An Efficient Query Processing Algorithm for Multiple Constrained Skyline Queries

I-Fang Su, Yu-Chi Chung, Yuan-Ko Huang, and Chang-Ming Tsai

***Abstract*—A multi-query optimization issue for constrained skyline query processing is studied in this paper. In traditional skyline query processing, each query is processed independently. In this paper, we exploit the dependencies among a collection of concurrent constrained skyline queries and design a framework to speed-up the processing of multiple constrained skyline queries. To the best of our knowledge, this is the first work to study the multi-query optimization for multiple constrained skyline queries. Based on the BBS (Branch and Bound Skyline) algorithm, we design a Multiple-Constrained-Skyline-Query (MCSQ) algorithm than can speed-up the query processing performance for multiple constrained skyline queries. We also conduct a series of experiments to evaluate the performance of our design. The result shows the efficiency of our algorithm.**

***Index Terms*—Constrained skyline queries, database, query processing, multi-query optimization.**

I. Introduction

In this paper, we propose an algorithm that can process multiple constrained skyline queries simultaneously. Skyline query processing is widely used for business decision making, or data analysis [1]. Many skyline query processing algorithms [1]-[7] have been proposed over the past few years. The difference between our algorithm and the previous methods is that the latter focused on processing each skyline query independently. However, the proposed algorithm can process a batch of skyline queries simultaneously. The advantages of our algorithm are decreased query processing time, and disk I/O cost, which in turn improves system scalability. In the following, we first introduce the idea of the constrained skyline queries and then explain the design of the proposed algorithm.

A typical skyline query example is that when a tourist wants to make a hotel reservation for a holiday in Maldives, the two main factors to be considered are the hotel rate (price per night) and distance from the beach. For example in Fig. 1, the x-axis and y-axis represent the distance and the price, respectively. Given the economic constraints faced by tourists, unlimited expenditure during their hotel stay is not possible. Hence, certain criteria and restrictions have to be set to exclude hotels that are beyond their financial capacity.

Manuscript received May 6, 2013; revised July 24, 2013.

Fang Su is with MIS of Fortune Institute of Technology, Taiwan (ifangsu@fotech.edu.tw).

Yu-Chi Chung and Chang-Ming Tsai are with Department of Computer Science and Information Engineering of Chang Jung Christian University, Taiwan (justim@mail.cjcu.edu.tw, tsai@mail.cjcu.edu.tw).

Yuan-Ko Huang is with Department of Communication of KAO YUAN University, Taiwan (huangyk@cc.kyu.edu.tw).

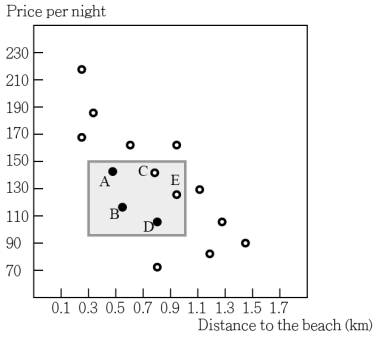

Fig. 1. Example of constrained skyline query processing (hotel finding).

For example, the tourist may wish to select a suitable hotel from those within the price range of 90–150 USD per night and located 0.3–1.0 km from the beach. When these criteria are plotted onto Fig. 1 as well, the result will be as shown in the gray box in Fig. 1. The aim of a skyline query is to determine the most competitive hotel within that two-dimensional gray box, i.e., Hotels A, B, or D (indicated as solid circles).

In order to understand the concept of hotel competitiveness, we must first explain what it means to be non-competitive. If Hotel X is regarded as non-competitive, this means that there exists a Hotel Y that is better than Hotel X on at least one dimension and not worse than Hotel X on the other dimensions. We say that Hotel Y *dominates* Hotel X. Fig. 1 shows that Hotel C is non-competitive since Hotel A is cheaper and close to the beach than Hotel C is. Thus, to be a competitive hotel means that the hotel does not have any *dominators*. And we call these competitive hotels the skyline points.

In the following, we refer to all data found via the skyline query as skyline points. Besides the hotel finding example, skyline queries can also be used in applications involving multi-criteria decision making [1]-[7]. Examples include the purchase of cell phones and procurement of parts.

As stated previously, existing skyline query processing algorithms only consider the effective processing of a single skyline query. This means that the algorithm treats every skyline query as an independent individual and processes it accordingly. Although such a design is simple and easy to implement, the disadvantage is the lack of scalability. This leads to deterioration in efficiency in cases with medium and large loads. We use Fig. 2 to explain it in more detail. Fig. 2 is based on Fig. 1 but with the addition of constrained skyline query of another tourist (i.e., q_2).

Assume that the database system processes q_1 first and then q_2. When q_1 is processed, data points inside the constrained region of q_1 are retrieved from the disk. Each retrieved data point p is checked for dominance. If the dominator is not

DOI: 10.7763/IJEEEE.2013.V3.285

found for p, then p is considered a skyline point. The following facts were determined after processing q_1: 1) data points A–E had already been retrieved from the hard drive and 2) data points C and E are dominated by B.

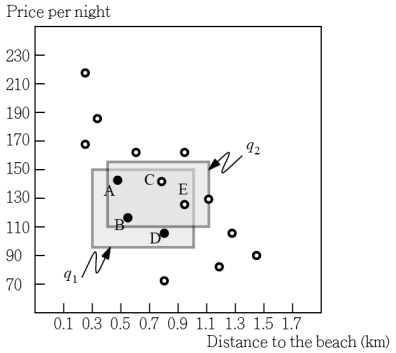

Fig. 2. An example for multiple constrained skyline queries.

After the constrained skyline query (i.e., q_1) for the first tourist is processed, the two facts stated above are "*forgotten*" by the system. When the constrained skyline query (i.e., q_2) for the second tourist is processed next, the system repeats the following actions to determine the skyline results of q_2:

1) The algorithm accesses the disk to retrieve A–E.
2) The algorithm looks for the dominator of every retrieved data point. Again, the algorithm finds out that data point B is the dominator of C and E.

Obviously, the above two actions waste the I/O cost and the CPU cost as the system load increases. When the system receives multiple constrained skyline queries, processing these queries independently leads to unnecessary wastage.

In the paper, we design an algorithm that can merge a group of related constrained skyline queries for processing and thus reduce the disk I/O cost and the CPU cost for processing these queries. This not only accelerates the skyline query processing but also improves system scalability.

To the best of our knowledge, this is the first study that investigates the processing of multiple constrained skyline queries. Although multi-query optimization has been studied for years in the database community, the focus of previous studies has not included skyline query processing. For example, studies have made use of materialized views and re-ordering of query plans to optimize the processing of multiple SQL queries [8], [9]. Other studies investigated to optimize the processing of multiple similarity searches or range queries [10]-[13]. However, none of these techniques can be directly applied to skyline query processing.

The remainder of this paper is organized as follows: previous related studies are briefly reviewed in Section 2; details of the proposed algorithm are discussed in Section 3; details on the experiment are described in Section 4; and Section 5 is the conclusion.

II. Related Works

Many algorithms have been proposed for skyline query processing such as BNL [1], D&C [1], SFS [14], SaLSa [15] and BBS [5]. BNL (block nested loop) compares each data point p with every other data points. If the dominator of p does not exist, then BNL report p as a skyline point. BNL is very efficient when the size of the database is small. D&C employs the *divide and conquer* strategy to process skyline queries. The data space is divided into smaller subspaces. The partial skyline results are computed in each subspace, and the final skyline results are obtained by merging the partial results from every subspace. SFS (sort first skyline) and SaLSa first sort the dataset based on a preference function. The algorithm then scans the sorted dataset to find skyline candidates. While scanning the dataset, a "*stop point*" is set. The stop point guarantees that all data points that are appeared after the stop point should not be skyline points. Thus the algorithm can terminate the search early, resulting in a lower computation cost.

BBS which is currently the most efficient online skyline search algorithm uses iterative nearest neighbor search to find skyline points in a dataset indexed by an *R*-tree. When processing a constrained skyline query, *R*-tree entries intersecting the constrained region are evaluated starting from the root. BBS visits each R-tree entry according its L_1 distance to the origin in the data space. Therefore, the first data point visited by BBS is the first nearest neighbor (nn) to the origin, then the 2nd nn, and so on. Every visited data point is check for dominance. If a data point p falls in the constrained region and no existing data point can dominate p, then BBS reports p as a skyline point.

There are several algorithms designed for constrained skyline queries. Dellis *et al.* [3] proposed STA algorithm to process subspace constrained skyline query. STA consists of two steps: a filter step and a refinement step. In the filter step, STA searches the potential skyline points inside the subspaces that are related to a constrained subspace skyline query. In the refinement step, a potential skyline point is tested for domination. Chen *et al.* [7] proposed PaDSkyline to handle constrained skyline queries in a large-scale unstructured distributed environment. Lin *et al.* [16] studied the problem of computing constrained skyline over data streams.

III. Multiple-Constrained-Skyline-Query Evaluation Algorithm

Here, we present the concept of the proposed MCSQ (i.e., Multiple-Constrained-Skyline-Query) processing algorithm. MCSQ is based on the BBS algorithm. We use an example to explain how MCSQ works. In the example, a set of 2D data points are indexed using an R-tree (see Fig. 3(b)). The leaf node of the R-tree corresponds to a data point while the directory node (i.e., e_i) indicates a minimum bounding rectangle (MBR) of a node N_i. In Fig. 3(a), we use a gray rectangle to indicate a constrained skyline query. Thus, in the example, there are two constrained skyline queries (i.e., q_1 and q_2).

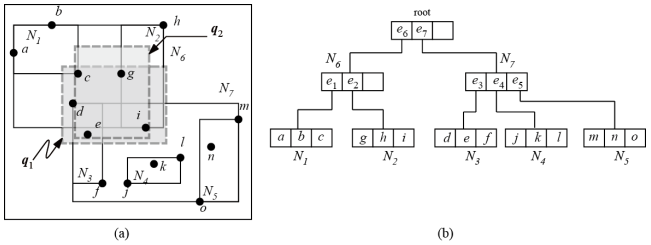

Fig. 3. An R-tree example.

Like BBS, MCSQ uses iterative nearest neighbor search to find skyline points that are inside the constrained region specified by the skyline query. MCSQ recursively traverses the R-tree, performs a nearest neighbor search to find the MBRs or the data points that are not dominated by the current skyline points.

Unlike BBS, we employ *incremental processing paradigm* to process multiple constrained skyline queries. Assume $Q = \{q_1, q_2, ..., q_n\}$ is a set of constrained skyline queries waiting for processing. When processing an R-tree entry (i.e., a MBR or a data point), MCSQ not only determines whether the entry is an answer of q_i, but MCSQ also checks if the entry is relevant to q_j (j=i+1, i+2,…, n). For example, when processing q_1, MCSQ learns that e_2 (i.e., MBR N_2) is dominated by data point e. Since e also resides in the constrained region of q_2, e_2 can be discarded when MCSQ processes q_2. To facilitate the incremental processing, each R-tree entry x is assigned two bit vectors: $r(x)$ and d(x). $r(x)$ records the relationship of each query with x. The ith bit of r(x) is set (i.e., 1) if the constrained region of q_i covers or intersects x. For example, after processing e_2, $r(e_2)$ = <1, 1> because e_2 intersects the constrained regions of q_1 and q_2. The ith bit of $d(x)$ indicates that whether x should be involved in the dominance test when processing q_i. For example, after processing q_1, $d(e_2)$ = <0, 0> which indicates that e_2 can be discard from the dominance test when processing q_2.

MCSQ maintains three data structures, H, S and CacheTbl, when processing a constrained skyline query. H is a heap that keeps track of unexamined directory nodes and data points in ascending L_1 distance to the origin in the data space. S is a buffer that keeps computed skyline points. Initially, H and S are empty. CacheTbl is a hash table. MCSQ stores the accessed R-tree entries in the CacheTbl so that these entries can be used later.

Assume MCSQ processes q_1 first and then deals with q_2. MCSQ starts from the root node of the R-tree and inserts its entries (e_6, e_7) that intersect the constrained region of q_1 in H. Then, the head entry of H is popped out and all its child entries are examined to see if they are *valid* in q_1. By "valid", we mean that (1) the entry intersects with the constrained region of q_1 and (2) the entry is not dominated by the current skyline points contained in S. That is, for each heap entry e in H, MCSQ should perform two tests to determine the validation of e. The two tests are the *intersection test* and the *dominance test*. The intersection test check if e intersects the constrained region while the dominance test determines whether e is dominated by the skyline points in S. In this step, MCSQ pops out e_7 and checks the validation of all its child entries (i.e., e_3, e_4, and e_5). Fig. 4 shows the pseudo code of the validation algorithm.

We take e_3 as an example to illustrate how the validation algorithm works. First, MCSQ inserts e_3 into the CacheTbl. Then, MCSQ performs the intersection test first and then the dominance test. For each query q_i, MCSQ test if e_3 overlaps with the constrained region of q_i. If yes, then the ith bit of $r(e_3)$ is set to 1, otherwise 0. As e_3 intersects the constrained regions of q_1 and q_2, $r(e_3)$ is set to be <1, 1>. Note that if the ith bit of $r(e_3)$ is 0 after the intersection test, then the validation algorithm can be terminated because e_3 does not intersect the constrained region of q_i. In this case, since the first bit of r(e_3) is set, MCSQ performs the dominance test on e_3. For the dominance test, MSCQ compares e_3 against the entry in S to see if e_3 can be dominated. Since none of the entries in S can dominate e_3, MSCQ puts e_3 into H for latter process. The same process can be applied to e_4 and e_5. Since e_4 and e_5 do not intersect the constrained region of q_1, they are not inserted in H.

Algorithm 1: The validation test algorithm.

```
GIVEN: A child entry e of an R-tree entry,
       a set of queries Q that is proceed by MCSQ,
       and the skyline candidate set S of q_i
1  e ← remove an entry from C ;
2  insert e into CacheTbl ;
   /* The intersection test                                   */
3  foreach query q_i ∈ Q do
4      if e intersect the constrained region of q_i then
5          r(e) ← 1 ;
6          d(e) ← 1 ;
7  if r(e) == 0 then
8      continue ;
   /* The dominance test                                      */
9  foreach skyline point sp ∈ S do
10     if e is dominated by sp then
           /* "-" is the set difference operation             */
11         d(d)− r(e) ;
12         break ;
```

Fig. 4. The validation algorithm.

The next popped out entry is e_3. MSCQ extracts e_3's children (i.e., *d*, *e*, and *f*) and performs the validation check on the e_3's referenced child nodes. After the execution of the validation algorithm, node *f* is pruned and nodes *d* and *e* are inserted to H. *e* is the next entry to be extracted. Since *e* is a data point and it belongs to the skyline, *e* is inserted into the list S. Following the same process, *d* is also inserted into S.

The next entry to be popped out is e_6. Two children, e_1 and e_2, are extracted for the validation check. After the intersection test, $r(e_2)$ = <1, 1> because e_2 intersects the constrained regions of q_1 and q_2. However, in the dominance test, MCSQ finds that e_6 is dominated by *e*. Since *e* resides in the constrained regions of q_1 and q_2, the entries inside e_6 (i.e., *h*, *i* and *g*) do not belong to the skyline results of q_1 and q_2. e_2 is pruned and d(e_2) is set to be <0, 0>. MCSQ proceeds in the same manner until the heap becomes empty. Table I shows the detail steps when processing q_1. The R-tree entries that participate in the intersection test and the intersection test when MCSQ accesses a specified entry are recorded in the "dominance test field" and the "intersection test filed", respectively. For example, MCSQ applies the intersection test on e_3, e_4 and e_5 and performs the dominance test on e_3 when e_7 is accessed.

TABLE I: THE DETAIL STEPS WHEN MCSQ PROCESSING Q_1.

Action	H	S	Dominance test	Intersection test
Access root	e_7, e_6	ϕ	e_7, e_6	e_7, e_6
Access e_7	e_3, e_6	ϕ	e_3	e_3, e_4, e_5
Access e_3	*d*, *e*, e_6	ϕ	*d*, *e*	*d*, *e*, *f*
Access *d*	*e*, e_6	*d*	*d*	
Access *e*	e_6	*d*, *e*	*e*	
Access e_6	e_1	*d*, *e*	e_1, e_2	e_1, e_2
Access e_1	*c*	*d*, *e*	*c*	*a*, *b*, *c*
Access *c*		*d*, *e*	*c*	

Up to now, MCSQ behaves like BBS algorithm. However, MCSQ can utilize the information obtained from the

processing of q_1 to reduce the computation cost when processing q_2. We now explain how the goal is achieved. Table shows the detail steps when processing q_2. Let us check the 3th row of Table II. When e_7 is accessed, MCSQ performs the validation test against the children of e_7 (i.e., e_3, e_4, and e_5). Since the second bit of r(e_4) and r(e_5) is 0, MCSQ directly prunes the two entries without doing the intersection test. When accessing e_6 (the 7th row of Table II), since the second bit of $d(e_2)$ is 0, e_2 can be eliminated without performing the dominance test.

TABLE II: THE DETAIL STEPS WHEN MCSQ PROCESSING Q_2.

Action	H	S	Dominance test	Intersection test
Access root	e_7, e_6	ϕ	e_7, e_6	
Access e_7	e_3, e_6	ϕ	e_3	
Access e_3	d, e, e_6	ϕ	d, e	
Access d	e, e_6	d	d	
Access e	e_6	d, e	e	
Access e_6	e_1	d, e	e_1	
Access e_1		d, e		

Fig. 5 shows the pseudo code of the MCSQ algorithm. The major difference between MCSQ and BBS lines in Lines 11-24. If an *R*-tree entry *e* is accessed for the first time, MCSQ checks it for validation (line 12). *e* is discarded if it does not intersect the constrained region of qi (line 14) or if it is dominated by some point in S (line 16). Conversely, if *e* has been accessed before (i.e., *e* can be found in CacheTbl) (line 20), MCSQ first test the *i*th bit of d(*e*) to see if MCSQ has to check *e* for dominance. In our previous example, the second bit of d(e_2) is 0 when MCSQ processes q_2. Therefore e_2 can be eliminated from further examination.

Algorithm 2: The MCSQ algorithm.

```
   GIVEN: An R-tree R,
          a set of queries Q that is proceed by MCSQ,
          and the skyline candidate set S of q_i
1  foreach query q_i ∈ Q do
2    H ← φ;
3    S ← φ;
4    insert the entries of the root R that intersect the constrained region of q_i in H ;
5    while H is not empty do
6      e ← the top entry of H ;
7      if e is dominated by some point in S then
8        discard e;
9      if e is a MBR then
10       foreach child entry e_i ∈ e do
11         if e ∉ CacheTbl then
           /* ref. Algorithm 1                                          */
12         e is checked for validation ;
13         if r(e)[i] == 0 then  /* e does not intersect the constrained region of q_i
           */
14           discard e ;
15         if d(e)[i] == 0 then
16           discard e ;
17         insert e into H ;
18       else
           /* If e has been accessed before q_i is processed            */
19         if d(e)[i] == 0 then
20           discard e ;
21         if e is dominated by some point in S then
22           discard e ;
23         insert e into H ;
24     else
         /* e is a data point                                           */
25       insert e into S ;
```

Fig. 5. The pseudo code of MCSQ algorithm.

IV. EXPERIMENTS

A. Performance Settings

In this section, we conducted several experiments to evaluate the performance of MCSQ and BBS. In BBS, each constrained skyline query *q* is processed independently while in MCSQ we employ incremental processing paradigm to accelerate the processing of *q*. In the experiments, synthetic datasets are generated follow independent distribution with various cardinalities (1000K~5000K). In the experimented datasets, the attribute value of each data point is normalized to [0, 100]. We use *total elapsed time* as the performance metric. It is measured as the duration from the beginning of evaluation to the time that *n* constrained skyline queries are processed. In our experiments, *n* (the number of constrained skyline queries) is varied from 20 to 70. We implement all the algorithms in MS Visual C++ on Windows 7 with Intel Core i5 CPU and 4GB RAM.

1) *Effect of number of queries*

We first study effect of number of queries. Figure 6 depicts the total elapsed time against the number of queries (n) from 20 up to 70 while the data cardinality and dimensionality are fixed at 1000K and 4, respectively. MCSQ shows its superiority over BBS. This means that the incremental processing paradigm can indeed facilitate constrained skyline query processing. The result also reverses that MCSQ performs better with the increasing query cardinality. The reason can be explained as follows. Given a data point *p* and a set of R-tree entries *R* dominating by *p*, *R* can be ignored from dominance test when processing a constrained query *q* if *p* lies in *q*. As n grows, more queries would cover *p* and the computation cost of these queries can be reduced, resulting in shorter elapsed time.

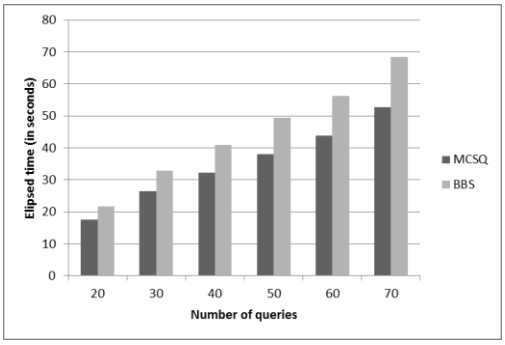

Fig. 6. The effect of number of constrained skyline queries.

2) *Effect of number of data points*

In this experiment, we study the effect of number of data points. Fig. 7 shows the elapsed time of MCSQ and BBS against the data cardinalities varied from 1,000K to 5,000K and *n* and *d* respectively fixed at 40 and 4. The performances of both algorithms deteriorate as the data cardinality increases. This is because the number of skyline points increases as the dataset is increased. The query processing algorithm should perform more dominance tests to retrieve all skyline points, resulting a longer query processing time. We also observe that the performance difference between MCSQ and BBS becomes larger when the data cardinality increases. This is because data point *p* in a larger data set can dominate more R-tree entries. Therefore, the processing cost for the queries that cover *p* can be reduced due to the incremental processing paradigm.

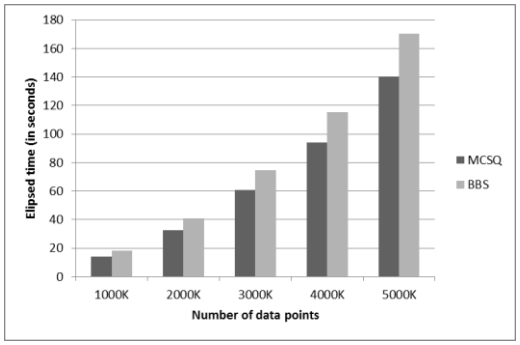

Fig. 7. The effect of the data cardinality.

V. Conclusion

We presented MCSQ that is an efficient query processing for multiple constrained skyline queries. The difference between MCSQ and the existing skyline query processing algorithms is that MCSQ does not process each skyline query independently. MCSQ identifies the R-tree entries that are dominated by a specified data point p when processing a constrained skyline query q_i. Later, these dominated R-tree entries can be discarded from dominance test when MCSQ processes another query that covers *p*. The performance results revealed that our approach can achieve efficient constrained skyline query processing.

References

[1] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proc. ICDE*, 2011, pp. 421-430.
[2] C. Y. Chan, H. V. Jagadish, K. L. Tan, A. K. H. Tung, and Z. Zhang, "Finding k-dominant skylines in high dimensional space," in *Proc. SIGMOD*, 2006, pp. 503-514.
[3] E. Dellis, A. Vlachou, I. Vladimirskiy, B. Seeger, and Y. Theodoridis, "Constrained subspace skyline computation," in *Proc. CIKM*, 2006, pp. 415-424.
[4] Y. Tao, X. Xiao, and J. Pei, "Efficient skyline and top-k retrieval in subspaces," *IEEE TKDE*, vol. 19, no. 8, pp. 1072-1088, 2007.
[5] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM TODS*, vol. 30, no. 1, pp. 41-82, 2005.
[6] K. C. K. Lee, W. C. Lee, B. H. Zheng, H. J. Li, and Y. Tian, "Z-SKY: an efficient skyline query processing framework based on Z-order," *VLDB*, vol. 19, pp.333-362, 2010.
[7] L. J. Chen, B. Cui, and H. Lu, "Constrained skyline query processing against distributed data sites," *IEEE TKDE*, vol. 23, no. 2, pp. 204-271, 2011.
[8] T. K. Sellis, "Multi-query optimization," *TODS*, vol. 13, no. 1, 1988.
[9] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe, "Efficient and extensible algorithms for multi query optimization," in *Proc. ACM SIGMOD*, 2000.
[10] X. P. Xiong, M. F. Mokbel, and W. G. Aref, " SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases," in *Proc. ICDE*, 2005, pp. 643-654.
[11] X. P. Xiong, M. F. Mokbel, and W. G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatiotemporal Databases," in *Proc. ACM SIGMOD*, 2004, pp. 623 - 634.
[12] B. Braunmüller, M. Ester, H. P. Kriegel, and J. Sander, "Multiple similarity queries: A basic DBMS operation for mining in metric databases," *IEEE TKDE*, vol. 13, no. 1, pp.79-95, 2001.
[13] Y. Zhuang, Q. Li, and L. Chen, "Multi-query optimization for distributed similarity query processing," in *Proc. ICDCS*, 2008, pp. 639-646.
[14] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *Proc. ICDE*, 2003, pp. 717-816.
[15] I. Bartolini, P. Ciaccia, and M. Patella, "SaLSa: Computing the skyline without scanning the whole sky," in *Proc. EDBT*, 2006, pp. 405-414.
[16] J. X. Lin and J. J. Wei, "Constrained skyline computing over data streams," in *Proc. IEEE international conference on e-Business Engineering*, 2008, pp. 155-161.

Fang Su received her Ph.D. degree in the Dept. of Computer Science and Information Engineering at the National Cheng-Kung University, Taiwan, in 2010. Currently, she is an assistant professor of the Dept. of Information Management, Fotech, Taiwan. Her research interests include mobile data management, sensor networks, skyline query processing, spatio-temporal databases, and web information retrieval.

Yu-Chi Chung received his Ph.D. degree in the Department of Computer Science and Information Engineering at the NCKU, Taiwan, in 2007. Currently, he is an assistant professor of the Department of Computer Science and Information Engineering, CJCU, Taiwan. His research interests include mobile/wireless/ spatio-temporal data management, sensor networks, skyline query processing, and web information retrieval.

Yuan-Ko Huang received the Ph.D. degree in the Department of Computer Science and Information Engineering at National Cheng-Kung University, Taiwan, in 2009. Currently, he is an assistant professor in the Department of Information Communication, Kao-Yuan University, Taiwan. His research interests include mobile data management, spatio-temporal databases, sensor networks, and social networks.

Chang-Ming Tsai received his Ph. D. degree in Electrical Engineering from Texas A&M University. He is currently an associate professor in the Department of Information Management, Chang Jung Christian University. His research interests include data mining and social networking.