LETTER

An Optimal Algorithm for Searching the Optimal Translation of Query Windows in Quadtree Decomposition

Hao CHEN[†a)], *Member and* **Guangcun LUO**[†], *Nonmember*

SUMMARY One of the efficient methods to build the index of continuous window queries over moving objects is by means of region quadtree index. In this paper, we present an optimal algorithm to search for the optimal position translation of query windows, where the total number of decomposed quadtree blocks for those windows in quadtree representation is minimal. We exploit the branch-and-bound concept to prune the particular paths of recursions in the search space. Evaluation proves that our optimal algorithm reduces search time greatly and the quadtree index based on optimal position translation works efficiently for continuous window queries. To the best of our knowledge, the algorithms and experiments reported in this paper are novel.

key words: *quadtree blocks, query windows, query indexing, position translation*

1. Introduction

Quadtree-based hierarchical data structures have been widely used in many fields such as geographic information systems, spatio-temporal databases, image processing, and so on [1]–[4]. A well-known strategy for window or range query processing is by decomposing a window into a set of maximal quadtree blocks. In the past a few years, some hierarchical decomposition algorithms have been presented to solve this window decomposition problem [5]–[7].

Recently, continuous window or range query over moving objects has attracted researchers' attention greatly [8]–[11] because of various emerging applications requirement, such as navigation and information services, location-dependent advertising, and moving target monitoring, etc. In contrast to traditional snapshot query, which runs once against a snapshot of databases, continuous window query over moving objects must be reevaluated continuously as new data of object location continues to arrive in a data stream, until it is canceled. Since a brute-force approach which evaluates each of the window queries for each incoming data tuple of object location is inefficient, it is natural to adopt main-memory-based solutions of indexing to speed up the processing of continuous window queries. Kalashnikov [8] proposed a cell-based query indexing scheme which was shown to perform better than the R-tree-based query index. Mokbel and Aref [11] presented a Scalable On-Line Execution (SOLE) algorithm for the evaluation of concurrent continuous spatio-temporal queries over data streams. SOLE maintains a simple grid structure that divides the space into equal non-overlapped rectangular cells as an in-memory shared buffer pool among all continuous queries. However, the simple grid structure in SOLE is not optimal for large scale queries. To optimize query processing further, some kinds of virtual constructs (VCs) for building query indexes were proposed, including covering tiles [9], virtual construct rectangles (VCRs) [16] and containment-encoded squares (CES) [10]. The CES-based query index uses quadtree blocks to decompose query windows and was shown to outperform other query indexing approaches. In CES method, the query index is implemented with a pointer array and dynamically maintained query ID lists, one for each quadtree block. Each window query is decomposed into quadtree blocks and the query ID is inserted into the ID lists associated with these decomposed quadtree blocks. Search operations are conducted indirectly via quadtree blocks. However, the CES method have performance limitation. The number of decomposed quadtree blocks for query windows could be cut down by doing optimal position translation on query windows, since the decomposed quadtree blocks for a rectangle query window is sensitive to the relative position of the window in the coordinate grid. A shift of the window may change the number of decomposed blocks and the cost of indexing greatly [13], [14]. For example, a square window of $2^d \times 2^d$ may be decomposed into just only one quadtree block or as many as $3(2^{d+1}-d)-5$ blocks [15] by placing the window at different positions in a global grid of size $2^N \times 2^N$ ($d < N$). Therefore, it's desired to locate an optimal position translation for window queries so that the total number of quadtree blocks decomposed by quadtree-based decomposition approach can be minimized, after an unified position translation. And then, the total index storage cost can be decreased.

In this paper, we study position translation of query windows for quadtree indexing and make the following contributions: 1) We design a basic search algorithm to find out the optimal translation of window queries where the number of decomposed quadtree blocks is minimal, exploiting the conception of quadtree decomposition and strip-splitting-based decomposing method [7]; 2) To speed up the search process further, we present an optimal search algorithm in which two new pruning methods are used to cut down the searching space further. Experiment evaluation shows that our optimal search algorithm works efficiently, and the quadtree indexing based on optimal position trans-

Manuscript received January 28, 2011.
Manuscript revised May 5, 2011.

[†]The authors are with the School of Computer Science, University of Electronic Science and Technology of China, ChengDu, China.
a) E-mail: chenhao@uestc.edu.cn

DOI: 10.1587/transinf.E94.D.2043

lation outperforms CES-based Indexing both in storage cost and query evaluation time.

The remainder of this paper is organized as follows: Section 2 describes the background and problem. Section 3 presents our algorithms for searching for the optimal translation of query windows. Section 4 reports the results and comparisons of our experimental study and Sect. 5 concludes this paper.

2. Background and Problem Description

We assume that there is a global grid area of size $2^N \times 2^N$, where moving objects are tracked. A continuous window query over moving objects in the global area is specified as a rectangle window along the grid lines. We denote a rectangle query window with width w, height h and bottom-left corner at (x, y) by $R(x, y, w, h)$. Let $d = min(\lfloor logw \rfloor, \lfloor logh \rfloor)$ and we call d the *dimension* of R. We use RS $(= \{R_1, R_2, \ldots, R_M\})$ to denote a set of registered query windows which could be intersected or disjointed. To speed up the processing of multiple continuous window queries, a main-memory-based region quadtree is used to index query windows RS and each node of the quadtree maintains a query ID list. In this index, each window query is decomposed into quadtree blocks and the query ID is inserted into query ID lists of corresponding quadtree nodes associated with these blocks. The processing of continuous window queries is data-driven and the query answer is computed incrementally on arrival of new location updates of moving objects. For any location update of moving object, a search of quadtree index is conducted to first identify the quadtree blocks that cover the object location. Then we can obtain the related queries from the associated query ID lists, and output the object ID or location data to the resulting queues associated with these identified queries. Since the quadtree decomposition of window queries is shift variant, we optimize the quadtree indexing of RS by translating all these query windows at locations and searching for an unified optimal position translation, where the total number of decomposed quadtree blocks is minimal. With position translation of query windows, it is expected that the total index storage cost is reduced and the total query evaluation time becomes lower.

3. Searching for the Optimal Position Translation of Query Windows in Quadtree Presentation

3.1 Problem Analysis and Basic Algorithm

If we translate RS in all possible locations of the global grid to look for the optimal translation with minimal quadtree block count, the algorithm must search exhaustively through the whole space. The worst time complexity of this naive algorithm is $O(4^N)$. Here, we first propose a basic search algorithm based on the following definition and lemmas.

Definition 1 (Cyclic Translation of Query Window). For a query window $R(x, y, w, h)$ in global area of $2^N \times 2^N$, we translate (move) R with magnitude X_{tm} and Y_{tm} in eastern and northern direction respectively. A window translation is called a cyclic one if the final position of an unit cell located originally at (x', y') in this window, after translation, can be derived from the following function:

$CT(x', y', X_{tm}, Y_{tm}) = ((x' + X_{tm}) \ mod \ 2^N, \ (y' + Y_{tm}) \ mod \ 2^N)$.

With the cyclic translation of query windows, we can construct a region quadtree of size $2^N \times 2^N$ to decompose all translated query windows without the need of expanding the quadtree to the size of $2^{N+1} \times 2^{N+1}$. From the property of region quadtree and cyclic translation, we present the following result without proof.

Lemma 1. Translating a query window embedded in $2^N \times 2^N$ grid cyclically by 2^d grid cells in east and north directions, where $d < N$, does not change the number of decomposed quadtree blocks of size less than $2^d \times 2^d$.

From lemma 1, we can derive the minimal searching space for finding the optimal position of a query window in quadtree decomposition.

Lemma 2. An optimal position for decomposing R into minimal quadtree blocks could be gotten by translating R(x, y, w, h) by less than 2^d units to the east and 2^d units to the north, where $d = min(\lfloor logw \rfloor, \lfloor logh \rfloor)$.

Proof. Since the largest quad block possibly decomposed from R is of size $2^d \times 2^d$,we can have the same number of decomposed blocks when we translate R with magnitudes of 2^{d+1}. It's easy to see that all blocks of size $2^d \times 2^d$ in R are always arranged in a line. Therefore, translating R with magnitudes of 2^d won't change the number of blocks of size $2^d \times 2^d$, the same as those blocks no more than $2^{d-1} \times 2^{d-1}$. We thus have the proof. □

By Lemma 2, we know that only translations by at most $2^d - 1$ grid elements to the east and to the north need be considered when we search for the optimal location of quadtree decomposition for a query window R. Hence, the sample solution space of optimal translation for a set of query windows RS is of size $2^D \times 2^D$, where $D = max(d_1, d_2, \ldots, d_M)$, d_m is the dimension of each window R_m in RS. Since any translation amount t $(< 2^D)$ can be expressed as the sum of some numbers from the set $1, 2, 4, \ldots, 2^{D-1}$ and the translations with larger magnitudes(2^i) would not affect the number of decomposed quadtree blocks with smaller magnitudes(2^{i-1}), we can derive a basic algorithm of searching for optimal position translation, based on the conception of strip-splitting-based decomposing method [7]. The basic idea of the algorithm is to consider the combination of moving rightward or upward by $1, 2, 4, \ldots, 2^{D-1}$ elements successively during the search process, recursively translating all windows of RS in four directions (no movement, north, east, northeast) by 2^i elements and then strip-splitting each of them using strip-splitting-based method, while evaluating the sum of stripped quadtree blocks. After each translation, each window of RS is stripped off some slices. Then, we recursively call the algorithm on remaining new query windows of each translation configuration. Recursion halts when we strip-split original windows $D - 1$ times or when

further translation will not produce any savings.

The pseudocode of the basic search algorithm is described in Algorithm 1. Here, $M(x, y) = x \bmod y$. The algorithm is constructed by a sequence of translations to RS, with magnitudes from 2^0 to 2^{D-1} in four different directions: no movement, north, east, and northeast (lines 6-7 of Algorithm 1). At each translation, we strip-split and compact each query window by a process which is a modification of strip-splitting-based algorithm [9] (lines 8 of Algorithm 1). In the strip-splitting process (lines 26-46), each window of RS is stripped off some slices of size $c \times 2^i \times 2^i$ where c is a positive integer. After that, we shrink these remaining windows (line 41) since the same number of blocks is maintained while making the granularity coarser [17]. Then, we recursively call the algorithm on new query windows (line 12). Note that translating these windows by 1 element at level i of recursion corresponds to a translation of unshrunk windows by 2^i elements. Actually, for each translation, instead of performing translations at all next levels, we calculate the total number of quadtree blocks stripped so far, which is computed from the bottom level of strip-splitting to the level being processed, and then compare it with the current least block counter derived so far (line 10). If the total number of blocks obtained so far is larger, the recursions of further levels are cut down. The algorithm breaks off when all recursions are executed or cut down. At last, the translation magnitudes for the optimal location in two directions, i.e. X_{tm} and Y_{tm}, can be acquired by the above process in a sequence of binary digits (line 9, 19). The worst time complexity of this basic algorithm is $O(4^D)$.

Algorithm 1 Basic Search Algorithm.

Require:
RS;
Ensure:
X_{tm}, Y_{tm} (translation magnitudes);
1: $d = 0$; $BN[D+1] = \{0, \ldots, 0\}$; $X_m = Y_m = 0$;
2: $least = \sum_{m=1}^{M} w_m \times h_m$;$X_{tm} = Y_{tm} = 0$;
3: $OptimalSearch(RS, X_m, Y_m, least, BN, d)$;
4:
5: $function : OptimalSearch(RS, X_m, Y_m, least, BN, d)$
6: **for** $(k = 0; k \leq 3; k++)$ **do**
7: $i = \lfloor k/2 \rfloor$; $j = M(k, 2)$; $RS' \leftarrow RS$; $BN' \leftarrow BN$;
8: $BN'[d] = RangesSplitShrink(RS', k)$;
9: $X'_m = X_m + i \times 2^d$; $Y'_m = Y_m + j \times 2^d$;
10: **if** $\sum_{i=0}^{d} BN'[i] \leq least$ **then**
11: **if** $(d < D-1)$ **then**
12: $OptimalSearch(RS', X'_m, Y'_m, least, BN', d+1)$;
13: **else**
14: **if** $RS' \neq \phi$ **then**
15: $BN'[D] = RangesSplitShrink(RS', 0)$;
16: **end if**
17: $BN_{total} = \sum_{i=0}^{D} BN'[i]$;
18: **if** $BN_{total} < least$ **then**
19: $least = BN_{total}$; $X_{tm} = X'_m$; $Y_{tm} = Y'_m$;
20: **end if**
21: **end if**
22: **end if**
23: **end for**
24:
25: $function : RangesSplitShrink(RS, k)$
26: $i = \lfloor k/2 \rfloor$; $j = M(k, 2)$; $C = 0$;
27: **for** each $R_m(x_m, y_m, w_m, h_m)$ in RS **do**
28: $x_m = x_m + i$; $y_m = y_m + j$;
29: **if** $M(x_m, 2) \neq 0$ **then**
30: $C = C + h_m$; $x_m = x_m + 1$; $w_m = w_m - 1$;
31: **end if**
32: **if** $M(y_m, 2) \neq 0$ **then**
33: $C = C + w_m$; $y_m = y_m + 1$; $h_m = h_m - 1$;
34: **end if**
35: **if** $M(x_m + w_m, 2) \neq 0$ **then**
36: $C = C + h_m$; $w_m = w_m - 1$;
37: **end if**
38: **if** $M(y_m + h_m, 2) \neq 0$ **then**
39: $C = C + w_m$; $h_m = h_m - 1$;
40: **end if**
41: $x_m = x_m/2$; $y_m = y_m/2$; $w_m = w_m/2$; $h_m = h_m/2$;
42: **if** $w_m = 0$ *or* $h_m = 0$ **then**
43: delete R_m from RS;
44: **end if**
45: **end for**
46: return C;

3.2 Optimal Algorithm

Algorithm 1 is quite efficient compared with the brute force approach in the sense that it makes use of the partial result computed so far to cut down the number of recursions. To enhance search performance further, we propose an optimal algorithm (Algorithm 2) adopting two more pruning methods.

The first pruning mechanism is to use a heuristic that can lead us to some branch of the configuration search tree with quadtree block count close to that of the optimal solution. It's obvious that, the sooner a particular path of recursion in Algorithm 1 is pruned, the greater is the saving of search time. In order to obtain a candidate close to the optimal solution as earlier as possible, We use an iterative process to quickly derive the "initial candidate" (lines 2, 47-57 of Algorithm 2). For each iteration, four translations to RS in different directions with magnitudes from 2^0 to 2^{D-1} are performed, each window is strip-splitted and shrunk (line 50). Then, the translated RS with the smallest number of striped quad blocks is selected for the next iteration (line 52). After at most D iterations, the "initial candidate" is identified to establish the bound.

To introduce the second pruning mechanism, we present following lemmas.

Lemma 3 Given a query window R of size $w \times h$, $d = \min(\lfloor logw \rfloor, \lfloor logh \rfloor)$. If we place the low left corner of R at $(i \times 2^d,\ j \times 2^d)$, where i and j are positive integers, the minimal number of decomposed quad blocks can be gotten.

Proof. The number of decomposed blocks of a window is obviously minimal if more cells are merged into larger quad blocks. It's easy to see that, if we shift the lower left corner of R to coordinates (0, 0) of global grid, then at each level of strip-splitting of the window, the least number of blocks (i.e. at most one column and one row strip) are stripped off, and more cells are left to merge into more larger blocks. From lemma 1 and 2, it's clear that the number of decomposed blocks is always minimal if we move the lower

left corner of R to $(i \times 2^d, j \times 2^d)$. We thus have the proof. □

From lemma 3, for any query window of size $w \times h$, the least number of decomposed blocks can be gotten by decomposing $R(0, 0, w, h)$. Lemma 3 can be extended to obtain the following:

Lemma 4 For a query window $R(x, y, w, h)$, let $B_1, B_2, \ldots, B_i$ be the numbers of decomposed quad blocks from the 1th to ith strip-splitting, where $i \leq min(\lfloor logw \rfloor, \lfloor logh \rfloor)$. The possible minimal total count of decomposed quad blocks, $B_{pmin}(R)$, is the sum of B_is and the number of quad blocks of a new window R'_i which is derived from strip-splitting i levels off original window and then shifting it's lower left corner to the coordinates $(0, 0)$*:*

$$B_{pmin}(R) = B_1 + B_2 + \ldots + B_i + B(R'_i).$$

Lemma 4 can be used for pruning the search process. For each translation configuration, we can calculate the current possible minimal number of quad blocks (line 9, 26-44). If the total possible minimal number of quadtree blocks is greater than current least counter (line 10), the search paths rooted by the translation configuration are cut down.

4. Experimental Results

In this section, we evaluate the performance of search algorithms for optimal position translation and compare translation-based quadtree indexing with CES-based indexing mechanism. In the simulations, the global area is defined by a $2^{10} \times 2^{10}$ grid. A total number of $|Q|$ continual window queries are registered. A total of 20000 moving objects are tracked and object locations are updated 10 times each minute. We conducted our simulations over a machine equipped with 2.4-GHz CPU, 512-Mbyte RAM, and Windows XP.

4.1 Performance of Optimal Search Algorithm

Now, we are in the position of comparing the performance of the optimal algorithm with the basic algorithm for finding optimal translation of query windows. Firstly, the width and height of query windows were randomly chosen between 2^3 and 2^9, and we varied the number of queries $|Q|$ from 100 to 1000. Figure 1 (a) shows the search time of two algorithms with different number of query windows. It is clear that the optimal algorithm outperforms the basic algorithm greatly. Secondly, $|Q| = 1000$. We varied the dimension of all query windows from 3 to 9. Figure 1 (b) shows the search time of two algorithms with different size of queries. The search time is decreased greatly when the dimension of query windows is smaller.

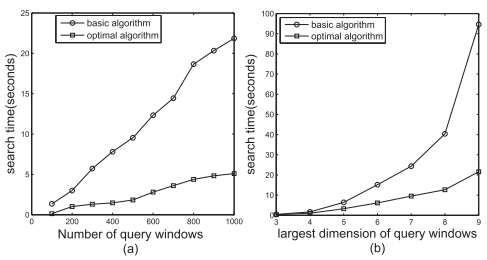

Fig. 1 The performance evaluation of search algorithms.

Algorithm 2 Optimal Search Algorithm.

Require:
RS;
Ensure:
X_{tm}, Y_{tm} (translation magnitudes);
1: $d = 0$; $BN[D+1] = \{0, \ldots, 0\}$; $X_m = Y_m = 0$; $sum = 0$; $X_{tm} = Y_{tm} = 0$;
2: $least = FindInitialCandidate(RS)$;
3: $OptimalSearch(RS, X_m, Y_m, least, BN, d)$;
4:
5: $function : OptimalSearch(RS, X_m, Y_m, least, BN, d)$
6: **for** $(k = 0; k \leq 3; k++)$ **do**
7: $i = \lfloor k/2 \rfloor$; $j = M(k, 2)$; $RS' \leftarrow RS$; $BN' \leftarrow BN$;
8: $BN'[d] = RangesSplitShrink(RS', k)$;
9: $X'_m = X_m + i \times 2^d$; $Y'_m = Y_m + j \times 2^d$; $sum = ComputeMinSum(RS')$;
10: **if** $\sum_{i=0}^{d} BN'[i] + sum \leq least$ **then**
11: **if** $(d < D - 1)$ **then**
12: $OptimalSearch(RS', X'_m, Y'_m, least, BN', d + 1)$;
13: **else**
14: **if** $RS' \neq \phi$ **then**
15: $BN'[D] = RangesSplitShrink(RS', 0)$;
16: **end if**
17: $BN_{total} = \sum_{i=0}^{D} BN'[i]$;
18: **if** $BN_{total} < least$ **then**
19: $least = BN_{total}$; $X_{tm} = X'_m$; $Y_{tm} = Y'_m$;
20: **end if**
21: **end if**
22: **end if**
23: **end for**
24:
25: $function : ComputeMinSum(RS)$
26: **for** each $R_m(x_m, y_m, w_m, h_m)$ in RS **do**
27: $x_m = 0$; $y_m = 0$;
28: **end for**
29: $sum = 0$;
30: **while** $RS \neq \phi$ **do**
31: **for** each $R_m = (x_m, y_m, w_m, h_m)$ in RS **do**
32: **if** $M(w_m, 2) \neq 0$ **then**
33: $w_m = w_m - 1$; $sum = sum + h_m$;
34: **end if**
35: **if** $M(h_m, 2) \neq 0$ **then**
36: $h_m = h_m - 1$; $sum = sum + w_m$;
37: **end if**
38: $w_m = w_m/2$; $h_m = h_m/2$;
39: **if** $w_m = 0$ *or* $h_m = 0$ **then**
40: delete R_m from RS;
41: **end if**
42: **end for**
43: **end while**
44: return sum;
45:
46: $function : FindInitialCandidate(RS)$
47: $C_k = 0, k = 0, 1, 2, 3$
48: **for** $(d = 0; d < D; d++)$ **do**
49: **for** $(k = 0; k \leq 3; k++)$ **do**
50: $RS_k \leftarrow RS$; $C_k = C_k + RangesSplittingShrink(RS_k, k)$;
51: **end for**
52: $K = argmin_k(C_k)$; $least = least + C_K$; $RS \leftarrow RS_K$;
53: **end for**
54: **if** $RS \neq \phi$ **then**
55: $least = least + RangesSplitShrink(RS, 0)$;
56: **end if**
57: return $least$;

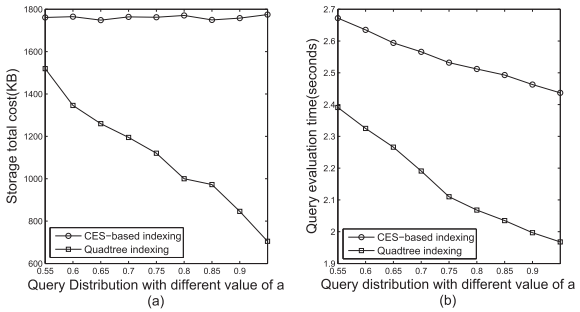

Fig. 2 (a) storage cost over different query distribution; (b) query evaluation time over different query distribution.

4.2 Storage Cost and Query Evaluation Time

Now we shall report the comparison results as to storage cost and query evaluation time between translation-based quadtree indexing against CES-based indexing approach. Here, $|Q| = 1000$. The width and height of query windows were randomly chosen between 2^3 and 2^9. The bottom-left corners of window queries were distributed according to an $\alpha - \beta$ rules as in [10]: α fraction of the bottom-left corners of query windows are located within β fraction of the monitoring area, where $\beta = 1 - \alpha$. We vary α from 0.55 to 0.95 and compare two indexing methods under different query distribution. Figure 2 (a) shows the total storage cost of both indexing methods. It is clear that our index outperforms CES-based index greatly, as more significant as the query positions become more skewed. Figure 2 (b) shows the impact of query distribution on the query evaluation time. Under all cases, our index outperforms CES-based index, as more significant as the query positions become more skewed. It is because the CES-based indexing is implemented with pointer arrays and constant times of accessing to the index are needed whenever a location update of moving object is received for query processing. For our approach, the query evaluation efficiency is higher when objects move in quadtree block regions not covered by queries, since no tree nodes corresponding to these regions are constructed in the index.

5. Conclusions

We propose an optimal algorithm of searching for the optimal position translation of query windows with the minimal total number of decomposed quadtree blocks. To reduce the searching time, we exploit the branch-and-bound concept to prune the unnecessary searching paths. A almost-optimal solution is identified at first to establish the bound. Then another pruning technique of computing the possible minimal number of quadtree blocks is used to cut the particular paths of recursions in the search space. Experiment results show that the optimal search algorithm works efficiently and outperforms another basic algorithm. Moreover, the optimal-translation-based quadtree indexing approach outperforms CES-based indexing in both storage cost and query evaluation time.

References

[1] E. Nardelli and G. Proietti, "Efficient secondary memory processing of window queries on spatial data," Inf. Sci., vol.84, pp.67–83, 1995.
[2] E. Nardelli and G. Proietti, "Time and space efficient secondary memory representation of quadtrees," Information Systems, vol.22, pp.25–37, 1997.
[3] S.-K. Chen, "An exact closed-form formula for d-dimensional quadtree decomposition of arbitrary hyperrectangles," IEEE Trans. Knowl. Data Eng., vol.18, no.6, pp.784–798, June 2006.
[4] J.A. Orenstein and F.A. Manola, "Probe spatial data modeling and query processing in an image database application," IEEE Trans. Softw. Eng., vol.14, no.5, pp.611–629, 1988.
[5] W.G. Aref and H. Samet, "Decomposing a window into maximal quadtree blocks," Acta Informatica, vol.30, pp.425–439, 1993.
[6] G. Proietti, "An optimal algorithm for decomposing a window into maximal quadtree blocks," Acta Informatica, vol.36, no.4, pp.257–266, 1999.
[7] Y.-H. Tsai, K.-L. Chung, and W.-Y. Chen, "A strip-splitting-based optimal algorithm for decomposing a query window into maximal quadtree blocks," IEEE Trans. Knowl. Data Eng., vol.16, no.4, pp.519–523, April 2004.
[8] D.V. Kalashnikov, S. Prabhakar, W.G. Aref, and S.E. Hambrusch, "Efficient evaluation of continuous range queries on moving objects," Proc. Int'l Conf. Database and Expert Systems Applications, pp.731–740, 2002.
[9] K.-L. Wu, S.-K. Chen, and P.S. Yu, "Efficient processing of continual range queries for location-aware mobile services," Information Systems Frontiers, vol.7, nos.4-5, pp.435–448, Dec. 2005.
[10] K.-L. Wu, S.-K. Chen, and P.S. Yu, "Incremental processing of continual range queries over moving objects," IEEE Trans. Knowl. Data Eng., vol.8, no.11, pp.1560–1575, 2006.
[11] M. Mokbel and W. Aref, "SOLE: Scalable on-line execution of continuous queries on spatio-temporal data streams," In VLDB J., pp.971–995, 2008.
[12] K.-L. Wu, S.-K. Chen, and P.S. Yu, "Interval query indexing for efficient stream processing," Proc. ACM 13th Conf. Information and Knowledge Management, pp.88–97, 2004.
[13] M. Li, W. Grosky, and R. Jain, "Normalized quadtrees with respect to translations," Computer Graphics and Image Processing, vol.20, pp.72–81, 1982.
[14] P.-M. Chen, "A quadtree normalization scheme based on cyclic translations," Pattern Recogn., vol.30, no.12, pp.2053–2064, 1997.
[15] C.R. Dyer, "The space efficiency of quadtrees," Computer Graphics and Image Processing, vol.19, no.4, pp.335–348, Aug. 1982.
[16] K.-L. Wu, S.-K. Chen, and P.S. Yu, "Processing continual range queries over moving objects using VCR-based query indexes," Proc. IEEE Int'l Conf. Mobile and Ubiquitous Systems: Networking and Services, Aug. 2004.
[17] C. Faloutsos, H.V. Jagadish, and Y. Manolopoulos, "Analysis of n-dimensional quadtree decomposition of arbitrary rectangles," IEEE Trans. Knowl. Data Eng., vol.9, no.3, pp.373–383, May/June 1997.