14	<e_8, 12> <e_9, 15>	Delete e_8 from heap	{F, G, B}
15	<e_9, 15>	Delete e_9 from heap	{F, G, B}
16	Φ	End	{F, G, B}

Case 3: If *e* cannot be dominated by any point in **S** and is a data point, then *e* is input into S.

Once the heap is empty, the algorithm is terminated and **S** is retained as the subspace skyline answer.

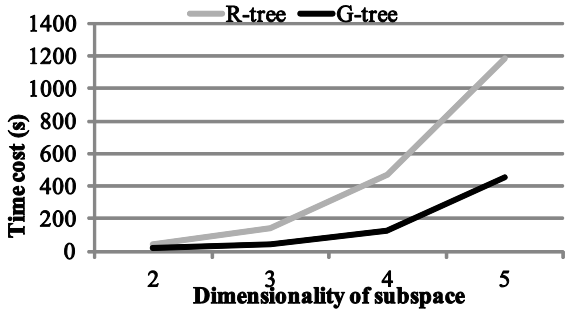
Table I is an extension of Fig. 4(a) to further illustrate the processes involved in this algorithm. In Step 1, the root node of the *G*-tree is extracted and e_1 and e_2 are added to the heap in accordance with their summations. In Step 2 the first element in the heap (i.e., e_1) is retrieved. Because **S** is Φ, e_1 cannot be dominated by the existing skyline (Case 2). Thus, e_1 is expanded into e_3, e_4, and e_5. These MBGs are inserted into the heap according to their summations. Through steps 3 – 16, the above-described processes are repeated. As a further interpretation of Cases 3 and 1, Steps 4 and 10 are detailed in the following.

In step 4, the algorithm extracts the first element F from the heap for processing. However, because **S** is Φ, F cannot be dominated by the existing subspace skyline point, which matches the scenario in Case 3. Therefore, we insert F into **S** and eliminate **S** from the heap.

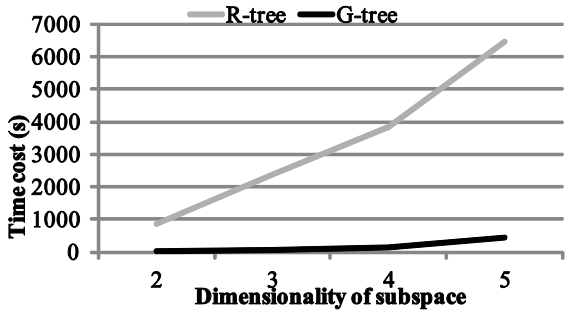
In Step 10, the algorithm extracts the first element F from the heap for processing. However, according to Fig. 4(a) we know that L is dominated by F, a scenario which matches Case 1. Therefore, L cannot be the subspace skyline point and is directly eliminated from the heap.

IV. PERFORMANCE

This section uses an experiment to verify the efficiency of the *G*-tree, using two types of data distribution [1], [6], [19], [20] frequently employed for skyline purposes: independent datasets and the anti-correlated datasets. Generally, anti-correlated datasets are the worst case for the skyline problem, producing the highest number of skyline results. The independent dataset is considered a general case. The simulation first built a six-dimensional independent dataset and anti-correlated dataset. Each dataset included 1 million items of data. We then constructed an *R*-tree and *G*-tree for each of the datasets and used these two trees to obtain subspace skyline points with 2-5 dimensions. Each performance curve shown in the figures represents an average of the experimental results of 30 datasets [21]. All of the experiments were performed on an Intel i7-3770 CPU at 3.40GHz with 4GB main memory, running on Microsoft Windows XP. All the programs were written in MATLAB®



(a) Independent dataset.



(b) Anti-correlated dataset.

Fig. 7. Comparing the time cost of the *R*-tree and *G*-tree by varying the dimensionality of subspace.

Fig. 7 illustrates the time required to obtain subspace skyline points with 2-5 dimensions using *R*-tree and *G*-tree. Fig. 7(a) shows the results of the independent dataset, while Fig. 7(b) displays the results of the anti-correlated dataset. From these two figures we can see that regardless of whether the *R*-tree or the *G*-tree was used, the time required to complete the algorithm increased with an increase in dimensions. This is because the number of skyline points grows with an increase in dimensions. The greater the number of skyline points, the more time required for the algorithm to identify all points. The figures also show that compared to the

G-tree, the *R*-tree time curve rises more quickly as dimensions increase. This is because when the algorithm employs a 6-dimenisonal *R*-tree to locate subspace skyline points with 2-5 dimensions, the data points of a single *R*-tree node may become dissimilar within the subspace. Although the algorithm reads only the nodes that intersect with the skyline, it is still necessary to read many far-off data points (i.e. data points with no possibility of being skyline data points), which significantly increases the I/O cost of the algorithm. The *G*-tree does not present this difficulty. When using a *G*-tree constructed on full space, we found that the data points of a single node in any type of subspace are always similar. Therefore, the nodes that intersect the skyline and are read by the algorithm are all close to the skyline (i.e. possible skyline data points), thereby controlling the I/O cost of the algorithm. These simulation results verify the efficiency of using the *G*-tree to process subspace skyline queries.

V. Conclusion

This paper proposes a novel index structure called *G*-tree to enhance the performance of subspace skyline queries. We began by outlining the difficulties involved in applying the *R*-tree in subspace skyline queries. We then illustrated the structure of the algorithm used to construct the *G*-tree. Finally, simulation results demonstrate the efficiency of the proposed *G*-tree in processing subspace skyline queries. In future work, we will apply the *G*-tree to other problems to verify whether the *G*-tree outperforms the R-tree in all types of skyline queries.

Acknowledgment

This work was supported by NSC under the Grant NSC 100-2221-E-006-250-MY3.

References

[1] S. Borzsonyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proc. ICDE*, 2001, pp. 235-254.
[2] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *Proc. ICDE*, 2003, pp. 717-719.
[3] P. K. Eng, B. C. Ooi, and K. L. Tan, "Indexing for progressive skyline computation," *Data & Knowledge Engineering*, vol. 46, pp. 169-201, 2003.
[4] I. Bartolini, P. Ciaccia, and M. Patella, "SaLSa: Computing the skyline without scanning the whole sky," in *Proc. CIKM*, 2006, pp. 405-414.
[5] I. Bartolini, P. Ciaccia, and M. Patella, "Efficient sort-based skyline evaluation," *ACM Transactions on Database Systems*, vol. 33, no. 4, pp.31-48, 2008.
[6] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An optimal and progressive algorithm for skyline queries," in *Proc. SIGMOD*, 2003.
[7] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: an online algorithm for skyline queries," in *Proc. VLDB*, 2002.
[8] A. Guttman, "*R*-trees: A dynamic index structure for spatial searching," *SIGMOD Record*, vol. 14, no. 2, pp. 47-57, 1984.
[9] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," in *Proc. SIGMOD*, 1990.
[10] S. Li, D. Zhao, and K. Bai, "A new approach of R-tree construction," in *Proc. ICCIS, 2012*, pp. 684-687.
[11] G. Li and J. Tang, "A new R-tree spatial index based on space grid coordinate division," in *Proc. ICCE*, 2011.
[12] Y. Tao, X. Xiao, and J. Pei, "SUBSKY: Efficient computation of skylines in subspaces," in *Proc. ICDE*, 2006, pp. 65-74.
[13] W. Jin, A. K. H. Tung, M. Ester, and J. Han, "On Efficient Processing of Subspace Skyline Queries on High Dimensional Data," in *Proc. SSDBM*, 2007.
[14] J. Pei, Y. Yuan, X. Lin, W. Jin, M. Ester, Q. Liu, W. Wang, Y. Tao, J. X. Yu, and Q. Zhang, "Towards multidimensional subspace skyline analysis," *ACM Transactions on Database Systems*, vol. 31, no. 4, pp. 1335-1381, 2006.
[15] M. Bai, J. Xin, and G. Wang, "Subspace global skyline query processing," in *Proc. EDBT*, 2013.
[16] J. Kim, J. Lee, and S. W. Hwang, "Skyline view: Efficient distributed subspace skyline computation," in *Proc. DaWaK*, 2009.
[17] A. Vlachou, C. Doulkeridis, Y. Kotidis, and M. Vazirgiannis, "SKYPEER: Efficient subspace skyline computation over distributed data," in *Proc. ICDE*, 2007, pp. 416-425.
[18] Y. C. Chen, H. C. Liao, and C. Lee, "A novel g-tree for accelerating the time-consuming skyline query," in *Proc. IKE*, 2013.
[19] K. L. Tan, P. K. Eng, and B. C. Ooi, "Efficient progressive skyline computation," in *Proc. VLDB*, 2001, pp. 301-310.
[20] M. L. Yiu, E. Lo, and D. Yung, "Measuring the sky: On computing data cubes via skylining the measures," *IEEE Trans. Knowledge and Data Engineering*, vol. 24, no. 3, pp. 492-503, 2012.
[21] Y. L. Hsu and J. S. Wang, "A Wiener-Type recurrent neural network and its control strategy for nonlinear dynamic applications," *Journal of Process Control*, vol. 19, no. 6, pp. 942-953, 2009.

Yi-Chung Chen is a PhD student in the Department of Computer Science and Information Engineering at National Cheng-Kung University, Taiwan. His research interests include databases and artificial intelligences.

Chiang Lee received the BS degree from National Cheng-Kung University, Taiwan, in 1980 and the ME and PhD degrees in electrical engineering from the University of Florida, Gainesville, in 1986 and 1989, respectively. He joined IBM Mid-Hudson Laboratories, Kingston, New York, in 1989 and participated in a project working on the design and performance analysis of a parallel and distributed database system. He joined the faculty of National Cheng-Kung University in 1990 and is currently a professor of the Department of Computer Science and Information Engineering. He has published many papers in major journals and conferences, and has been invited as an author of a chapter for several technical books.