Comput Syst Sci & Eng (2013) 2: 101–115

International Journal of
Computer Systems
Science & Engineering

One-dimensional spatial join processing using a DOT-based index structure

Jung-Im Won[1]*, **Hyun Back**[2], **Jee-Hee Yoon**[2], **Sanghyun Park**[3], **Sang-Wook Kim**[4]

[1] *Research Center of Information and Electronic Engineering, Hallym University, Korea, Hallymdaehak-gil, Chuncheon, Gangwon-do 200-702, Korea*
E-mail: jiwon@hallym.ac.kr
[2] *Division of Information Engineering and Telecommunications, Hallym University, Korea*
E-mail: {backhyun@sysgate.co.kr, jhyoon@hallym.ac.kr}
[3] *Department of Computer Science, Yonsei University, Korea. E-mail:sanghyun@cs.yonsei.ac.kr*
[4] *College of Information and Communications, Hanyang University, Korea 17 Haengdang-dong, Seongdong-gu, Seoul 133-791, Korea*
E-mail: wook@hanyang.ac.kr

Spatial join is an operation that finds a set of object pairs with a given spatial relationship from a spatial database. It is very costly, and thus requires an efficient algorithm for its execution that fully exploits the features of underlying spatial indexes. In this paper, we propose a novel one-dimensional spatial join algorithm based on DOT indexing. The proposed algorithm reduces the cost of disk accesses by deciding the access order of pages containing spatial objects to minimize the number of buffer replacements. It also minimizes the CPU cost by using a *quarter division technique*, which divides a query region into a set of subregions that contain consecutive space-filling curves as long as possible. Our algorithm is very easy to integrate with an existing DBMS because it uses B^+-trees as a base structure for DOT indexing. We verify the effectiveness of the proposed algorithm via extensive experiments using data sets with various sizes and distributions. The results show that the proposed join algorithm performs up to 3 times better than the previous R^*-tree-based join algorithm.

Keywords: Spatial database, spatial index, DOT index, spatial join, space-filling curve

1. INTRODUCTION

Spatial objects are objects that have locations and sizes within space [18, 47]. *Spatial database systems* provide for the storage and management of a large number of those spatial objects and support applications such as GIS, VLSI, and CAD/CAM [18, 46]. A variety of spatial queries such as *spatial point queries, spatial range queries*, and *spatial join queries* are frequently used in spatial database systems [8, 23, 28, 32].

For rapid processing of spatial queries, the selection of an efficient indexing method is crucial. An excellent survey on spatial indexing methods is found in [3]. They are classified into three categories as follows.

The first is a class of methods that divide the original space into a set of subspaces and maintain the relationship between those subspaces and their objects. R-trees [17], R^*-trees [2, 33, 45], R^+-trees [41], and Cell-trees [15] are typical examples. We call this class of methods *original space indexing methods (OS-IM)* in this paper.

The second is a class of methods that transform *spatial objects* in original D-dimensional space into *point objects* in 2D-dimensional space and then manage those transformed objects by using multidimensional point indexing methods such as grid files [30], KDB-trees [38], and LSD-trees [19]. For transformation, *corner transformation* and *center transformation* are widely used [34, 40]. We call them *high-dimensional space transformation indexing methods (HDST-IM)*.

*Corresponding author and address: E-mail: jiwon@hallym.ac.kr, Tel: +82-33-248-2382, Fax: +82-33-242-2524

The third is a class of methods that transform objects in original D-dimensional space into objects in 1- dimensional space and then maintain them using the B-tree [6], a traditional 1-dimensional indexing method. Typical examples are proposed in [12, 13, 22, 27, 31, 32]. We call them *low-dimensional space transformation indexing methods (LDST-IM)*.

Among the three kinds of methods, OS-IM has been used most widely. Recently, the methods have been extended for indexing a large number of *moving objects*. 3DR-trees [14, 44], HR-trees [44], STR-trees [36], TB-trees [37], and MV3R-trees [43] are typical examples.

Spatial join is an operation that is performed with two sets of spatial objects. It searches for pairs of spatial objects satisfying a spatial condition such as overlap or containment. Spatial join normally proceeds in two steps: a *filtering step* and a *refinement step* [4]. In the *filtering step*, it is performed with MBRs (Minimum Bounding Rectangles), simplified versions of spatial objects, and it finds candidate pairs of objects, only a part of which can be contained in the final result set. In the *refinement step*, it produces the final result set from those candidate pairs by geometrical computations. Spatial join accesses all the spatial objects repeatedly, and so it requires a fairly high cost for disk accesses and CPU processing [4, 16]. Therefore, spatial join algorithms should be devised with careful consideration of the characteristics of underlying spatial index structures [42].

Based on the three classes of spatial indexing methods mentioned above, several spatial join algorithms have been proposed. References [4, 5, 9, 20] proposed spatial join algorithms based on the R-tree, the most widely used OS-IM. Reference [42] proposed an algorithm that uses HDST-IM. Reference [26] proposed an algorithm that employs the R-tree but performs joins referring to the transformed space. References [10, 31] proposed join algorithms that use LDST-IM. Also, references [1, 35] proposed algorithms for effective processing of spatial joins with limited main memory in situations where spatial indexes do not exist.

Double transformation (DOT) [11, 12], a form of LDST-IM, transforms every minimum bounding rectangle (MBR) in D-dimensional original space into a value in 1-dimensional space by using a *space-filling curve*, and then stores all those values using the well-known B^+-tree [21, 25, 48, 49]. Because the B^+-tree is widely adopted as a basic index structure in most DBMSs, DOT-based query processing algorithms can be easily integrated into existing DBMSs. Also, they solve a problem that occurs in other LDST-IMs [31, 32] where a spatial object in original space is represented as multiple objects in transformed space. In reference [12], DOT-based algorithms for processing spatial point queries and spatial range queries were proposed. To the best of our knowledge, however, there are no DOT-based algorithms for processing spatial joins.

In this paper, we propose a DOT-based join algorithm for one-dimensional spatial objects and show its superiority via performance evaluation. With DOT, spatial objects are clustered in a disk according to their spatial proximity. In processing spatial joins, we determine the access order of the pages containing spatial objects so that we can minimize the amount of buffer replacement.

However, it should be noted that the DOT-based spatial join algorithm requires many operations to transform a query region in original space into multiple values in transformed space. This causes performance degradation of the entire join processing. The proposed algorithm employs the *quarter division technique*, which was devised by analyzing the regularity of the space-filling curve, thereby reducing the number of space transformation operations. To show the superiority of the proposed algorithm, we conduct performance comparisons with the R-tree-based spatial join algorithm, which is the one most widely-used, with data sets of various distributions and sizes. The results reveal that the proposed algorithm achieves a performance up to three times better than the existing one in spatial join processing.

The organization of the paper is as follows. Section 2 briefly reviews the DOT indexing method. Section 3 presents the method for range query processing using DOT and discusses a strategy for optimization by transforming query regions using the quarter division technique. Section 4 proposes an efficient spatial join algorithm that uses DOT indexing. Section 5 presents the experimental results for verifying the performance of the proposed algorithm. Finally, Section 6 summarizes our contributions and suggests further research possibilities.

2. DOT INDEXING

In this section, we describe the DOT indexing method [11, 12] briefly.

DOT transforms an MBR of a spatial object in original space into a point in one-dimensional space by using the high dimensional and low dimensional transformations together. The transformation is performed in two steps: (1) An MBR in k-dimensional space is transformed into a point in $2k$-dimensional space. We call it the *first transformation*. Coner transformation or center transformation can be used for this purpose. (2) A point in $2k$-dimensional space thus obtained is also transformed into a point in one-dimensional space. We call it the *second transformation* and the value in one-dimensional space is the *X value*. The space-filling curve, which is used as the second transformation, should preserve the proximity in the original space [7, 24, 29]. The *Hilbert curve, Peano curve, tri-Hilbert curve*, and *tri-Peano curve* are typically used for this purpose.

Here, the k-dimensional space where spatial objects originally exist, the $2k$-dimensional space after the first transformation, and the one-dimensional space after the second transformation are defined as the *original space*, the *intermediate space*, and the *final transformed space* [12], respectively.

Figure 1 shows an example of a transformation process using DOT. In this example, we assume that we use the *tri-Hilbert curve* [11] as a space-filling curve for the second transformation. Figure 1(a) shows two spatial objects A and B in one-dimensional original space. Figure 1(b) shows objects A and B mapped into the intermediate space by the first transformation. The start and end points of every MBR in one-dimensional original space are represented as values of the X_s and X_e axes corresponding to a point in two-dimensional intermediate space. So, objects A and B are mapped onto two points, (1, 3) and (5, 6), in intermediate space, respectively. Since the end point is always larger than or equal to the start point as shown in Figure 1(b), every object can be mapped only into a right-angled triangle-shaped region, which is located above a diagonal line in intermediate space. Figure 1(c) shows objects A and B mapped into the final one-dimensional transformed space by the second

transformation. If we apply the tri-Hilbert curve [11] for the two-dimensional intermediate space, A and B are mapped into 6 and 32, respectively.

DOT-based indexing maintains one-dimensional points, which are transformed with DOT by using the B^+-tree structure. Consequently, it is easy to integrate itself into an existing DBMS. It also solves the problem of the method in [32] in which one object is represented as multiple points in transformed space.

3. PROPOSED RANGE QUERY PROCESSING ALGORITHM BASED ON DOT INDEX

In this section, we propose an efficient range query processing algorithm based on a DOT index. Section 3.1 introduces a naive DOT-based algorithm for processing range queries and points out some of its drawbacks. Section 3.2 presents a quarter division technique proposed for reducing the cost of space transformation operations, describes our range query processing algorithm based on the quarter division technique, and analyzes the complexity of our algorithm. Finally, Section 3.3 describes a tri-quarter division technique devised for achieving better performance.

3.1 Naive algorithm

A range query is an operation that finds all the spatial objects overlapping a given query range. Let us continue the example in Figure 1 to illustrate a naive DOT-based algorithm for processing range queries. Figure 1(a) shows a query q in one-dimensional original space. In the first query processing step, query q is mapped into a region in intermediate space via the first transformation. That is, $q = (q_s, q_e)$ is transformed into a region that satisfies both $X_s \leq q_e$ and $X_e \geq q_s$ in two-dimensional space. The shaded area in Figure 1(b) shows the query region. Note that, in intermediate space, the region below the diagonal line is not in use.

Next, the query region is mapped into a set of line segments (e.g., *Segment1* and *Segment2* in Figure 1(c)) in one-dimensional final space via the second transformation. Last, the DOT index with a B-tree structure is traversed to find the spatial objects whose search keys are contained in the set of line segments. In our example, the query region is transformed into a set of two line segments, {[3...26], [28...30]}, and spatial object A whose search key is 6 is found after the traversal of the DOT index.

Algorithm **Naive-Range-Query** shown in [Algorithm 1] is a naive range query processing algorithm. Let us examine the algorithm in detail. It transforms the query range into a set of line segments in onedimensional space (line 1), and then, using the DOT index, it retrieves the spatial objects overlapping the query region (line 2). The actual query transformation process is executed in function **QueryRangeToLineSegments()** shown in [Algorithm 2]. It transforms the query region in intermediate space into the corresponding X values (lines 2-4). Here, function **SpaceFillingCurves()** uses such a space-filling curve as a *Hilbert curve*, a *Peano curve*, a *tri-Hilbert curve*, or a *tri-Peano curve* to convert an individual grid cell into the corresponding X value in the final transformation space.

Note that as the size of a query range increases, the performance of algorithm **Naive-Range-Query** deteriorates rapidly. This is because the algorithm requires a great number of space transformation operations to convert a spacious query region into a large set of one-dimensional X values. As mentioned before, a query range in one-dimensional original space is transformed into a query region in twodimensional intermediate space. However, the trail of the space-filling curve may not be continuous within the given query region. Therefore, function **QueryRangeToLineSegments()** has to execute a space transformation operation (i.e., function **SpaceFillingCurves()**) for each of the grid cells within the query region in intermediate space to make a set of line segments in the final transformation space.

Let Nt denote the number of space transformation operations for a range query. It is obvious that Nt is always between n and $(n^2 + n)/2$ (i.e., $n \leq Nt \leq (n^2 + n)/2$), where n denotes the grid size of the intermediate space. As a result, Nt is expressed as $O(n^2)$ in the worst case, which can be a primary factor that deteriorates the performance of algorithm **Naive-Range-Query**.

Algorithm 1: **Naive-Range-Query**

Input: query start q_s, query end q_e, grid size n
Output: set of results QR
1 LS = **QueryRangeToLineSegments**(q_s, q_e, n);
2 QR = **GetObjectsUsingBTreeIndex**(LS);
3 return QR;

Algorithm 2: **QueryRangeToLineSegments**

Input: query start q_s, query end q_e, grid size n
Output: set of line segments LS
1 *Xvalues* = { };
2 for each $q(X_s, X_e)$ in intermediate space do
3 **if**($X_s \leq q_e$ and $X_e \geq q_s$) **then**
4 *Xvalues* = *Xvalues* ∪ **SpaceFillingCurves**(X_s, X_e, n);
5 LS = **MakeLineSegments**(*Xvalues*);
6 return LS;

3.2 Proposed algorithm with the quarter division technique

This section presents a *quarter division technique* proposed for reducing the cost of space transformation operations and describes our range query algorithm based on the quarter division technique. The idea of the proposed quarter division technique can be explained briefly as follows: If a query region is divided into a set of non-overlapping subregions within each of which the trail of a space-filling curve is continuous, then only a single execution of a space transformation operation is needed for each subregion, not for each grid cell of the query region. The quarter division technique recursively divides the intermediate space into four subregions (i.e., quarters) within each of which the trail of a space-filling curve is continuous and examines whether each quarter is contained completely in the query region. If so, a space transformation operation is applied to the quarter only once in order to extract the corresponding line segment into the final transformation space.

Here, we assume that the Hilbert curve is used as a space-filling curve. The trail of the Hilbert curve is continuous within

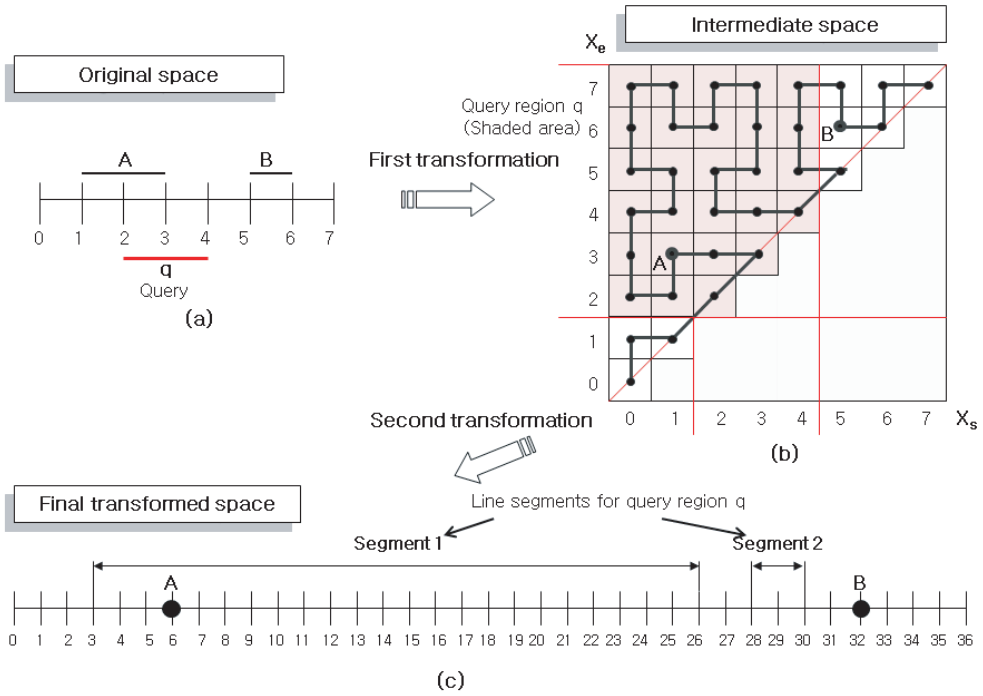


Figure 1 Example of a transformation process using DOT.

the intermediate space of n^*n grids. Also, since the intermediate space of n^*n grids is divided into four quarters of $n/2^*n/2$ grids, the trail of the Hilbert curve is still continuous within each quarter. Figure 2 shows the trails of the Hilbert curves of order 1, 2, 3, and 4 in two-dimensional space. If we analyze these trails, we can estimate each quarter's range of continuous X values in the final transformation space. However, it is necessary to determine the X value of the grid cell at which the Hilbert curve begins within each quarter. Let *firstXvalue* denote such an X value. Then, for a given quarter, its range of continuous X values in the final transformation space is expressed as $[firstXvalue, firstXvalue + n^2 - 1]$.

The trail of the Hilbert curve can be summarized as follows. For simplicity, we call the four sub-quarters of a quarter the 1st, 2nd, 3rd, and 4th sub-quarters, respectively, as shown in Figure 3(a).

1. In the case of order 1, the Hilbert curve visits each quarter in the order of 3, 4, 1, and 2 in intermediate space, and its trail is represented as Pattern3412. For each sub-quarter, the trail of the Hilbert curve of order 2 is shown in Figure 3(b).

2. The trail of the Hilbert curve has regularity in the recursive division of a quarter. The Hilbert curve consists of four patterns, Pattern3412, Pattern3214, Pattern1432, and Pattern1234. Table 1 shows the patterns of the Hilbert curve in sub-quarters according to the recursive division of a quarter. We can find a starting point of the Hilbert curve from these patterns. For example, when a quarter has a pattern, either Pattern3412 or Pattern3214, the trail begins at the lower left corner of the quarter (i.e., at the 3rd sub-quarter).

[Algorithm 3] shows a new query transformation algorithm, **QueryRangeToLineSegments()**. The algorithm calls function **SplitQueryRangeByQuarter()** to obtain a set of line segments by using the proposed quarter division technique (line 3). Let us now look into function **SplitQueryRangeByQuarter()** described in [Algorithm 4]. Here, Q and Q_{sub} denote a quarter and a sub-quarter, respectively, and both of them have three fields: *size, upperleft*, and *pattern* which represent the grid size, the upper left point, and a trail pattern of the Hilbert curve. Function *Hilbert()* represents a space transformation function based on the Hilbert curve. If a quarter Q is completely contained in a query region, the algorithm finds the starting and ending points of the Hilbert curve and returns a continuous line segment for this quarter (lines 1-4). Otherwise, the algorithm divides the quarter into four sub-quarters and calls function **SplitQueryRangeByQuarter()** for each sub-quarter, recursively (lines 5-10).

The function **SplitQueryRangeByQuarter()** executes a space transformation operation for each quarter. When the grid size of an intermediate space is n, the maximum and minimum numbers of quarters are $2n - 1$ and n, respectively, depending on the query range. As mentioned before, Nt denotes the number of space transformation operations for a range query. Then, it is obvious that, when using algorithm **QueryRangeToLineSegments()** in [Algorithm 3], Nt is between n and $2n - 1$ (i.e., $n \leq Nt \leq 2n - 1$). Therefore, the algorithm has complexity $O(n)$ in the worst case and achieves a great improvement in performance.

Algorithm 3: **QueryRangeToLineSegments**

Input: query start q_s, query end q_e, grid size n
Output: set of line segments LS
1 LS = { };
2 Q = **InitializeQuarter**(q_s, q_e, n);
3 **SplitQueryRangeByQuarter**(Q, q_s, q_e, LS);
4 return LS;

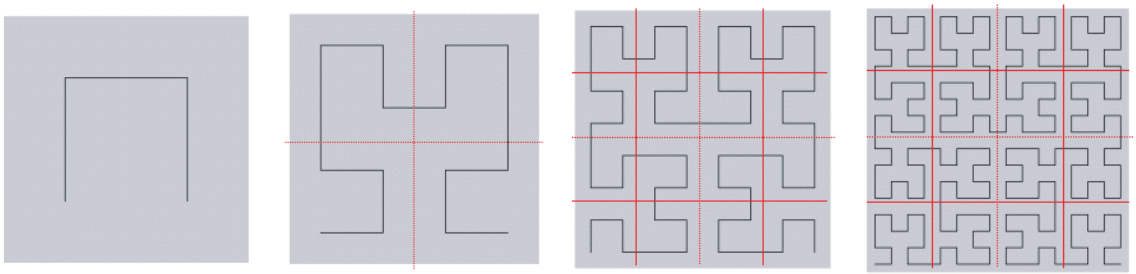


Figure 2 Hilbert curves of order 1, 2, 3, and 4.

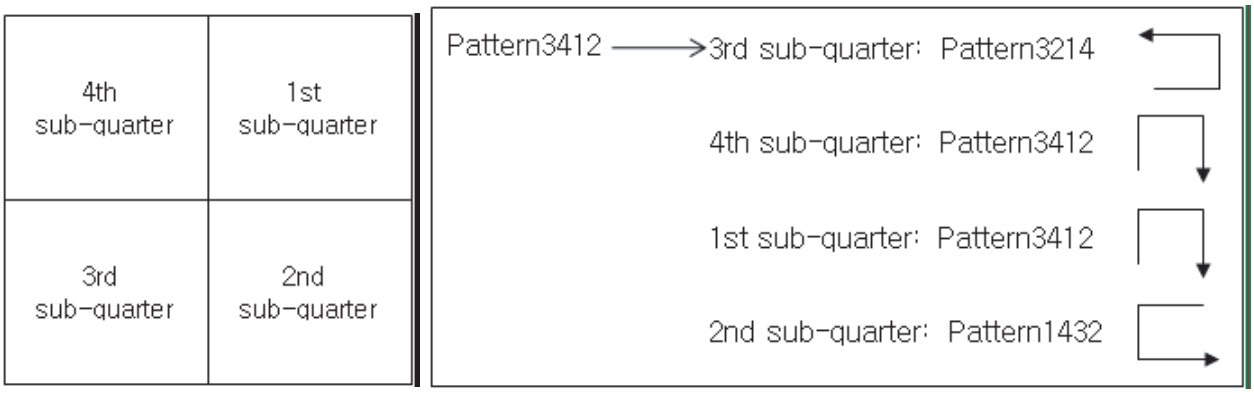


Figure 3 Ordering of the sub-quarters of a quarter, and example of patterns of sub-quarters in the case of a Hilbert curve of order 2.

Table 1 Patterns of the Hilbert curve in sub-quarters according to the recursive division of a quarter.

Order	Order + 1			
	1st sub-quarter	2nd sub-quarter	3rd sub-quarter	4th sub-quarter
Pattern3412	Pattern3412	Pattern1432	Pattern3214	Pattern3412
Pattern3214	Pattern3214	Pattern3214	Pattern3412	Pattern1234
Pattern1432	Pattern1234	Pattern3412	Pattern1432	Pattern1432
Pattern1234	Pattern1432	Pattern1234	Pattern1234	Pattern3214

Algorithm 4: **SplitQueryRangeByQuarter**

Input: quarter Q, query start q_s, query end q_e, set of line segments LS
Output: set of line segments LS

```
1   if(IsInRegion(Q, q_s, q_e)) then
2       LS_new.Start = Hilbert(StartPoint(Q));
3       LS_new.End = LS_new.Start + Q.size^2 − 1;
4       LS = LS ∪ LS_new;
5   else if (Q.size > 1) then
6       Q_sub.size = Q.size/2;
7       for(SubQuarter = 1; SubQuarter <= 4;
            SubQuarter + +) do
8           Qsub.pattern =
              nextpattern(Q, SubQuarter);
9           Qsub.upperleft =
              nextupperleft(Q, SubQuarter);
10          SplitQueryRangeByQuarter
              (Q_sub, q_s, q_e, LS);
11  return LS;
```

3.3 Redefinement with a tri-quarter division technique

As explained in Section 3.1, we assume that all the spatial objects and query regions are located within the upper triangular region in intermediate space. If we employ such a space-filling curve (e.g., tri-Hilbert curve) that traverses only the upper triangular region, a right-angled triangle whose oblique side is a part of the diagonal line can be a region where the trail of the space-filling curve is continuous. Here, we call such a right-angled triangle a tri-quarter. By exploiting this concept, we derive a *tri-quarter division technique*, which divides a query region into quarters or tri-quarters. This technique prevents a query region from being divided into quarters that are too small near the diagonal line, and further reduces the cost of space transformations. Let us illustrate a tri-quarter division technique for processing range queries. While Figure 4 shows a query region that is divided into quarters, Figure 5 shows the same query region that is divided into tri-quarters by the tri-quarter division technique. Here, the bold-lined 17 squares in Figure 4 and the bold-lined 3 triangles

near the diagonal line in Figure 5 correspond to quarters and tri-quarters, respectively. Therefore, a total of 14 space transformation operations are cut down by the proposed tri-quarter division technique.

Function **SplitQueryRangeByQuarter()** needs to be modified to consider tri-quarter regions in intermediate space. The modified version of function **SplitQueryRangeByQuarter()** is shown in [Algorithm 5]. Here, the function **TriHilbert()** in line 2 represents a space transformation function based on the tri-Hilbert curve. Note that a new variable, *TriQuarter*, is added to check the existence of triquarter (line 3), and a new formula is inserted into line 4 to calculate a line segment for a tri-quarter.

Algorithm 5: **SplitQueryRangeByQuarter**

Input: quarter Q, query start q_s, query end q_e, set of line segments LS
Output: set of line segments LS

```
1  if(IsInRegion(Q, q_s, q_e)) then
2    LS_new.Start = TriHilbert(StartPoint(Q));
3      if(Q.TriQuarter = TRUE) then
4        LS_new.End = LS_new.Start+
         (Q.size^2 + Q.size)/2 − 1;
5      else
6          LS_new.End = LS_new.Start + Q.size^2 − 1;
7          LS = LS ∪ LS_new;
8  else if (Q.size > 1) then
9        Q_sub.size = Q.size/2;
10       for(SubQuarter = 1; SubQuarter <= 4;
         SubQuarter++) do
11         Q_sub.pattern =nextpattern(Q, SubQuarter);
12         Q_sub.upperleft =nextupperleft(Q, SubQuarter);
13         SplitQueryRangeByQuarter(Q_sub, q_s, q_e, LS);
14   return LS;
```

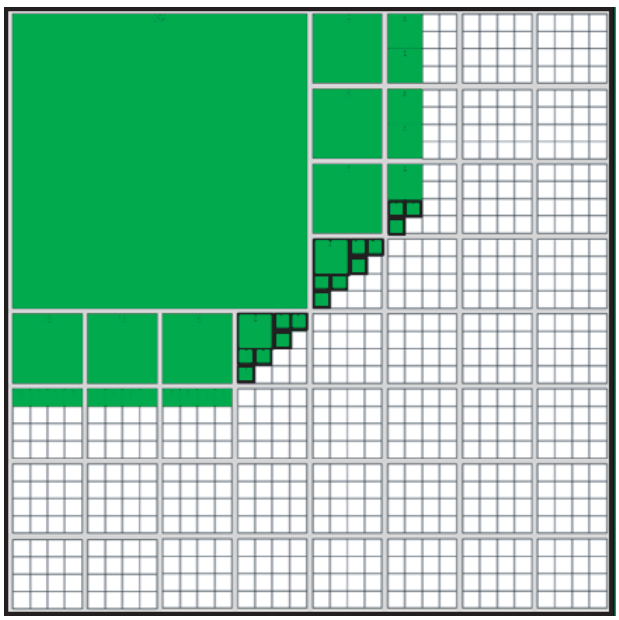


Figure 4 Example of a query region divided into quarters.

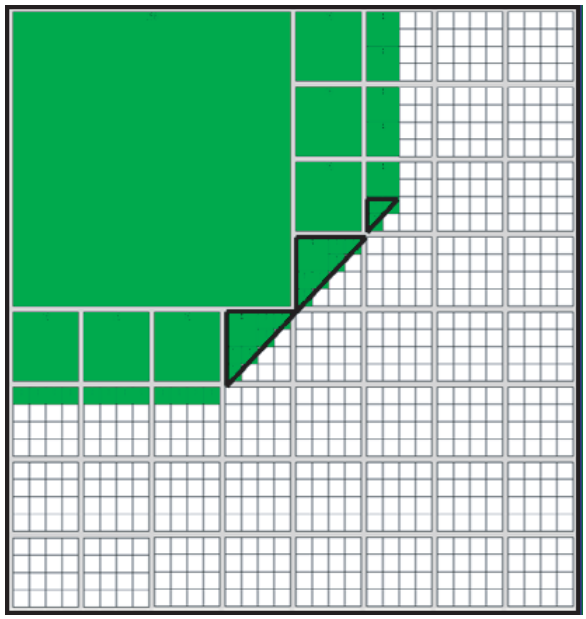


Figure 5 Example of a query region divided into quarters and tri-quarters.

4. SPATIAL JOIN ALGORITHM BASED ON DOT

In this section, we propose an efficient spatial join algorithm based on the DOT index. First, in Section 4.1, we illustrate the proposed spatial join algorithm with an example, which was designed for helping readers obtain an intuitive understanding of our algorithm, and explain how our algorithm improves performance. Next, in Section 4.2, we describe the proposed algorithm in detail.

4.1 Basic concept and intuitive example

A spatial join involves two sets of spatial objects and, unlike a range query and a point query, its query region is not fixed. When two files, each of which stores a set of spatial objects, have no indices on them, their spatial join is costly because each spatial object of one file has to be compared with all spatial objects of another file. Let us suppose we have two files, R and S, equipped with DOT indices. A simple method to obtain the result of their spatial join is to execute the range query algorithm explained in Section 3, repeatedly. That is, we can obtain the result of the spatial join between files R and S by, for each spatial object of file R, first establishing a query region from its MBR and then executing a range query to discover all the spatial objects of file S that overlap the query region.

Let us illustrate such a procedure for spatial join processing with an example in Figure 6. Figure 6(a) shows four one-dimensional objects (i.e., a, b, c, and d) stored in file R with which file S is to be joined spatially. Through the first transformation, these four objects are mapped into the corresponding four points in two-dimensional intermediate space in Figure 6(b). In this example, the operation of joining files R and S is the same as the repetitive execution of a range query, whose query region

is from the MBR of each one of four objects (i.e., a, b, c, and d) of file R on file S. The final result of this join operation is obtained by collecting the results of four range queries.

For example, let us consider the range query for spatial object a of file R. Since object a is positioned at (6, 8) in the intermediate space of file R, its query region is the intersection of the three regions (i.e., the shaded region in Figure 6(c)), $X_s \leq 8$, $X_e \geq 6$, and the region above the diagonal line, in the intermediate space of file S. The query regions of objects b, c, and d are similar to those of object a. Through the second transformation, such query regions in intermediate space are converted to the corresponding query regions in final one-dimensional space, as shown in Figure 6(d). Here, sa, sb, sc, and sd represent the final query regions of objects a, b, c, and d, respectively. Next, using a set of X value ranges of each final query region, we search file S for the objects overlapping some of the query region.

Note that, if objects are close to each other in original space, like the objects of file R in the above example, then they are mapped to the adjacent points in intermediate space via the first transformation. Adjacent points in intermediate space are then mapped to similar X values in final space via the second transformation, which is based on a space-filling curve. The reason for adjacent points in intermediate space to be mapped to similar X values in final space is that space-filling curves are usually designed to preserve the adjacency of points in intermediate space as much as possible. We explained in Section 2 that, from the X values of the final space, we construct a DOT index of a B^+-tree structure for efficient processing of range queries and spatial joins. Hence, we can expect it is highly possible for adjacent objects to be stored on the same leaf page of a DOT index.

We store objects in a DOT index by inserting their X values and MBR information (i.e., $[X_s, X_e]$) into appropriate leaf pages as shown in Figure 7(b). Here, we can think of the smallest MBR that encloses the MBRs of all objects in a leaf page. That is, for a given leaf page, when we let $minX_s$ denote the smallest one among the X_s values of all objects in the leaf page and $maxX_e$ denote the largest one among the X_e values of all objects in the leaf page, $[minX_s, maxX_e]$ becomes the smallest MBR that encloses the MBRs of all objects in the leaf page. In this paper, we call such an MBR an 'LP-MBR' (Leaf Page MBR).

By exploiting the concept of LP-MBR, we can significantly reduce the number of range queries needed for executing spatial joins. That is, rather than sequentially taking each object of file R and executing a range query with its MBR as a query region, we can take each leaf page of the DOT index constructed for file R and execute a range query with its LB-MBR as a query region. Let N be the number of objects of file R and M be the average number of objects in a leaf page of the DOT index for file R. While the naive approach requires N range queries for a spatial join, the approach exploiting the concept of LP-MBR requires N/M range queries for the same operation. Therefore, we can expect a considerable reduction in the number of range queries for spatial joins and consequently a significant reduction in execution time.

Of course, this approach necessitates an extra step for examining every object pair returned from the proposed spatial join algorithm to verify whether they really overlap with each other. However, since the number of object pairs returned from the proposed spatial join algorithm is not that large in most cases, this extra step does not cost much. Here, we call such a query region of LP-MBR a *candidate join region*.

Figure 7 illustrates each step of our spatial join algorithm based on the DOT index. Figure 7(a) shows 12 objects (i.e., $A \ldots L$) of file R, with which file S is to be joined spatially, in one-dimensional space. Figure 7(b) depicts the DOT index structure for file R. This index structure has three leaf pages and their LPMBRs are [1, 7], [4, 9], and [8, 15], respectively. Figure 7(c) depicts the candidate join region for each LP-MBR of the three leaf pages. In this figure, we can easily observe that there are substantial overlaps among the candidate join regions of the three leaf pages. For example, the candidate join region of leaf page 1 consists of ①, ②, and ③, and the candidate join region of leaf page 2 consists of ②, ③, and ④. Therefore, in this example, regions (2) and (3) are overlaps between the candidate join regions of leaf pages 1 and 2. Figure 7(d) shows the overlapping status of the candidate join regions in the final one-dimensional space.

In case the concept of LP-MBR is not employed, spatial join requires processing of a range query for *every object* by using its MBR as a query region. This causes a lot of disk accesses, thereby deteriorates the performance of overall join processing. Figure 7(e) shows a number of line segments obtained from 12 objects in Figure 7(b), each of which is used as a query region for range query processing.

The substantial overlaps among the candidate join regions to be accessed sequentially can be exploited to further improve the performance of the proposed spatial join algorithm. More specifically, with the LRU buffering policy, if we execute range queries while visiting every leaf page of file $R's$ DOT index from left to right, then we can significantly reduce the number of disk accesses to file S.

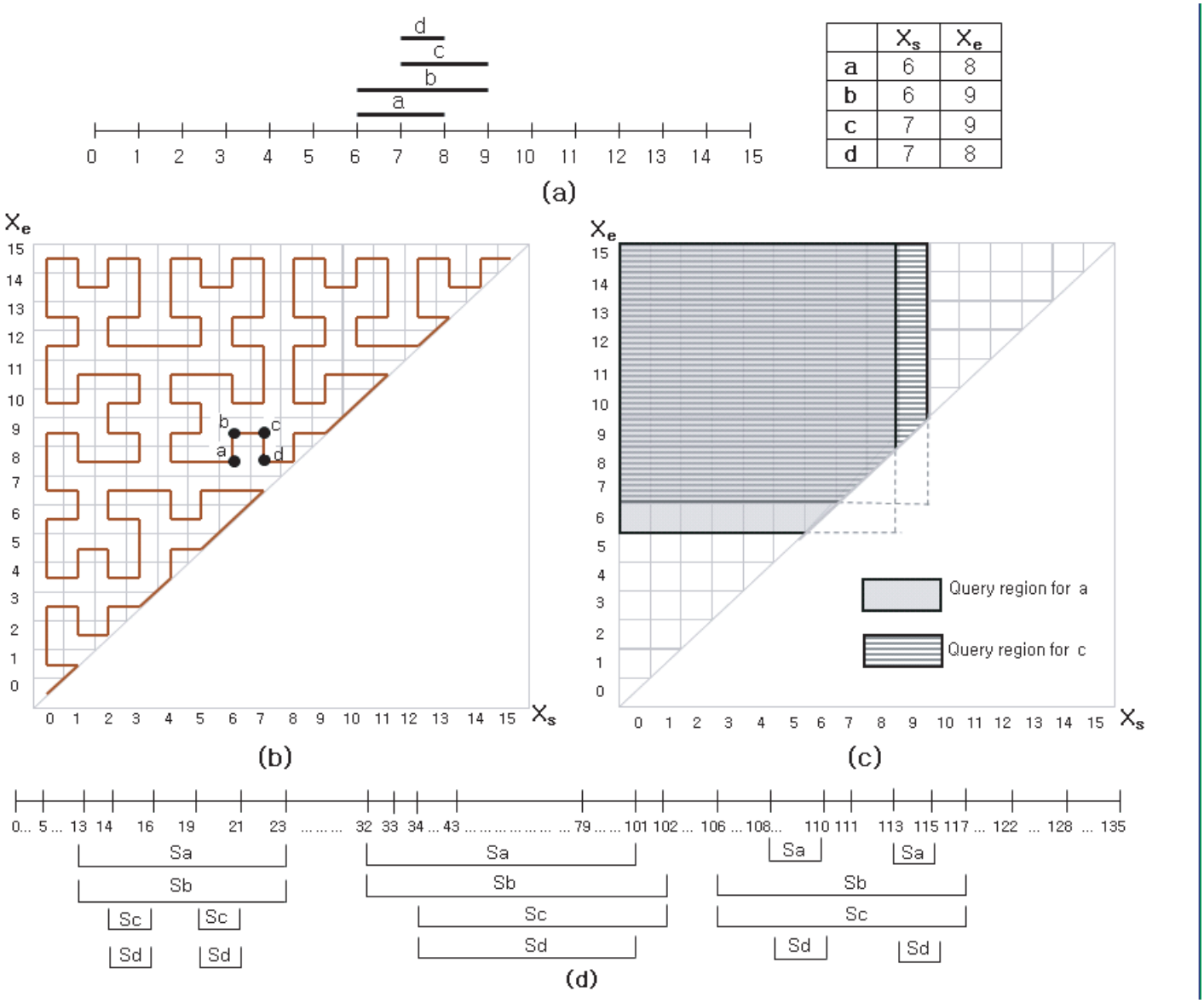
4.2 Detailed algorithm

[Algorithm 6] shows the proposed spatial join algorithm based on the DOT index. Here, we assume that files R and S store one-dimensional objects and DOT indices are constructed for them. In this algorithm, n is the number of grids in each dimension of intermediate space. We employ the LRU buffering policy, as mentioned in Section 4.1, to reduce the number of disk accesses to file S. Among the various spatial relationships, we consider only the intersection of spatial objects as their join condition.

Algorithm 6: **DOT-based Spatial Join**

Input: file R, file S, grid size n
Output: set of results RS

```
1   RS = { };
2   LeafPages = GetLeafPagesFromLeftToRight(DOT(R));
3   for(i = 0; i < |LeafPages|; i++) do
4     PageR = LeafPages(i);
5     minX_s = MIN(PageR.X_s);
6     maxX_e = MAX(PageR.X_e);
7     LS = QueryRangeToLineSegments(minX_s, maxX_e, n);
8     for(j = 0; j < |LS|; j++) do
9           PageS = GetIntersectedLeafPage(DOT(S), LS(j));
10          RS = RS ∪ Intersect(PageR, PageS);
11  return RS;
```



	X_s	X_e
a	6	8
b	6	9
c	7	9
d	7	8

Figure 6 Relationships among query regions.

[Algorithm 6] is organized to execute the internal statements of the outer for loop while visiting the leaf pages of the B^+-tree for file R from left to right. Within the outer for loop, we first read a corresponding leaf page of file R (line 4) and construct its LP-MBR by assigning to $minX_s$ the smallest one among the X_s values of all objects in the leaf page and assigning to $maxX_e$ the largest one among the X_e values of all objects in the leaf page (lines 5-6). Next, we compute the candidate join region of the leaf page by calling function **QueryRangeToLineSegments()** with $minX_s$ and $maxX_e$ as its parameters (line 7). Here, function **QueryRangeToLineSegments()** converts the query range of the current leaf page of file R to a set of line segments using the quarter division technique (or tri-quarter division technique) explained in Section 3. Next, using the DOT index of file S, we retrieve the objects of file S that intersect some of the candidate join regions of the current leaf page of file R (line 9) and examine every pair of objects in the current leaf page of file R and objects retrieved from file S. Only the pairs of objects that intersect each other are included in the final result set (line 10).

5. PERFORMANCE EVALUATION

This section verifies the superiority of the proposed method via performance evaluation with extensive experiments.

5.1 Environment

Two kinds of data sets were used for experiments: synthetic data sets and real world geographical data sets. We generated synthetic data sets that consist of spatial objects of various sizes and distributions. Figure 8 shows three spatial object distributions in intermediate space used in our experiments. The sizes of objects are within the range of [0, 1023]. Figure 8(a) shows the uniform distribution of spatial objects with various sizes. Figure 8(b) shows the strip distribution, where only small spatial objects are located densely near the diagonal line. Figure 8(c) shows the 'strip plus uniform' distribution where there are many small spatial objects with a few large spatial objects in uniform

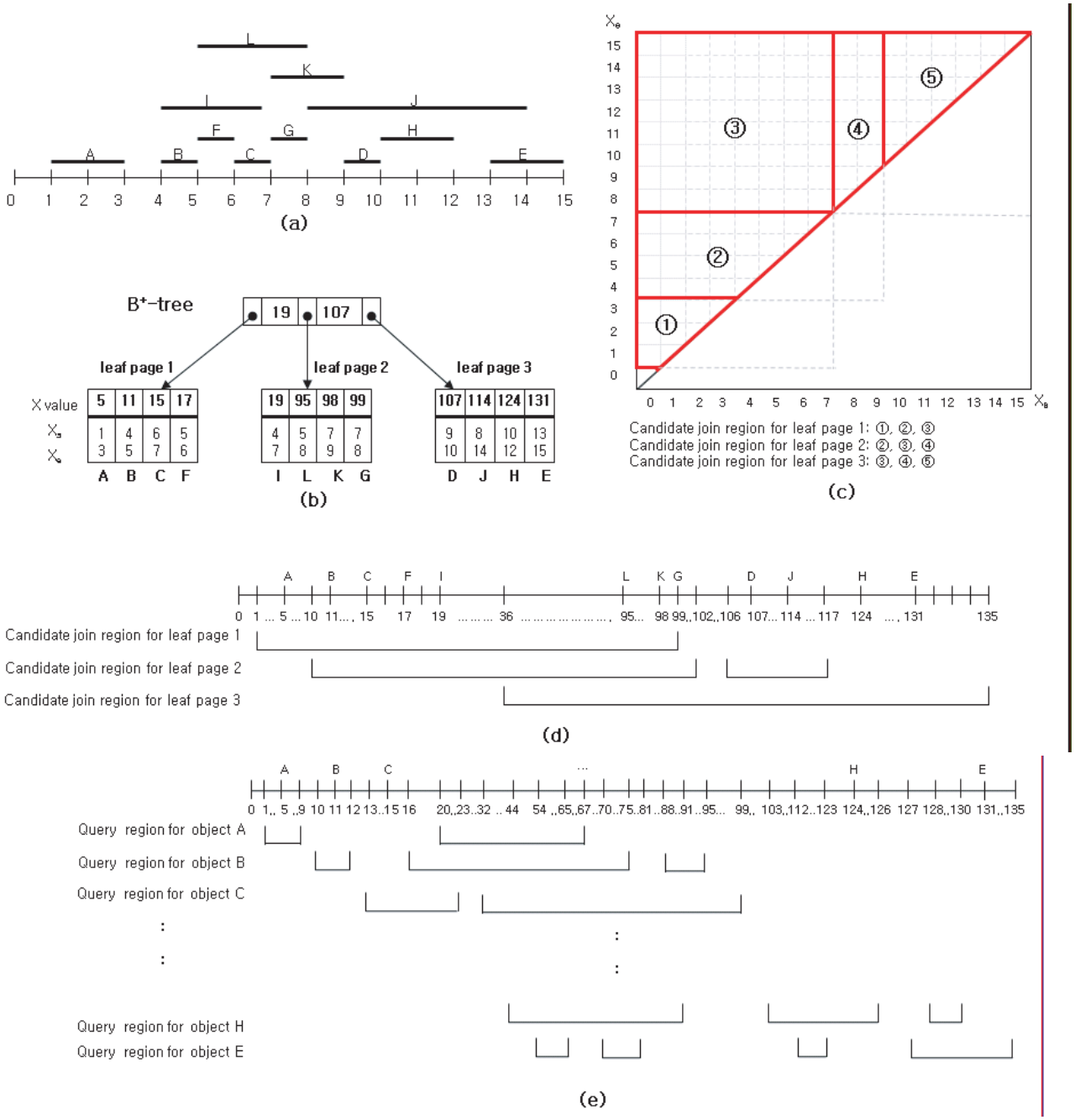


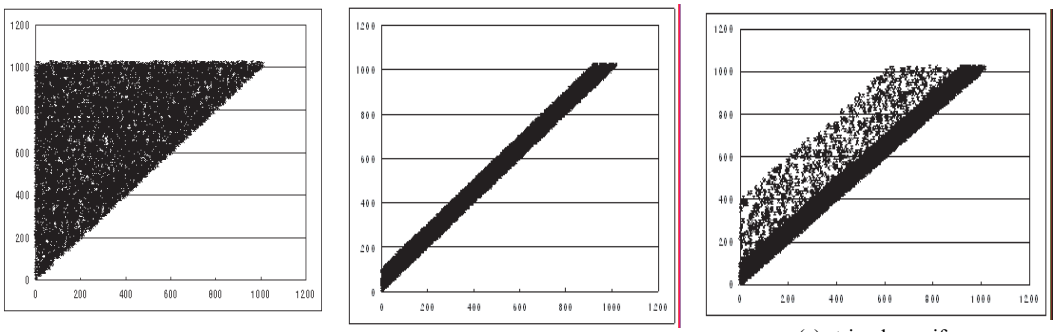
Figure 7 Spatial join processing with DOT index.

random sizes. For experiments, we produced 10,000 to 300,000 spatial objects with three distributions.

Real geographical data sets were downloaded from R-tree Portal [39]. Table 2 shows four geographical data sets used in our experiments.

We evaluated the performance of the spatial join methods using the DOT index and that of the R*-treebased spatial join method. We compared the following four spatial join methods using the DOT index. **DOT-TriQuarter** and **DOT-Quarter** are the methods that perform spatial joins with our *tri-quarter* and *quarter* division techniques, respectively. **DOT-Range** is a method that performs spatial joins without any quarter division technique. These three methods employ the concept of LP-MBR (Leaf Page MBR) and optimize the number of range queries needed for executing spatial joins. The last method, **DOT-TriQuarter-Naive**, is a naive version of **DOT-TriQuarter**, which does not use the concept of LP-MBR. **DOT-TriQuarter-Naive** takes each spatial object sequentially and executes a range query with its MBR as a query region for executing spatial joins. The grid size, n, was set to 1,024 in all these methods. **R*-TreeJoin** is the previous method based on the R*-tree, which was developed by Seeger et al. [39]. We used the *tri-Hilbert curve* as a space-filling curve. We also set the page size to 4 KB.

The hardware platform was a Pentium IV 2GHz PC equipped with 1GB main memory and 120 GB HDD. The software platform was a Pedoracore 3 Linux system. The performance factors were the number of spatial transformation operations, the number of disk accesses, and the total elapsed time for processing a



(a) uniform distribution. (b) strip distribution. (c) strip plus uniform distribution.

Figure 8 Three object distributions.

Table 2 Real geographical data sets used in experiments.

Data Set	Data Size(MB)	Number of Points	
CITY	0.1	5,922	cities and villages of Greece
SEQ	0.5	62,556	place names of California
NE	0.8	89,332	postal addresses of North East
DCW	1.1	110,836	populated places in US, Canada, Mexico

spatial join. In the experiments, we performed spatial joins on two identical sets of data.

5.2 Results and analyses

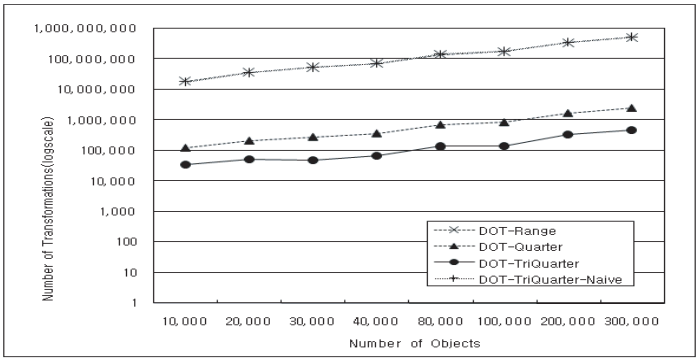
In Experiment 1, we compared the performance of **DOT-TriQuarter**, **DOT-Quarter**, **DOT-Range**, and **DOT-TriQuarter-Naive** in terms of the number of spatial transformation operations. Figure 9 shows the results with different numbers of spatial objects. The X axis denotes the number of spatial objects, and the Y axis denotes the number of spatial transformation operations required for processing a join. The results show that **DOT-TriQuarter** and **DOT-Quarter** noticeably reduce the number of spatial transformation operations as compared with that of **DOT-Range** in all the distributions owing to the quarter division techniques. Also, **DOT-TriQuarter** requires only one transformation operation for every leaf page MBR, and therefore is noticeably better than **DOT-TriQuarter-Naive**, which requires the transformation operations for the MBRs of all the objects stored in the leaf page.

The results show that **DOT-TriQuarter** performs noticeably better than **DOT-Quarter** in uniform distribution and performs slightly better than **DOT-Quarter** in strip and strip plus uniform distributions. When small spatial objects are close to a diagonal line as in strip and strip plus uniform distributions, the performances of these two methods are almost the same because a query region is divided into very small quarters. According to our experimental results, **DOT-TriQuarter** shows the best performance regardless of distribution. Compared with **DOT-TriQuarter-Naive**, **DOT-Range**, and **DOT-Quarter**, **DOTTri-Quarter** performs 514.5 to 1305.8 times, 533 to 1250.8 times, and 3.5 to 6.2 times better, respectively, in uniform distribution. In strip distribution, it performs 461.6 to 1032.8 times, 210 to 420.7 times, and 1.4 to 1.6 times better, respectively. In strip plus uniform distribution, it performs 605.4 to 1079.3 times, 300.1 to 474 times, and 1.5 to 1.9 times better, respectively.

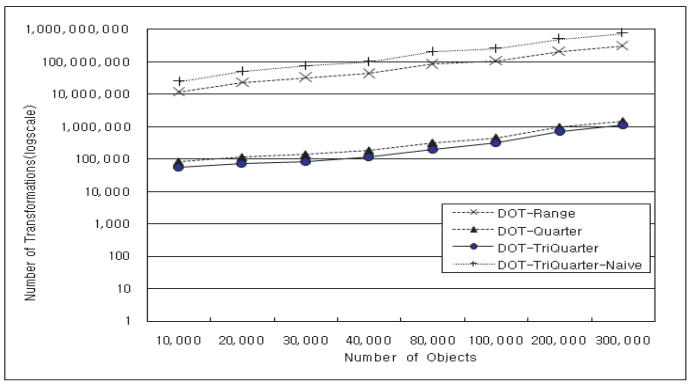
In Experiment 2, we compared the performance of **DOT-TriQuarter**, **DOT-TriQuarter-Naive**, and **R*-TreeJoin** in terms of the number of disk accesses. Here, we excluded **DOT-Quarter** and **DOT-Range** because the number of disk accesses is the same as **DOT-TriQuarter**. The data sets used in this experiment have 300,000 spatial objects with the three distributions. In uniform, strip, and strip plus uniform distributions, the numbers of data pages in **DOT-TriQuarter** (or **DOT-TriQuarter-Naive**) were 1,510, 1,471, and 1,490, respectively. The numbers of data pages in **R*-TreeJoin** were 2,106, 2,059, and 2,075, respectively. Figure 10 shows the results with different sizes of buffers. The X axis denotes the size of buffers under the LRU buffer replacement strategy. The Y axis denotes the number of disk accesses occur while processing a spatial join.

As shown in Figure 10, **DOT-TriQuarter** and **DOT-TriQuarter-Naive** require more disk accesses than **R*-TreeJoin** when the buffer is small. This is because the spatial candidate join regions, which are overlapped among adjacent data objects (pages), may not stay in the buffer for continuous range queries. In particular, because **DOT-TriQuarter-Naive** establishes a query region from the MBRs of all the spatial objects instead of a leaf page MBR, it requires more disk accesses than **DOT-TriQuarter**. However, as the buffer size becomes larger, these methods run faster than **R*-TreeJoin** because overlapped spatial candidate join regions tend to stay in the buffer.

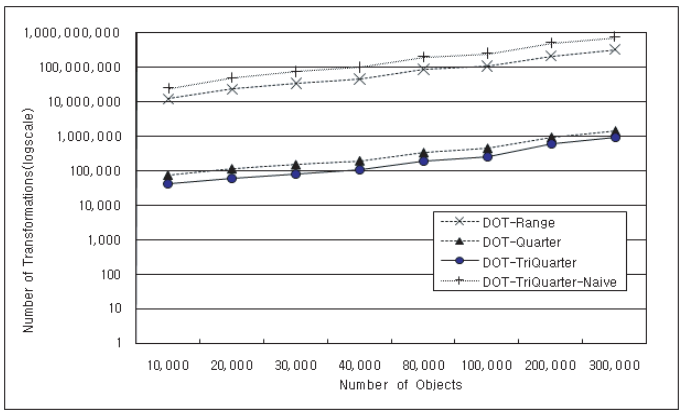
We assume that the optimal spatial join algorithm accesses every data page only once with the minimal buffer. Figure 10 shows that the number of disk accesses with **DOT-TriQuarter** and **DOT-TriQuarter-Naive** approaches the optimal number when the buffer size is 2,048 KB (about 17% of total data pages) in uniform and strip plus uniform distributions and when the buffer size is 1,024 KB (about 8.7% of total data pages) in strip distribution. On the other hand, the number of disk accesses with **R*-TreeJoin** is close to the optimal number when the buffer size is 3,072 KB (about 18% of total data pages) in uniform and strip distributions and when the buffer size is 4,096 KB (about 24.6% of total data pages) in strip plus uniform distribution.



(a) uniform distribution.



(b) strip distribution.

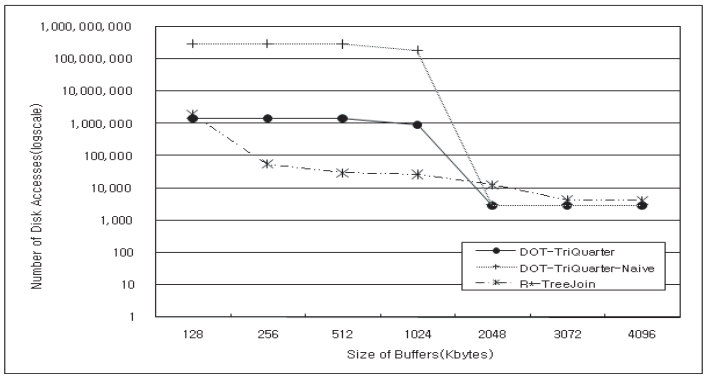


(c) strip plus uniform distribution.

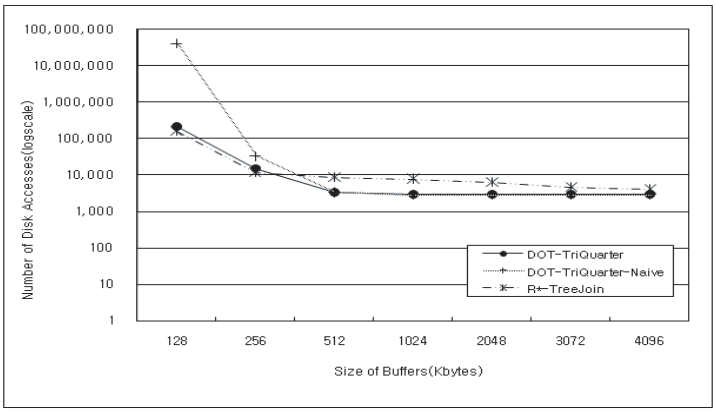
Figure 9 Number of spatial transformation operations with different numbers of objects.

In Experiment 3, we compared the performance of **DOT-TriQuarter**, whose performance is best in Experiments 1 and 2, with that of **R*-TreeJoin** in terms of the total elapsed time for spatial join processing. Here, we developed the spatial join algorithm using the tri-Peano curve in order to compare the performance of the proposed method with different space-filling curves. **DOT-TriQuarter-Hilbert** denotes our method employing the *tri-Hilbert curve* for the second transformation. **DOT-TriQuarter-Peano** denotes another method employing the *tri-Peano curve* instead of the tri-Hilbert curve. The total elapsed time for these two methods consists of the extraction time, the search time, and the post processing time. The extraction time is the time spent in obtaining a set of line segments (spatial candidate join region) by spatial transformation for a query range using the *quarter division technique*. The search time is the time spent in finding spatial objects in a DOT index. The post processing time is the time spent in verifying if the searched spatial object intersects with a query range. Figure 11 shows the results according to the change in the number of objects for the three distributions. The buffer size was set to 4,096 KB, where we saw that the numbers of disk accesses with **DOT-TriQuarter** and **R*-TreeJoin** were almost the same as those in Experiment 2. The Y axis denotes the elapsed time in msec on a logscale.

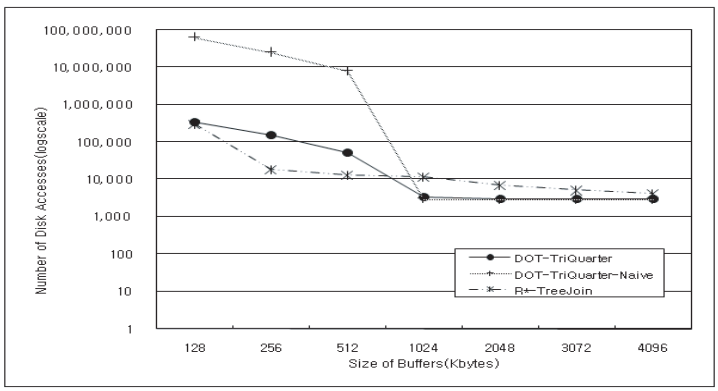
The results show that **DOT-TriQuarter-Hilbert** performs slightly better than **DOT-TriQuarter-Peano** in all three distributions. **DOT-TriQuarter-Hilbert** is 2.7 to 3.3 times faster for uniform distribution compared with **R*-TreeJoin**. Also, we see that spatial join processing with **DOT-TriQuarter-Hilbert** and **R*-TreeJoin** for uniform distribution requires more processing time than those in strip and strip plus uniform distributions. Because spatial objects of various sizes are uniformly distributed over entire spaces, these methods produce a large result set.



(a) uniform distribution.



(b) strip distribution.



(c) strip plus uniform distribution.

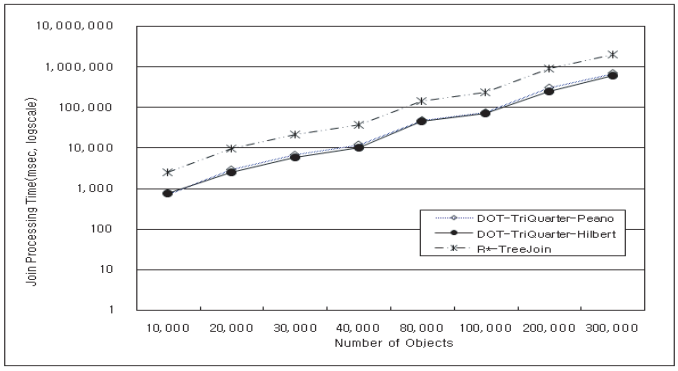
Figure 10 Number of disk accesses according to the change of the buffer size.

Compared with **R*-TreeJoin, DOT-TriQuarter-Hilbert** is 2.9 to 4.5 times faster for strip distribution and 2.9 to 3.8 times faster for strip plus uniform distribution.

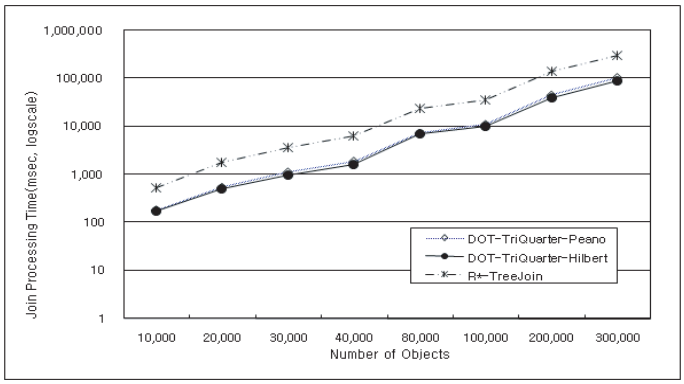
In Experiment 4, using four real geographical data sets, we compared the performance of the **DOT-TriQuarter-Hilbert** and **R*-TreeJoin** methods in terms of the index size, the index construction time and the total elapsed time for spatial join processing. For all four data sets, the buffer sizes were set to 4,096 KB according to the result of Experiment 2. The index construction time for the **DOT-TriQuarter-Hilbert** consists of the transformation time, the sort time, and the insertion time. The transformation time is spent for transforming MBRs of spatial objects in one-dimensional original space into X values in final space. The sort time is spent for sorting spatial objects according to their X values. The insertion time is spent for inserting spatial objects with X values into the DOT index using the B^+-tree structure. As shown in Table 3, **DOT-TriQuarter-Hilbert** has smaller index size than **R*-TreeJoin** for all four data sets. In terms of the index construction time, **DOT-TriQuarter-Hilbert** was 7.97 to 28.6 times faster than **R*-TreeJoin**.

Figure 12 compares the spatial join processing time of **DOT-TriQuarter-Hilbert** and **R*-TreeJoin** for four data sets. The Y axis denotes the elapsed time in msec on a log-scale. The results show that **DOT-TriQuarter-Hilbert** was 2.7 to 2.8 times faster than **R*-TreeJoin** for all of four data sets.

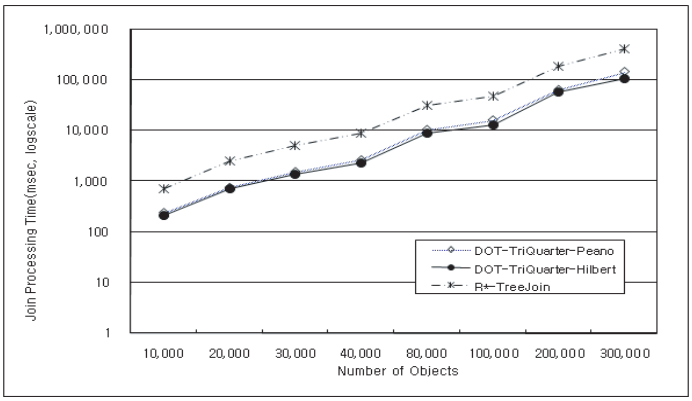
In summary, **DOT-TriQuarter-Hilbert** shows the best performance regardless of the distribution of spatial objects and is about 3 times faster than **R*-TreeJoin**.



(a) uniform distribution.



(b) strip distribution.



(c) strip plus uniform distribution.

Figure 11 Elapsed time for processing a spatial join with different numbers of objects.

Table 3 Index size and construction time with real geographical data sets.

Data Set	DOT-TriQuarter-Hilbert					R*-TreeJoin	
	Index Size (Bytes)	Construction Time (msec)				Index Size (Bytes)	Construction Time (msec)
		Trans_ Time	Sort_ Time	Insert_ Time	Total_ Time		
CITY	126,976	6	6	1	13	172,032	239
SEQ	1,261,568	71	70	13	154	1,822,720	4,410
NE	1,798,144	106	99	18	223	2,613,248	5,606
DCW	2,232,320	115	119	23	257	3,141,632	5,493

6. CONCLUDING REMARKS

DOT-based indexing maps spatial objects in original space onto points in one-dimensional space and stores them in a B^+-tree. The most important advantage of DOT-based indexing is that it is very easy to integrate into existing DBMSs because it employs ubiquitous B^+-trees as a base index structure. In this paper, we have proposed a novel spatial join algorithm using DOT-based indexing for one-dimensional spatial objects.

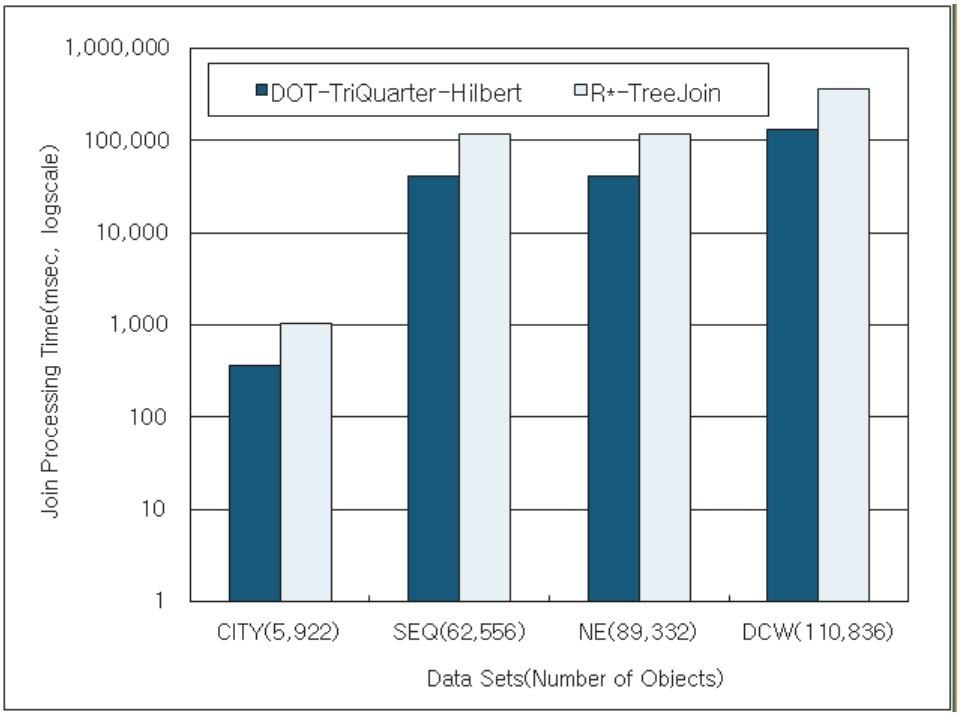


Figure 12 Elapsed time for processing a spatial join with real geographical data sets.

The proposed algorithm determines the access order of pages containing spatial objects and minimizes the number of buffer replacements occurring during the spatial join processing. Also, in order to reduce the number of space transformation operations, it uses a quarter division technique, which divides a query region into a set of subregions that contain consecutive space-filling curves as long as possible. The quarter division technique is applicable in any environment where space-filling curves are utilized. It is fairly effective, particularly with DOT or a corner transformation technique when objects are located in the right-angled triangle-shaped region above the diagonal in space.

To evaluate the performance of the proposed algorithm, we have performed extensive experiments with data sets of various distributions and sizes. With proper buffering, the proposed algorithm was shown to outperform the R*-tree-based spatial join algorithm, which is the one most widely used, by a factor approaching three.

As a further study, we plan to extend our algorithm to efficiently process spatial objects in two-dimensional original space. We note that the extension is not straightforward and also needs to deal with the following sub-issues: (1) the extension of DOT for two-dimensional original space, (2) the representation of objects and query regions in four-dimensional intermediate space, (3) the design of an effective space-filling curve in four-dimensional intermediate space, (4) the extension of a range query processing algorithm with DOT-based indexing to two-dimensional original space, and (5) the design of a spatial join processing algorithm based on the extended range query processing algorithm.

Acknowledgement

We thank professor Kyu-Young Whang for his providing us with the code of R*-tree buffering. This research was supported by the Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology (2010-0004815) and by the Semiconductor Industry Collaborative Project between Hanyang University and Samsung Electronics Co. Ltd, and the MKE(Ministry of Knowledge Economy), Korea and Microsoft Research, under IT/SW Creative Research Program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2012-H0503-12-1018).

REFERENCES

1. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. Vitter (1998). Scalable Sweeping Based Spatial Join. *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, 570–581.
2. N. Beckmann, H. Kriegel, and R. Schneider (1990). The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *Proc. Int'l. Conf. on Management of Data (ACM SIGMOD)*, 322–331.
3. C. Bohm, S. Beerchtold, and D. Keim (2001). Searching in High Dimensional Spaces-Index Structures for Improving the Performance of Multimedia Databases. *ACM Computing Surveys*, 33(3), 322–373.
4. T. Brinkhoff, H. P. Kriegel, and B. Seeger (1993). Efficient Processing of Spatial Joins Using R-trees. *Proc. Int'l. Conf. on Management of Data (ACM SIGMOD)*, 237–246.
5. T. Brinkhoff, H. P. Kriegel, R. Schneider, and B. Seeger (1994). Multi-Step Processing of Spatial Joins. *Proc. Int'l. Conf. on Management of Data (ACM SIGMOD)*, 197–208.
6. D. Comer, (1979). The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2), 121–137.
7. H. Dai and H. Su (2003). Approximation and Analytical Studies of Inter-cluster Performances of Space-Filling Curves. *Discrete Mathematics and Theoretical Computer Science*, 53–68.
8. J. Dai, and C. T. Lu (2008). Concurrency Control for Spatial Access Method. *Encyclopedia of Geographical Information Science*, 124–

128.
9. H. Ding, G. Trajcevski, and P. Scheuermann (2008). Efficient Similarity Join of Large Sets of Moving Object Trajectories. *Int'l. Symp. on Temporal Representation and Reasoning (IEEE TIME)*, 79–87.
10. J. Enderle, M. Hampel, and T. Seidl (2004). Joining Interval Data in Relational Databases. *Proc. Int'l. Conf. on Management of Data (ACM SIGMOD)*, 683–694.
11. C. Faloutsos, and S. Roseman (1989). Fractals for Secondary Key Retrieval. *Proc. Int'l. Symp. on Principles of Database Systems (ACM PODS)*, 247–252.
12. C. Faloutsos, and Y. Rong (1991). DOT: A Spatial Access Method Using Fractals. *Proc. Int'l. Conf. on Data Engineering (IEEE ICDE)*, 152–159.
13. R. Fenk, V. Markl, and R. Bayer (2002). Interval Processing with the UB-Tree. *Proc. Int'l. Symp. on Database Engineering and Applications (IDEAS)*, 12–22.
14. E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis (2007). Algorithms for Nearest Neighbor Search on Moving Object Trajectories. *Geoinformatica*, 11(2), 159–193.
15. O. Gunther (1986). The Cell Tree: An Index for Geometric Data. *Memorandum No. UCB/ERL M86/89*, Univ. of California, Berkeley.
16. O. Gunther (1993). Efficient Computation of Spatial Joins. *Proc. Int'l. Conf. on Data Engineering (IEEE ICDE)*, 50–59.
17. A. Guttman (1984). R-trees: A Dynamic Index Structures for Spatial Searching. *Proc. Int'l. Conf. on Management of Data (ACM SIGMOD)*, 47–57.
18. R. H. Guting (1994). An Introduction to Spatial Database Systems. *Special Issue on Spatial Database Systems of the VLDB Journal*, 3(4), 357–399.
19. A. Henrich, H -W. Six, and P. Widmayer (1989). The LSD Tree: Spatial Access to Multidimensional Point and Non Point Objects. *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, 45–53.
20. Y. -W. Huang, N. Jing, and E. A. Rundensteiner (1997). Rundensteiner: Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations. *Proc. Int'l. Conf. on Very Large Data Bases, (VLDB)*, 396–405.
21. C. S. Jensen, D. Lin, and B. C. Ooi (2004). Query and Update Efficient B+-Tree Based Indexing of Moving Objects. *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, 768–779.
22. H. Kriegel, M. Potke, and T. Seidl (2001). Interval Sequences: An Object-Relational Approach to Manage Spatial Data. *Proc. Int'l. Symp. on Spatial and Temporal Databases (SSTD)*, 481–501.
23. H. Kriegel, M. Schiwietz, R. Schneider, and B. Seeger (1989). Performance Comparison of Point and Spatial Access Methods. *Proc. Int'l. Symp. on Design and implementation of large spatial databases (SSD)*, 89–114.
24. J. Lawder and P. King (2001). Querying Multi-dimensional Data Indexed Using the Hilbert Space-Filling Curve. *ACM SIGMOD Record*, 30(1), 19–24.
25. M. Lee, W. Han, and K. Whang (2004). Transformation-based Spatial Partition Join. *Int'l Journal of Computer Systems Science and Engineering (CSSE)*, 19(6), 355–362.
26. M. Lee, K. Whang, W. Han, and I. Song (2006). Transform-Space View: Performing Spatial Join in the Transform Space Using Original-Space Indexes. *IEEE Trans. on Knowledge and Data Engineering*, 18(2), 245–260.
27. S. Lee, S. Park, W. Kim, and D. Lee (2007). An Efficient Location Encoding Method for Moving Objects Using Hierarchical Administrative District and Road Network. *Information Sciences*, 177(3), 832–843.
28. H.-Y. Lin (2008). Efficient and Compact Indexing Structure for Processing of Spatial Queries in Linebased Databases. *Data and Knowledge Engineering*, 64(1), 365–380.
29. B. Moon, H. V. Jagadish, C. Faloutsos, and J. Saltz (2001). Analysis of the Clustering Properties of Hilbert Space-Filling Curve. *IEEE Trans. on Knowledge and Data Engineering*, 13(1), 124–141.
30. J. Nievergelt, H. Hinterberger, and K. C. Sevcik (1984). The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. on Database Systems*, 9(1), 38–71.
31. J. Orenstein (1986). Spatial Query Processing in an Object-Oriented Database System. *Proc. Int'l. Conf. on Management of Data (ACM SIGMOD)*, 326–336.
32. J. Orenstein and F. Manola (1988). PROBE Spatial Data Modeling and Query Processing in an Image Database Applications. *IEEE Trans. on Software Engineering*, 14(5), 611–629.
33. W. Osborn and K. Barker (2007). An Insertion Strategy for Two-Dimensional Spatial Access Method. *Proc. Int'l. Conf. on Enterprise Information Systems (ICEIS)*, 295–300.
34. B. -U. Pagel, H. -W. Six, and H. Toben (1993). The Transformation Technique for Spatial Objects Revisited. *Proc. Int'l. Symp. on Advances in Spatial Databases (SSD)*, 73–88.
35. J. M. Patel and D. J. DeWitt (1996). Partition Based Spatial-Merge Join. *Proc. Int'l. Conf. on Management of Data (ACM SIGMOD)*, 259–270.
36. D. Pfoser, Y. Theodoridis, and C. S. Jensen (1999). Indexing Trajectories in Query Processing for Moving Objects. *Chorochronos Technical Report*, CH-99-3, 1999.
37. D. Pfoser, C. S. Jensen, and Y. Theodoridis (2000). Novel Approaches in Query Processing for Moving Objects. *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, 395–406.
38. J. T. Robinson (1981). The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes. *Proc. Int'l. Conf. on Management of Data (ACM SIGMOD)*, 10–18.
39. http://www.rtreeportal.org
40. B. Seeger and H.-P. Kriegel (1988). Techniques for Design and Implementation of Efficient Spatial Access Methods. *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, 360–371.
41. T. Sellis, N. Roussopoulos, and C. Faloutsos (1987). The R^+tree: A Dynamic Index for Multidimensional Objects. *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, 507–518.
42. J. Song, K. Whang, Y. Lee, M. Lee, and S. Kim (1999). Spatial Join Processing Using Corner Transformation. *IEEE Trans. on Knowledge and Data Engineering*, 11(4), 688–695.
43. Y. Tao and D. Papadias (2001). MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. *Proc. Int'l. Conf. on Very Large Data Bases (VLDB)*, 431–440.
44. Y. Theodoridis, M. Vazirgiannis, and T. K. Sellis (1996). Spatio-Temporal Indexing for Large Multimedia Applications. *Proc. Int'l. Conf. on Multimedia Systems (IEEE ICMCS)*, 441–448.
45. K. P. Udagepola, Z. Decheng, W. Zhibo, and Y. Xiaozong (2007). Semantic Spatial Objects Data Structure for Spatial Access Method. *Int'l. Journal of Applied Mathematics and Computer Science*, 4(1), 43–50.
46. G. Varga-Solar, J. -L. Zechinelli-Martini, and V. Cuevas-Vicenttin (2008). Integrating and Querying Astronomical Data on the e-GrOV Data Grid. *Int'l. Journal of Computer Systems Science and Engineering (CSSE)*, 23(2), 107–120.
47. S. Y. Wang and C. L. Chou (2008). On the Characteristics of Information Dissemination Paths in Vehicular Ad Hoc Networks on the Move. *Int'l. Journal of Computer Systems Science and Engineering (CSSE)*, 23(6), 403–414.
48. M. L. Yiu, Y. Tao, and N. Mamoulis (2008). The Bdual-Tree: indexing moving objects by space filling curves in the dual space. *VLDB Journal*, 17(3), 379–400.
49. R. Zhang, P. Kalnis, B. C. Ooi, and K. -L. Tan (2005). Generalized Multidimensional Data Mapping and Query Processing. *ACM Transactions on Database Systems*, 30(3), 661–697.