Information Sciences 180 (2010) 1292–1312

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Efficient processing of spatial joins with DOT-based indexing

Hyun Back [a], Jung-Im Won [b,*], Jee-Hee Yoon [a], Sanghyun Park [c], Sang-Wook Kim [d]

[a] Division of Information Engineering and Telecommunications, Hallym University, Republic of Korea
[b] College of Information and Communications, Hanyang University, 17 Haengdang-dong, Seongdong-gu, Seoul 133-791, Republic of Korea
[c] Department of Computer Science, Yonsei University, Republic of Korea
[d] College of Information and Communications, Hanyang University, Republic of Korea

ARTICLE INFO

Article history:
Received 6 August 2008
Received in revised form 4 August 2009
Accepted 6 November 2009

Keywords:
Spatial databases
Spatial indexing
DOT index
Spatial join
Space-filling curve

ABSTRACT

A *spatial join* is a query that searches for a set of object pairs satisfying a given spatial relationship from a database. It is one of the most costly queries, and thus requires an efficient processing algorithm that fully exploits the features of the underlying spatial indexes. In our earlier work, we devised a fairly effective algorithm for processing spatial joins with double transformation (DOT) indexing, which is one of several spatial indexing schemes. However, the algorithm is restricted to only the one-dimensional cases. In this paper, we extend the algorithm for the two-dimensional cases, which are general in Geographic Information Systems (GIS) applications. We first extend DOT to two-dimensional original space. Next, we propose an efficient algorithm for processing range queries using extended DOT. This algorithm employs the *quarter division technique* and the *tri-quarter division technique* devised by analyzing the regularity of the space-filling curve used in DOT. This greatly reduces the number of space transformation operations. We then propose a novel spatial join algorithm based on this range query processing algorithm. In processing a spatial join, we determine the access order of disk pages so that we can minimize the number of disk accesses. We show the superiority of the proposed method by extensive experiments using data sets of various distributions and sizes. The experimental results reveal that the proposed method improves the performance of spatial join processing up to three times in comparison with the widely-used R-tree-based spatial join method.

1. Introduction

Spatial objects are objects that have locations and sizes within space [15]. *Spatial database systems* provide functionalities for the storage and management of a large number of spatial objects and support applications such as Geographic Information Systems (GIS), Very Large Scale Integration (VLSI), and Computer Aided Design/Computer Aided Manufacturing (CAD/CAM) [15,9]. A variety of spatial queries such as *spatial point queries*, *spatial range queries*, and *spatial join queries* are frequently used in spatial database systems [26,19,7,23].

For rapid processing of spatial queries, the selection of an efficient indexing method is crucial. An excellent survey on spatial indexing methods can be 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 [14,18], R^*-trees [2], R^+-trees [33], and Cell-trees [12] are typical examples. We refer to this class of methods as *original space indexing methods* (OS-IMs) in this paper.

* Corresponding author. Tel./fax: +82 2 2220 4567.
E-mail addresses: backhyun@sysgate.co.kr (H. Back), jiwon@hanyang.ac.kr (J.-I. Won), jhyoon@hallym.ac.kr (J.-H. Yoon), sanghyun@cs.yonsei.ac.kr (S. Park), wook@hanyang.ac.kr (S.-W. Kim).

0020-0255/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2009.11.029

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 [24], KDB-trees [30], and LSD-trees [16]. For the transformation, *corner transformation* and *center transformation* are widely used [27,32]. We call members of this second class of indexing methods *high-dimensional space transformation indexing methods* (HDST-IMs).

The third is a class of methods that transform objects in original D-dimensional space into objects in one-dimensional space and then maintain them using the B-tree [6], which is a traditional one-dimensional index structure. Typical examples are proposed in [25,26,11]. We refer to these methods as *low-dimensional space transformation indexing methods* (LDST-IMs).

Of the methods in these three categories, OS-IMs have been the most widely used. Recently, OS-IMs have been extended to index a large number of *moving objects*. 3DR-trees [36], HR-trees [36], STR-trees [28], TB-trees [29], and MV3R-trees [35] are typical examples.

A *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. A spatial join normally proceeds in two steps: a *filtering step* and a *refinement step* [4]. The *filtering step* is performed with minimum bounding rectangles (MBRs), which are simplified versions of the spatial objects, and finds candidate object pairs of which only some can be contained in the final result set. The *refinement step* produces the final result set from those candidate pairs using geometrical computations. The spatial join accesses all the spatial objects repeatedly, and so it requires a high cost in terms of disk access and CPU processing [4,13,37]. Therefore, spatial join algorithms should be devised with careful consideration of the characteristics of the underlying spatial index structures [34].

Several spatial join algorithms have been proposed based on the three classes of spatial indexing methods mentioned above. References [4,5,17] proposed spatial join algorithms based on the R-tree, the most widely used OS-IM. Reference [34] proposed an algorithm that uses HDST-IM. Reference [21] proposed an algorithm that employs the R-tree, but performs joins by reference to the transformed space. Reference [25] proposed an algorithm that uses LDST-IM.

Double transformation (DOT) [10,11], a form of LDST-IM, transforms every MBR in D-dimensional original space into a value in one-dimensional space using a *space-filling curve*, and then stores all those values using the well-known B^+-tree structure. Because the B^+-tree is widely used as a basic index structure in DBMSs, DOT-based query processing algorithms can easily be integrated. In other LDST-IMs [25,26], an object in D-dimensional space is directly transformed into objects in one-dimensional space by using a space-filling curve. The problem with this approach, however, is the proliferation of objects to store because each object is divided into many pieces in one-dimensional space [10,11]. By means of the first transformation that maps an object in D-dimensional space into a single point in 2D-dimensional space, DOT can avoid this problem. In Ref. [11], DOT-based algorithms for processing spatial point queries and spatial range queries are proposed. To the best of our knowledge, however, there are no DOT-based algorithms for processing spatial joins.

In this paper, we address efficient join processing using DOT indexing and show its superiority via a performance evaluation. Reference [1] proposed an initial version of a DOT-based join method that is restricted to the one-dimensional case. This paper deals with various issues that arise when extending the method proposed in Ref. [1] to the two-dimensional case. We first extend DOT, which was proposed for one-dimensional original space [10,11], to two-dimensional original space. Next, we discuss how to represent objects in the four-dimensional space obtained from the first transformation of DOT. Then, we propose an efficient algorithm to process range queries based on extended DOT. The proposed algorithm employs the *quarter division technique* and the *tri-quarter division technique* which were devised by analyzing the regularity of the space-filling curve used in DOT, thereby reducing the number of space transformation operations, which are a performance bottleneck. We then propose a spatial join algorithm based on this range query processing algorithm. 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.

We compared the performance of our method with that of the widely-used R-tree-based spatial join method using data sets of various distributions and sizes. The experimental results revealed that the proposed method achieved a spatial join processing performance up to three times better than R-tree-based spatial join method.

The organization of the paper is as follows. Section 2 briefly reviews the DOT indexing method and our previous works with an example. Section 3 discusses a way of extending DOT indexing to two-dimensional original space. Section 4 proposes an algorithm for range query processing using DOT in two-dimensional original space, and discusses a strategy for optimizing space transformation operations using the quarter division technique. Section 5 proposes an efficient spatial join algorithm that uses DOT indexing, and discusses the effects of performance improvement. Section 6 presents the experimental results verifying the performance of the proposed algorithms. Finally, Section 7 summarizes and concludes the paper.

2. Related work

For the paper to be self-contained, we briefly review the DOT indexing method [8,10], and the one-dimensional spatial join method developed in our previous work [1].

DOT transforms an MBR of a spatial object in original space into a point in one-dimensional space using both high-dimensional and low-dimensional transformations. 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 this the *first transformation*. A corner or center transformation [27] can be used for this purpose.
2. A point in $2k$-dimensional space thus obtained is transformed into a point in one-dimensional space. We call this the *second transformation*, and the value in one-dimensional space is called the X value. The space-filling curve, which is used as the second transformation, should preserve the proximity in the original space [20,22,8]. The *Hilbert curve* [20,22], *Peano curve* [22,8], *tri-Hilbert curve* [10], and *tri-Peano curve* [10] 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* [11], respectively.

Fig. 1 shows an example of a transformation process using DOT. In this example, we have used the corner transformation [27] as the first transformation and the *tri-Peano curve* [10] as the space-filling curve for the second transformation. Fig. 1(a) shows two spatial objects, A and B, and a query, q, in one-dimensional original space. Fig. 1(b) shows A, B, and q 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. Therefore, objects A and B are respectively mapped onto two points in intermediate space, $(1, 3)$ and $(5, 6)$. Since the end point is always larger than or equal to the start point, as shown in Fig. 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. A query region in original space is also mapped into a region. If the start and end points of q are q_s and q_e, respectively, the region is represented as satisfying the condition ($X_s \leqslant q_e$ and $X_e \geqslant q_s$) in the intermediate space, which is shown as a shaded region in Fig. 1(b). If we apply the *tri-Peano curve* [10] for the two-dimensional intermediate space, a distinct X value is assigned to each point on the right-angled triangle-shaped region, as shown in Fig. 1(c). Fig. 1(d) shows the objects A and B as well as the query region q mapped into the final one-dimensional transformed space by the second transformation. The spatial objects A and B are mapped into 6 and 31, respectively. The query region, q, is mapped into two sets of points (denoted as Segments 1 and 2 in the figure) whose X values belong to [3,27] and [29,30], respectively.

DOT-based indexing maintains one-dimensional points, which are transformed by DOT using the B^+-tree structure. The processing of a range query using DOT indexing is a matter of finding spatial objects with X values that belong to one of the line segment ranges of the given query in the final transformed space. In the example in Fig. 1, query processing retrieves spatial objects whose X values are within [3,27] or [29,30], thereby producing object A with an X value of 6 as the query result.

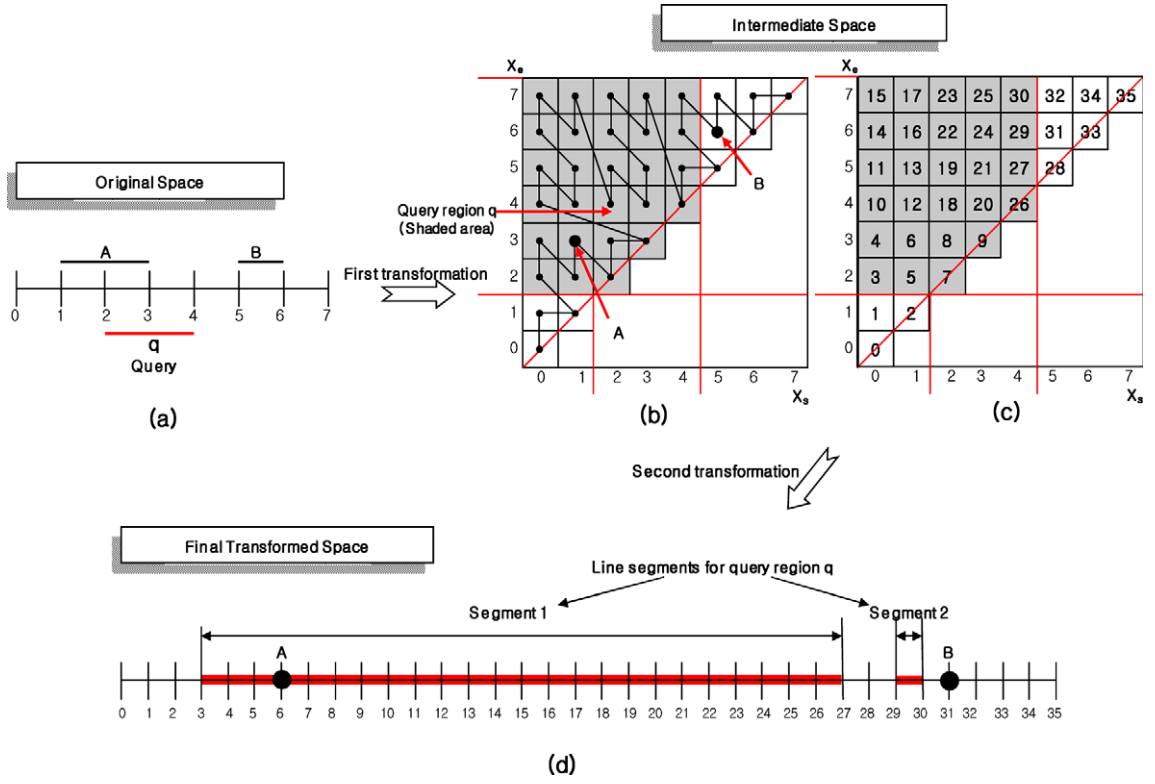


Fig. 1. Examples of a transformation process using DOT.

Spatial join operations can be implemented by the repetitive execution of a range query. As an example, let us consider the spatial join operation between two files R and S, when each of the files has two spatial objects A and B, as shown in Fig. 1(a). The result of this spatial join is obtained by executing two range queries, one with the MBR of object A in file R and another with the MBR of object B in file R, over the DOT index built for file S and then merging the results of the two range queries.

In research preceding this study, we proposed a DOT-based spatial join algorithm for *one-dimensional* spatial objects in [1]. For efficient processing of spatial join operations, the proposed algorithm developed the two optimization techniques shown below:

1. The execution of a range query over a DOT index requires a step transforming the query region in two-dimensional intermediate space into a set of one-dimensional X values. However, as shown in Fig. 1(b), it often occurs that the trajectory of a space-filling curve is not completely consecutive within a query region in two-dimensional intermediate space. Therefore, the X values transformed from a query region in two-dimensional intermediate space tend to be non-contiguous in the one-dimensional final space. This observation implies that the X value of each point contained in a two-dimensional query region must be calculated individually. As a result, as a query region becomes larger, the overhead of obtaining the corresponding X values increases. By analyzing the traversal patterns of a space-filling curve within a query region on the two-dimensional intermediate space, the proposed algorithm made it possible to obtain one-dimensional X values with the smallest computational cost.
2. When executing range queries repetitively for spatial join operations, we are able to reduce the cost for accessing the disk by utilizing the overlaps among query regions. That is, since the DOT index structure preserves the proximity of one-dimensional spatial objects very well, we can reduce the cost for accessing the disk by executing the range queries sequentially according to the order of physical locations of objects stored in the DOT index. In addition, we reduced the number of range queries needed for a spatial join operation by combining the MBRs of several objects close to each other and then using the combined MBR as a representative query region.

Since the aforementioned optimization techniques have been designed and developed for *one-dimensional* spatial objects, they must be revised extensively to be employed in the algorithm for spatially joining *two-dimensional* objects. That is, to develop an efficient algorithm for joining two-dimensional spatial objects, in this study, we propose a visualization technique for four-dimensional objects and analyze the traversal patterns of a space-filling curve within four-dimensional query regions. Using the proposed visualization technique and the analyzed traversal patterns, we substantially extend the prior optimization techniques originally proposed for one-dimensional objects.

3. Extension of DOT

In GIS applications, objects are distributed in two-dimensional rather than in one-dimensional space. References [10,11] discuss DOT only for the one-dimensional case, but do not address its extension to the practical two-dimensional cases used in GIS applications. In this section, we deal with the extension of DOT for two-dimensional cases.

3.1. Representation of objects in intermediate space

Objects in k-dimensional original space are mapped onto points in $2k$-dimensional intermediate space using the first transformation of DOT. For instance, GIS objects in two-dimensional space are transformed into points in four-dimensional intermediate space. However, because space composed of more than three dimensions does not exist in reality, it is difficult to represent an object's distribution within such a space. Nonetheless, such representation is necessary to devise efficient join algorithms by considering object distributions in four-dimensional intermediate space.

Every object in two-dimensional original space is represented as a minimum bounding rectangle. An MBR is represented as two pairs of values: X_s and X_e $(X_s \leqslant X_e)$ from coordinate X, and Y_s and $Y_e (Y_s \leqslant Y_e)$ from coordinate Y. As a result of using the corner transformation as the first transformation, an object is represented as a point in four-dimensional space formed by four dimensions corresponding to X_s, X_e, Y_s, and Y_e. Hereafter, we call these four dimensions: dimension 4 (corresponding to X_s), dimension 3 (corresponding to X_e), dimension 2 (corresponding to Y_s), and dimension 1 (corresponding to Y_e).

The four-dimensional intermediate space can be represented by being projected onto two two-dimensional spaces: one is formed by dimensions 4 and 3, and the other by dimensions 2 and 1. We define the two-dimensional space with dimensions 4 and 3 as the *4–3 plane* or *higher plane space*. Similarly, we define the two-dimensional space with dimensions 2 and 1 as the *2–1 plane* or *lower plane space*. Since $(X_s \leqslant X_e)$ always holds for every object, all the objects in the higher plane space are located only in the right-angled triangular region above the diagonal line. In the same way, all the objects in the lower plane space are located only in the right-angled triangular region above the diagonal line because $(Y_s \leqslant Y_e)$ always holds for every object.

Fig. 2 shows the MBRs a, b, and c for three objects in two-dimensional original space. In the examples used in this paper, we assume the resolution of the original space to be (4×4). The MBRs a, b, and c are mapped by the first transformation into the three points $(0,1,2,3)$, $(2,2,1,1)$, and $(3,3,3,3)$ in four-dimensional intermediate space. Fig. 3 represents the points pro-

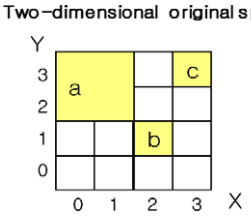


Fig. 2. Three objects in two-dimensional original space.

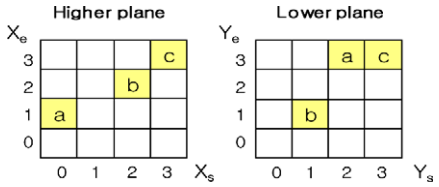


Fig. 3. Three objects of Fig. 2 projected into the higher and lower plane spaces.

jected into the higher and lower plane spaces. In this representation, however, an object appears simultaneously in both the higher and lower plane spaces. These simultaneous appearances make it difficult to understand the characteristics of an object's distribution in four-dimensional intermediate space.

To solve this problem, we represent the higher and lower plane spaces together using a single two-dimensional plane. This method causes the entire lower plane space to be contained in every cell of the higher plane space. With this method, the object distribution in Fig. 2 becomes that shown in Fig. 4, where the right-angled triangular region below the diagonal in which no objects appear is shaded. With this method, we can represent the four-dimensional intermediate space in the two-dimensional plane, and thus examine the characteristics of an object's distribution in four-dimensional intermediate space.

3.2. Representation of a query region in intermediate space

In this section, we show how we represent a query region in four-dimensional intermediate space.

Suppose that the MBR of a query region is given as (QX_s, QX_e, QY_s, QY_e) in two-dimensional original space. Its corresponding query region in four-dimensional intermediate space is defined as $(X_s \leqslant QX_e$ and $X_e \geqslant QX_s)$ and $(Y_s \leqslant QY_e$ and $Y_e \geqslant QY_s)$.

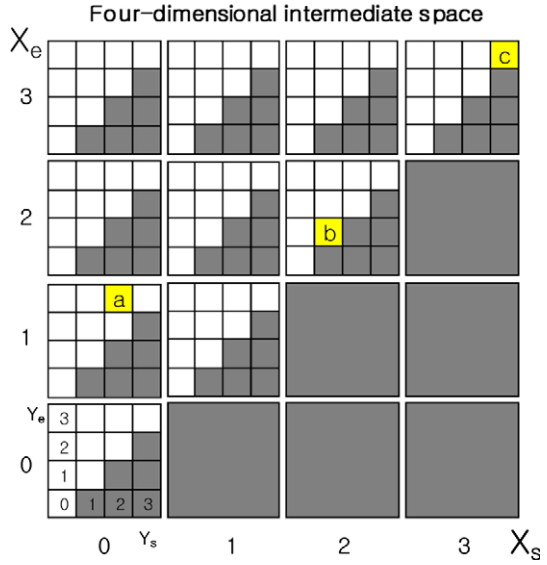


Fig. 4. Three objects of Fig. 2 in four-dimensional intermediate space.

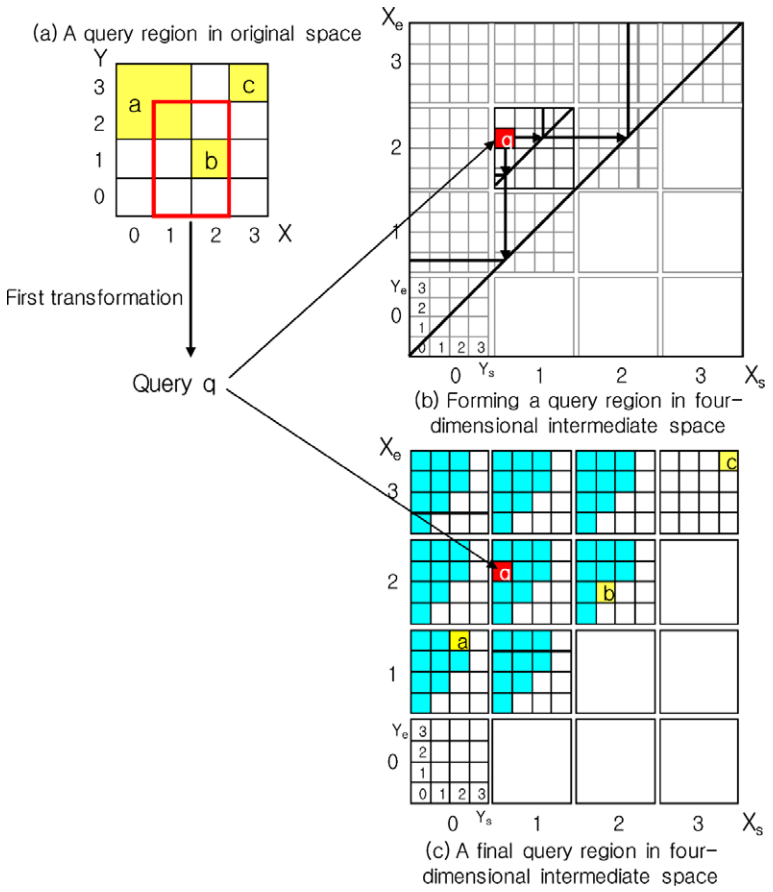


Fig. 5. A query region in four-dimensional intermediate space.

Here, the region of $(X_s > X_e)$ in the higher plane space and the region of $(Y_s > Y_e)$ in the lower plane space, which are below the diagonal line, are not used in query processing.

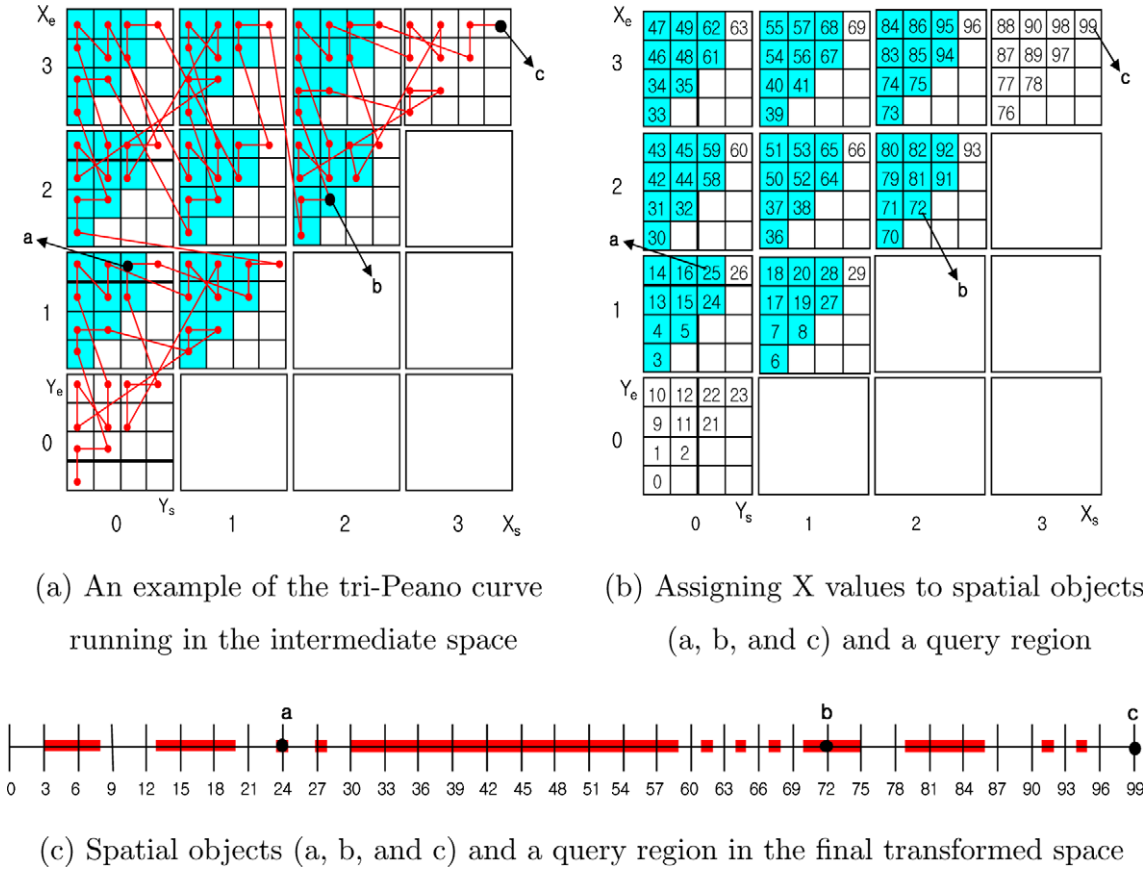
Fig. 5 shows an example of transforming a query region in original space into that in intermediate space. First, the query region within the original space in Fig. 5(a) is mapped onto point q of (1, 2, 0, 2) in intermediate space, as in Fig. 5(b), by the first transformation. Next, a query region in the higher plane of the intermediate space is formed by extending point q to the diagonal line, thereby becoming an upper pentagon. Also, for every lower plane corresponding to cells that belong to a query region within the higher plane, a pentagonal query region is formed in the same way as in the higher plane. The shaded area in Fig. 5(c) shows the final query region in four-dimensional intermediate space using our proposed representation.

3.3. Space-filling curve in four-dimensional intermediate space

For every spatial object, DOT produces an X value in the final one-dimensional space from that in the intermediate space using the second transformation. For this transformation, the method uses the space-filling curve in four-dimensional space.

In this paper, we use a four-dimensional *tri-Peano curve* [10] that runs only on a right triangle above the diagonal in both the higher and lower planes. The *Peano curve* runs on a simple path compared with other space-filling curves. Also, the Peano curve always starts from the lower left quarter of the four quarters, and thus has the advantage of having a path that is easy to predict and analyze in four-dimensional space.

Fig. 6 shows the process of obtaining X values in the final one-dimensional space by applying the tri-Peano curve to objects a, b, c, and a query region in the four-dimensional intermediate space of Fig. 5(c). Fig. 6(a) shows an example of the *tri-Peano curve* running. The tri-Peano curve crosses the higher and lower planes alternately in the four-dimensional intermediate space. As a result, it assigns 25, 72, and 99 to a, b, and c, and similarly assigns corresponding X values to all of the cells in the shaded query region, as in Fig. 6(b). Notice that the trail of the tri-Peano curve is not continuous within the query region. Therefore, the X values for the query region are not consecutive within the query region. Fig. 6(c) shows objects a, b, and c, as well as a query region transformed into the final one-dimensional space by the second transformation.



(a) An example of the tri-Peano curve running in the intermediate space

(b) Assigning X values to spatial objects (a, b, and c) and a query region

(c) Spatial objects (a, b, and c) and a query region in the final transformed space

Fig. 6. Object distribution in the final one-dimensional space.

A query region in the figure is transformed into a set of 12 line segments, each of which consists of cells consecutive in the order of the tri-Peano curve. Here, we observe that object *a* belongs to the range represented by line segment 3, object *b* belongs to the range represented by line segment 9, and object *c* does not belong to any line segment.

4. Range query processing algorithm

In this section, we propose an efficient range query processing algorithm with DOT-based indexing for two-dimensional spatial objects.

4.1. Naive algorithm

The range query is an operation that finds every spatial object overlapping a given query region. The algorithm Range-Query, shown in Algorithm 1, is a range query processing algorithm for two-dimensional spatial objects. It takes a query region (i.e. the MBR of a query) and the grid size of the intermediate space as input and returns the spatial objects overlapping the query region as output.

Algorithm 1. Range-Query (range query algorithm for two-dimensional objects)

Input: X axis Query Start QX_s, Query End QX_e,
Y axis Query Start QY_s, Query End QY_e,
Grid size n
Output: Set of Results QR
1. LS = QueryRangeToLineSegments($QX_s, QX_e, QY_s, QY_e, n$);
2. QR = GetObjectsUsingBTreeIndex(LS);
3. return QR;

Algorithm 2. QueryRangeToLineSegments (naive query transformation algorithm)

Input: X axis Query Start QX_s, Query End QX_e,
Y axis Query Start QY_s, Query End QY_e,
Grid size n
Output: Set of Line Segments LS
1. $Xvalues$={};
2. **for** *each point* $q(X_s, X_e, Y_s, Y_e)$ *in intermediate space* **do**
3. **if** $(X_s \leqslant QX_e$ and$X_e \geqslant QX_s)$ and$(Y_s \leqslant QY_e$ and $Y_e \geqslant QY_s)$ **then**
4. $X values = X values \bigcup$TriPeano$(X_s, X_e, Y_s, Y_e, n)$;
5. LS=MakeLineSegments($X values$);
6. return LS;

Let us examine the algorithm in detail. It transforms the query range into a set of line segments in the one-dimensional space (line 1) and then, using the DOT index, it retrieves the spatial objects overlapping the query region (line 2). GetObjects-UsingBTreeIndex is a function that performs general index searching with the B-tree. The actual query transformation process is executed in function QueryRangeToLineSegments, shown in Algorithm 2. It transforms the query region in the intermediate space into the corresponding X values (lines 2–4) and then produces a set of continuous line segments by sorting the X values (line 5). Here, TriPeano represents a space transformation function that converts each individual grid cell into the corresponding X value in the final transformation space. Currently, we obtain the tri-Peano value indirectly from the Peano curve, rather than directly from the tri-Peano curve. For this, we pre-compute the tri-Peano values by using the Peano function and store them in main memory for future processing. To the best of our knowledge, there have been no algorithms proposed for computing the tri-Peano and tri-Hilbert values directly in k-dimensional space $(k >= 2)$.

Note that as the size of a query region increases, the performance of the algorithm Range-Query deteriorates rapidly. This is because the algorithm requires a large number of space transformation operations to convert a spacious query region into a large set of one-dimensional X values.

As explained in Section 3.2, a two-dimensional query region is mapped into a four-dimensional query region represented by the two-dimensional query regions in the higher and lower plane spaces. The size of the four-dimensional query region is then estimated as the product of its sizes in the higher and lower planes. However, the trail of the space-filling curve may not be continuous within the given query region. Therefore, QueryRangeToLineSegments has to execute a space transformation operation (i.e. function TriPeano) for each one of the grid cells in the query region of the four-dimensional intermediate space to make a set of line segments in the final transformation space.

Let us assume that a query region in two-dimensional original space is mapped to a grid cell, q, in the four-dimensional intermediate space. Then, we can estimate the size of the query region in the intermediate space using the location of grid cell q. For example, if q is located at the cell $(0, n-1, 0, n-1)$, it is mapped to the upper-left corner of the 2–1 plane corresponding to the upper-left cell of the 4–3 plane, and its query region occupies the entire immediate space. In this case, the number of space transformation operations required to process the range query reaches the square of $(n^2 + n)/2$. Here, n denotes the grid size of the intermediate space, and $(n^2 + n)/2$ represents the size of the upper triangle, including the diagonal line, of the intermediate space. Let Nt denote the number of space transformation operations for a range query. Then, it is obvious that Nt is always between n^2 and $(n^4 + 2n^3 + n^2)/4$ (i.e. $n^2 \leqslant Nt \leqslant (n^4 + 2n^3 + n^2)/4$). As a result, the number of space transformation operations for a range query is expressed asymptotically as $O(n^4)$ in the worst case, which is a primary factor in the deterioration of the performance of the algorithm Range-Query.

4.2. Proposed range query algorithm

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 an execution of a space transformation operation is needed only for each subregion, not for each grid cell in 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 each quarter to check that it is contained completely in the query region. If so, a space transformation operation is applied to the quarter only once to extract the corresponding line segment in the final transformation space.

To illustrate the quarter division technique, we assume that the *Peano curve* [22,8] is used as a space-filling curve. Fig. 7 shows the trails of the Peano curves of order 1, 2, 3, and 4 in the two-dimensional space. The trail of the Peano curve is continuous within the intermediate space of $n \times n$ grids. Also, as the intermediate space of $n \times n$ grids is divided into four quarters of $n/2 \times n/2$ grids, the trail of the Peano curve remains continuous within each quarter. Let *firstXvalue* denote

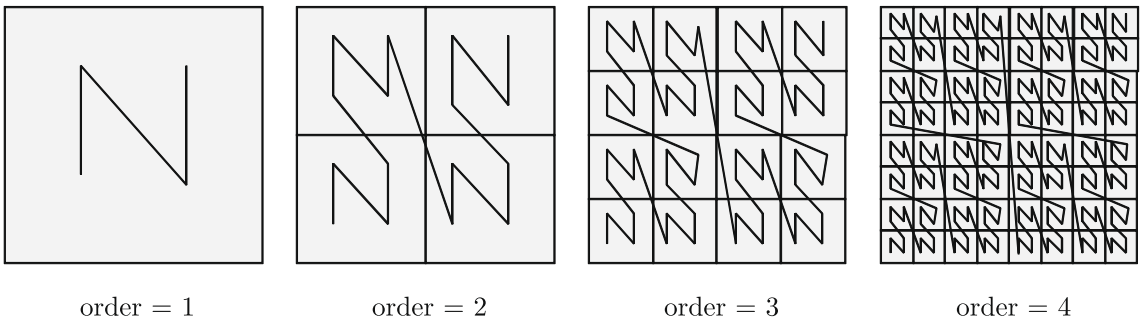


Fig. 7. Peano curves of order 1, 2, 3, and 4 in two-dimensional space.

the X value of the grid cell at which the Peano curve begins within each quarter. Then, for a given quarter, its range of continuous X values in the final transformation space is easily expressed as $[firstX\,value, firstX\,value + n^2 - 1]$.

The four-dimensional Peano curve is extended from the two-dimensional Peano curve. The trail of the Peano curve can be summarized as follows: the four-dimensional Peano curve alternately crosses the higher and lower planes, within each of which it retains the approximation of the two-dimensional Peano curve. It first moves on a minimum quarter of 2×2 grids in the lower plane which is itself contained in a cell of the higher plane. Then, it goes to the next cell in the higher plane and moves on another minimum quarter of 2×2 grids in the lower plane that is contained in a cell of the higher plane, recursively.

As explained previously, a four-dimensional tri-Peano curve runs on a right triangle above the diagonal in both the higher and lower planes. Fig. 8(a) illustrates a trail of the tri-Peano curve in four-dimensional intermediate space. The visiting order of the tri-Peano curve is shown in Fig. 8(b). If we analyze these trails, we can see that the trail of the tri-Peano curve is continuous within a four-dimensional quarter, each dimension of which has the same size. Hereafter, for simplicity, we call such a four-dimensional quarter a '*4D-Quarter*.' The four-dimensional and two-dimensional space-filling curves have in common the property that the trail of the space-filling curve is continuous within a quarter, each dimension of which has the same size.

By using the quarter division technique, the *4D-Quarters* can be found as follows. First, the quarter division technique recursively divides the 4–3 plane into four subregions (i.e. 4–3 plane quarters), and examines each quarter to check that

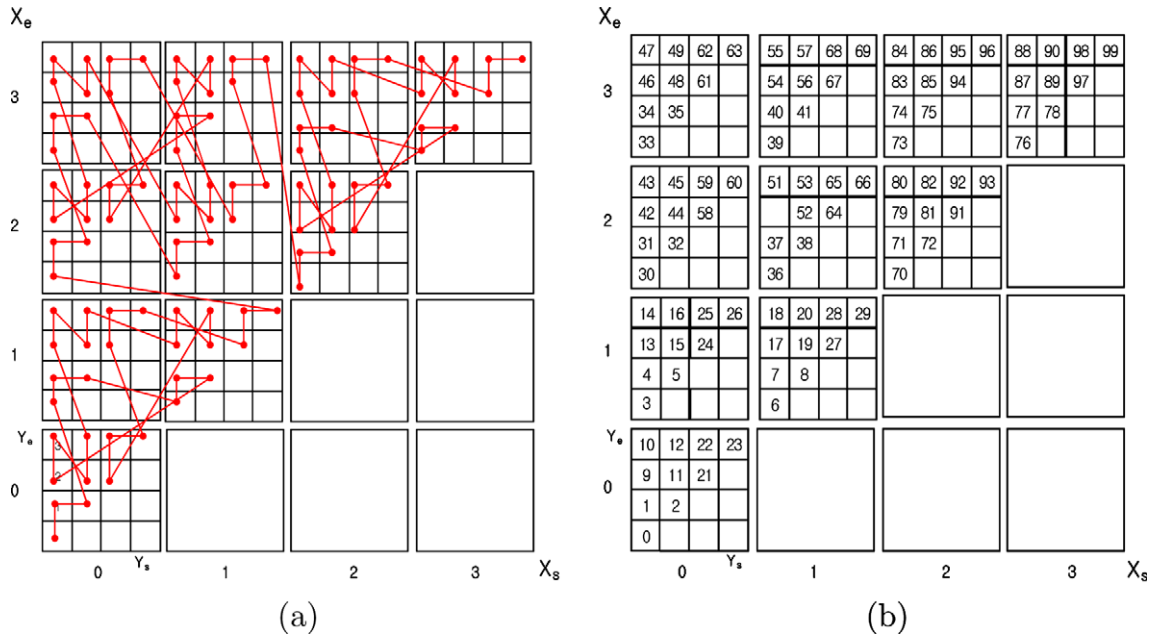


Fig. 8. A trail of the tri-Peano curve in four-dimensional space (intermediate space of $4 \times 4 \times 4 \times 4$ grids).

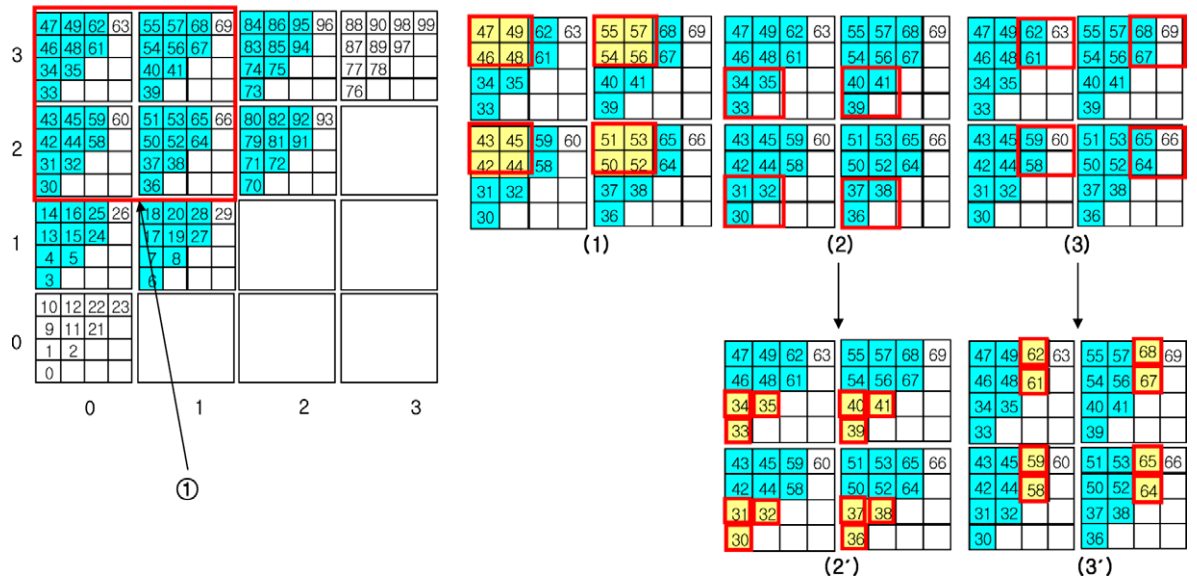


Fig. 9. An example of quarter division in four-dimensional space (intermediate space of $4 \times 4 \times 4 \times 4$ grids).

it is contained completely in the query region. If so, it then recursively divides the 2–1 plane contained in each cell of the 4–3 plane into four subregions (i.e. 2–1 plane quarters), and searches for the *4D-Quarters* which are contained completely in the query region.

Fig. 9 shows an example of the quarter division in the four-dimensional intermediate space. The shaded area shows the query region, and each number in the cell represent the traversing order of the tri-Peano curve. Here, we only show the quarter division process for quarter region ①. Since the sizes of the 4–3 and 2–1 planes are 2×2 and 4×4, respectively, the trail of the space-filling curve is not continuous within quarter region ①. Therefore, the 2–1 plane is divided into four subregions, and the *4D-Quarters* of $2 \times 2 \times 2 \times 2$ are obtained. The bold-lined quarters in Fig. 9(1), (2), and (3) represent the first, second, and third *4D-Quarters*, respectively. Here, the fourth quarter is omitted, as it is not contained in the query region at all. First, since the *4D-Quarters* in Fig. 9(1) are completely contained in the query region, a continuous line segment of [42,57] is obtained by a single space transformation operation. However, since the *4D-Quarters* in Fig. 9(2) and (3) are not completely contained in the query region, they have to be divided into smaller quarters. As a result, the quarters are divided into 12 and 8 quarters of $1 \times 1 \times 1 \times 1$, which are shown in Fig. 9(2′) and (3′), respectively.

Algorithm 3 shows a new query transformation algorithm QueryRangeToLineSegments. The algorithm calls function SplitQueryRangeBy4DQuarters to obtain a set of line segments using the proposed quarter division technique (line 3).

Let us now look into the function SplitQueryRangeBy4DQuarters described in Algorithm 4. Here, $Q43$ and $Q21$ denote quarters in the 4–3 and 2–1 planes, respectively. If $Q43$ is completely contained in the query region, the algorithm calls function 4DQuarterToLineSegment, which returns a set of line segments for this quarter (lines 1–3). Otherwise, it divides the quarter into four sub-quarters and calls function SplitQueryRangeBy4DQuarters for each sub-quarter recursively (lines 4-6). The actual quarter division process is executed in function 4DQuarterToLineSegment. This function divides the current $Q43$ and $Q21$ quarters into *4D-Quarters*, each dimension of which has the same size, and then for each *4D-Quarter*, it calls the space transformation function (i.e. TriPeano) to extract the corresponding line segments.

Algorithm 3. QueryRangeToLineSegments (new query transformation algorithm)

Input: X axis Query Start QX_s, Query End QX_e,
Y axis Query Start QY_s, Query End QY_e,
Grid size n
Output: Set of Line Segments LS
1. LS={};
2. $Q43$= InitializeQuarter43(n);
3. SplitQueryRangeBy4DQuarters($QX_s, QX_e, QY_s, QY_e, Q43, LS$);
4. return LS;

Algorithm 4. SplitQueryRangeBy4DQuarters (quarter division algorithm)

Input: X axis Query Start QX_s, Query End QX_e,
Y axis Query Start QY_s, Query End QY_e,
4–3Plane Quarter $Q43$, Set of Line Segments LS
Output: Set of Line Segments LS

```
1. if IsInQueryRegion(Q43) then
2.     Q21 = InitializeQuarter21(n);
3.     LS = 4DQuarterToLineSegment(Q43, Q21);
4. else
       if Q43.size > 1 then
5.         for sq = 1; sq <= 4; sq++ do
6.             SplitQueryRangeBy4DQuarter (QX_s, QX_e, QY_s, QY_e, GetSubQuarter(Q43, sq), LS);
7. return LS;
```

The number of quarters required to process the range query is the primary factor that affects the algorithm's performance. Note that in two-dimensional intermediate space, the minimum and maximum numbers of quarters to be divided are n and $2n-1$, respectively, depending on the size and location of the query region [1]. In the case of four-dimensional intermediate space, the minimum and maximum numbers of the quarters are n^2 and $(4n^2-4n+1)$, respectively, each of which is calculated by the product of the number of quarters in the 4–3 plane and that in the 2–1 plane. As mentioned before, Nt denotes the number of space transformation operations for a range query. It is obvious, then, that when using algorithm QueryRangeToLineSegments in Algorithm 3, Nt is between n^2 and $(4n^2-4n+1)$ (i.e. $n^2 \leqslant Nt \leqslant 4n^2-4n+1$). Therefore, the algorithm has a complexity of $O(n^2)$ in the worst case, and a significant improvement in performance can thus be expected.

4.3. Further performance improvement

As explained previously, we assume that all the spatial objects and query regions are located within the upper triangle above the diagonal in both the higher and lower planes. If we employ a space-filling curve (i.e. tri-Peano curve) that traverses only the upper triangular regions in both planes, a right-angled triangle which has its oblique side as part of the diagonal line can be a region where the trail of the space-filling curve is continuous. Here, we call such a 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 near the diagonal line into quarters which are too small, and further reduces the cost of space transformations.

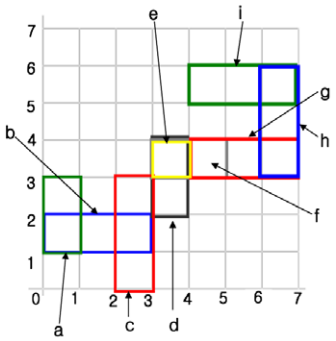
Let us continue the example of Fig. 9 to illustrate a *tri-quarter division technique* for processing range queries. Note that each bold-lined quarter in Fig. 9(2) corresponds to a tri-quarter with an oblique side which is part of the diagonal line of the 2–1 plane, and is completely contained in the query region. Therefore, a continuous line segment of $[30, 41]$ is obtained by only a single space transformation operation. That is, a total of 10 space transformation operations is needed for the quarter region of ① in the previous example, and using the proposed *tri-quarter division technique* reduces the total necessary transformation operations by 11.

The function SplitQueryRangeBy4DQuarters shown in Algorithm 4 needs to be modified when considering the tri-quarter regions in the four-dimensional intermediate space. This function checks whether the current quarters (i.e. $Q43$ and $Q21$) are tri-quarters. In addition, a new formula is inserted into the function 4DQuarterToLineSegment to calculate a continuous line segment for a tri-quarter.

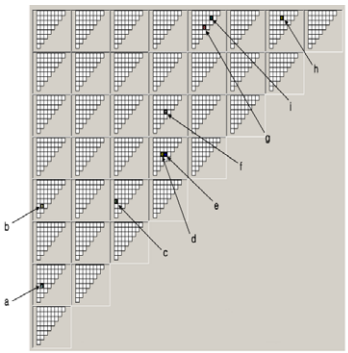
5. Spatial join algorithm

In this section, we propose an efficient DOT-based spatial join algorithm for two-dimensional spatial objects. Our spatial join algorithm for two-dimensional spatial objects is similar to that for one-dimensional spatial objects. Let us suppose we have two files, R and S, equipped with DOT indices for two-dimensional spatial objects. A simple method to obtain the result of their spatial join is to repeatedly execute the range query algorithm explained in Section 4. That is, we can obtain the result of the spatial join between files R and S by, for each two-dimensional spatial object of file R, first establishing a query region from its MBR and then executing a range query to discover all the two-dimensional spatial objects of file S that overlap the query region.

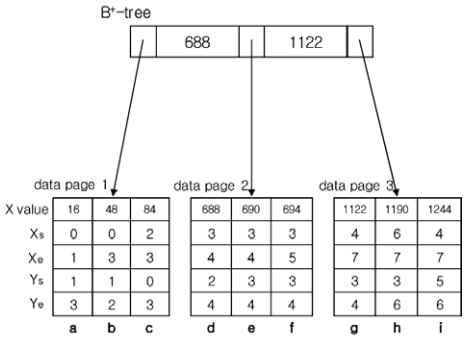
Note that if objects are close to each other in the original two-dimensional space, then they are mapped to the adjacent points in intermediate space via the first transformation. Adjacent points in intermediate space are then mapped to similar X values in final space via the second transformation, which 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. As explained in Section 2 from the X values of final space, we construct a DOT index with a B^+-tree structure for efficient processing of range queries



(a) 9 objects in the original space

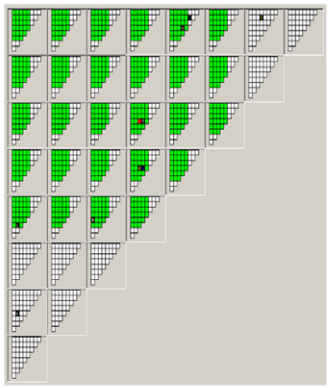


(b) 9 points in the intermediate space

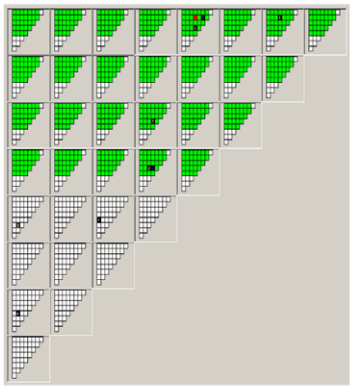


(c) DOT index structure for 9 objects

(1) Query region for leaf page 1



(2) Query region for leaf page 2

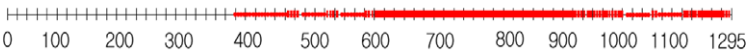


(3) Query region for leaf page 3

(d) The intermediate space containing the query regions for the three leaf pages

(1) Candidate join region for leaf page 1

(2) Candidate join region for leaf page 2



(3) Candidate join region for leaf page 3

(e) The final one-dimensional space containing the candidate join regions for three pages

Fig. 10. Example of two-dimensional spatial join algorithm based on the DOT index.

and spatial joins. Hence, we can expect that adjacent objects in two-dimensional space are likely to be stored on the same leaf page of a DOT index.

We store two-dimensional objects in a DOT index by inserting their X values and MBR information (i.e. $[X_s, X_e, Y_s, Y_e]$) into appropriate leaf pages as shown in Fig. 10(c). Here, we consider the smallest MBR that encloses the MBRs of all objects in a leaf page. That is, for a given leaf page, if we let $minX_s, maxX_e, minY_s$, and $maxY_e$ denote the smallest of the X_s values, the largest of the X_e values, the smallest of the Y_s values, and the largest of the Y_e values, respectively, for all objects in the leaf page, then $[minX_s, maxX_e, minY_s, maxY_e]$ becomes the smallest MBR that encloses the MBRs of all the objects in the leaf page. We call such an MBR an '*LP-MBR*' (Leaf Page MBR) in this paper.

By exploiting the concept of an LP-MBR, we can significantly reduce the number of range queries needed to execute spatial joins. That is, rather than sequentially taking each two-dimensional object in 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 LP-MBR as a query region. Let N be the number of two-dimensional objects of file R, and let M be the average number of two-dimensional objects stored 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 to examine every two-dimensional object pair returned from the proposed spatial join algorithm to verify that they really overlap with each other. However, since the number of two-dimensional object pairs returned from the proposed spatial join algorithm is not large in most cases, this extra step is not costly. Here, we call such a query region of the LP-MBR a *candidate join region*.

Fig. 10 illustrates each step of our two-dimensional spatial join algorithm based on the DOT index. First, the original two-dimensional space in Fig. 10(a) contains 9 objects (i.e., $a, \ldots, i$) of file R with which file S is to be joined spatially. As shown in Fig. 10(b), these 9 objects are then mapped to the four-dimensional points via the first transformation. Fig. 10(c) shows the DOT index structure for file R where the LP-MBRs of its leaf pages 1, 2, and 3 are [0,3,0,3], [3,5,2,4], and [4,7,3,6], respectively. Next, Fig. 10(d) and (e) depict the intermediate space containing the query regions of the three leaf pages and the final one-dimensional space containing the candidate join regions of the three leaf pages, respectively. Lastly, these candidate join regions are applied successively to the DOT index for file S. Note that a large portion of the three candidate join regions are overlapped with each other.

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.

Algorithm 5 shows the proposed DOT-based spatial join algorithm for two-dimensional spatial objects. Here, we assume that files R and S store two-dimensional objects and that DOT indices have been 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 before, to reduce the number of disk accesses to file S. Of the various spatial relationships, we consider only the intersection of spatial objects as their join condition.

Algorithm 5 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), then construct its LP-MBR by assigning to $minX_s, maxX_e, minY_s$, and $maxY_e$ the smallest of the X_s values, the largest of the X_e values, the smallest of the Y_s values, and the largest of the Y_e values for all of the objects in the leaf page (lines 5–8), respectively. Next, we compute the candidate join region of the leaf page by calling function QueryRangeToLineSegments with $minX_s$, $maxX_e, minY_s$, and $maxY_e$ as its parameters (line 9). Here, function QueryRangeToLineSegments converts the query region of the current leaf page of file R to a set of line segments using the quarter division technique described in Algorithm 3. Next, using the DOT index of file S, we retrieve the two-dimensional objects of file S that intersect some of the candidate join regions of the current leaf page of file R (line 11), and examine every pair of two-dimensional objects in the current leaf page of file R and two-dimensional objects retrieved from file S. Only the pairs of two-dimensional objects that intersect each other are included in the final result set (line 12).

Algorithm 5. DOT-based Spatial Join for two-dimensional objects

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.     minY_s=MIN(PageR.Y_s);
8.     maxY_e=MAX(PageR.Y_e);
9.     LS=QueryRangeToLineSegments(minX_s, maxX_e, minY_s, maxY_e, n);
10.    for j = 0; j < |LS|; j++ do
11.        PageS=GetIntersectedLeafPage (DOT(S), LS(j));
12.        RS = RS ⋃ Intersect(PageR, PageS);
13. return RS;
```

6. Performance evaluation

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

6.1. Environment

Two kinds of data sets were used for these experiments: synthetic data sets and real-world road network data sets. We generated synthetic data sets that consisted of spatial objects of various sizes and distributions. Fig. 11 shows three kinds of spatial object distributions over the intermediate space used in our experiments. To show the characteristics of object distribution in four-dimensional intermediate space, we depicted its higher and lower plane spaces using a single two-dimensional plane as proposed in Section 3.1. Spatial objects in the original space with a resolution of (32×32) were mapped onto those in a two-dimensional plane with a resolution of (1024×1024) by the first transformation. Fig. 11(a) shows the uniform distribution, where spatial objects with various sizes in the original space are uniformly distributed above a diagonal line in intermediate space. Fig. 11(b) shows the strip distribution, where objects which are small in the original space are located densely near the diagonal line in intermediate space. Fig. 11(c) shows the strip + uniform distribution, where there are a large number of small spatial objects and a small number of large spatial objects. For our experiments, we produced 100,000 to 1,000,000 spatial objects with these three distributions.

Road network data sets were downloaded from R-tree Portal [31]. Table 1 shows the four road network data sets used in our experiments.

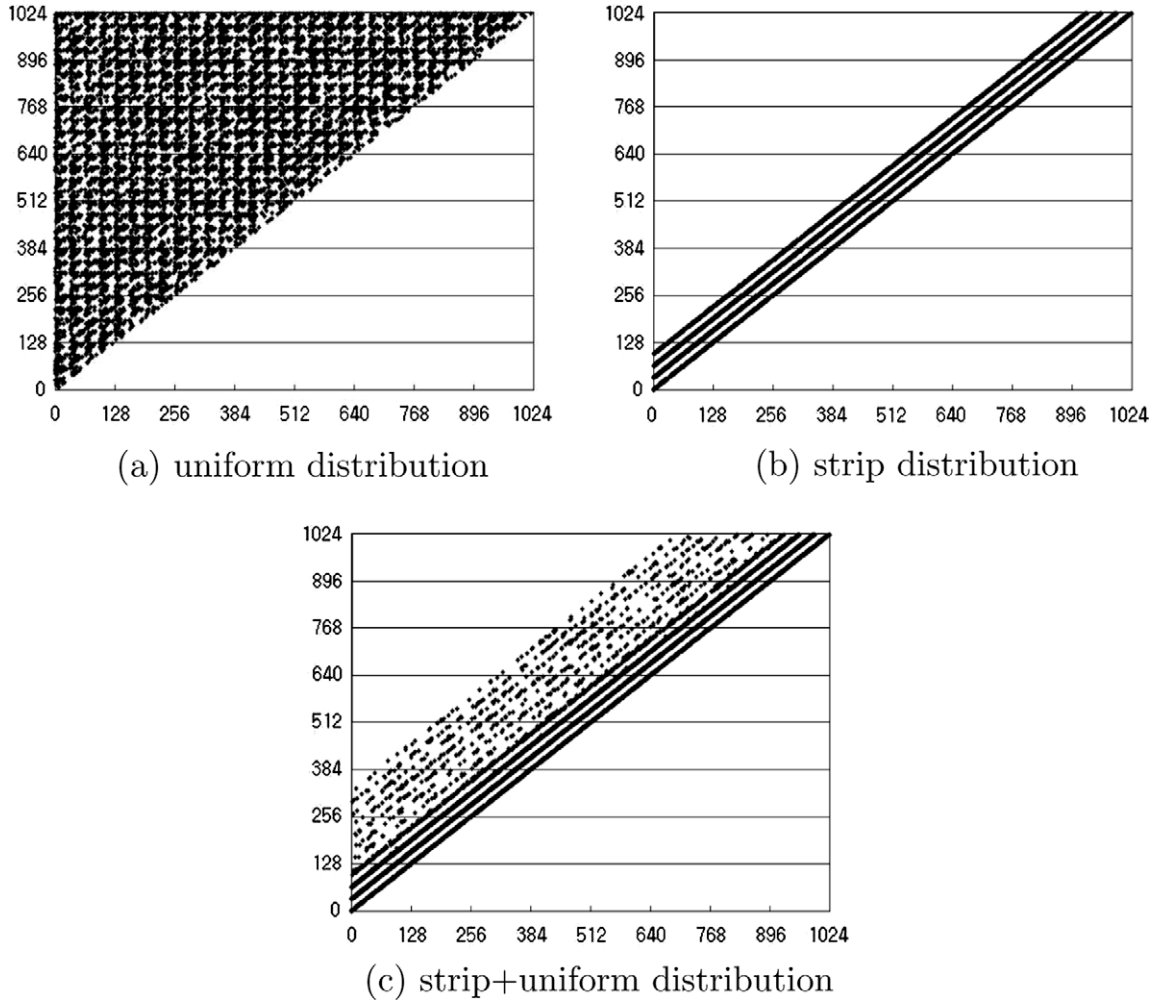


Fig. 11. Three object distributions.

Table 1
Road network data sets used in experiments.

Data set	Data size (Mb)	Number of MBRs	
CAS	0.9	98,451	California streams
RR	2.1	128,971	Tiger/Line LA rivers and railways
TCB	4.3	556,696	Tiger Census blocks
CAR	12.5	2,249,727	California roads

We compared the performance of our spatial join method using the DOT index to that of the R*-tree-based spatial join method. The DOT-TriQuarter and DOT-Quarter methods perform spatial joins with our tri-quarter and quarter division techniques, respectively. These methods make use of the concept of the leaf page MBR (LP-MBR) to significantly reduce the number of range queries needed to execute spatial joins. We set the grid size and the page size to 128 and 4 kB, respectively. We used the *tri-Peano curve* as a space-filling curve for all methods. Our quarter division algorithm recursively divides the intermediate space with two-dimensional planes as a division unit. In a four-dimensional case, our algorithm divides the higher plane first, and then divides the lower plane belonging to every higher plane quarter. Unlike the Peano curve, the Hilbert curve has a characteristic such that it often comes and goes across lower and higher planes in four-dimensional space. This makes the Hilbert curve disagree with our quarter division algorithm. The R*-TreeJoin method is a standard method based on the R*-tree, which was developed by Seeger et al. [31].

Our hardware platform was a Pentium IV 2 GHz PC equipped with 1 GB of main memory and a 120 GB HDD. Our software platform was a Pedoracore 3 Linux system. We measured the number of spatial transformation operations, the number of disk accesses, and the total elapsed time for processing a spatial join. In the experiments, we performed spatial joins on two identical data sets.

6.2. Results and analysis

In Experiment 1, we compared the performance of the DOT-TriQuarter and DOT-Quarter methods in terms of the number of spatial transformation operations. Fig. 12 shows the results obtained 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 to process a join.

The results show that DOT-TriQuarter significantly outperformed DOT-Quarter for the uniform distribution, and performed better than DOT-Quarter for the strip and strip + uniform distributions. This is because a query region is divided into very small quarters and tri-quarters when small spatial objects are located close to a diagonal line, as occurs in the strip and strip + uniform distributions.

Compared with DOT-Quarter, DOT-TriQuarter performed 47.8–97.5 times better over a uniform distribution. Over a strip distribution, DOT-TriQuarter performed 3.7–7.3 times better. Over a strip + uniform distribution, it performed 4.2–5.7 times better.

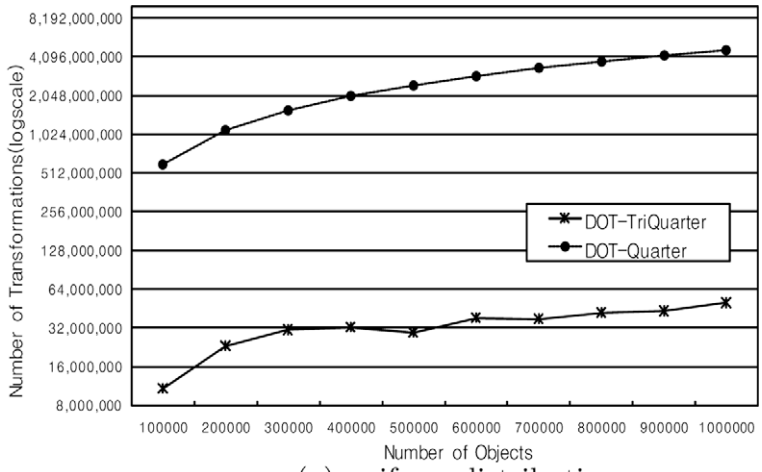
In Experiment 2, we compared the performance of DOT-TriQuarter and R*-TreeJoin in terms of the number of disk accesses. Here, we excluded DOT-Quarter because its required number of disk accesses is the same as DOT-TriQuarter. The data sets used in this experiment contained 1,000,000 spatial objects in each of the three distributions. We also set the storage utilization of leaf pages at around 75% for the two methods. In the uniform, strip, and strip + uniform distributions, the numbers of leaf pages in DOT-TriQuarter were 6535, 6591, and 6650, respectively. The numbers of leaf pages in R*-TreeJoin were 6535, 6597, and 6656 respectively. Fig. 13 shows the results obtained with different buffer sizes. The *X*-axis denotes the buffer size run by the LRU replacement strategy. The *Y*-axis denotes the number of disk accesses required while processing a spatial join.

As shown in Fig. 13, DOT-TriQuarter required a greater number of disk accesses than R*-TreeJoin when the buffer was small. This is because the spatial candidate join regions may not stay in the buffer for continuous range queries despite the fact that the regions to be accessed sequentially have substantial areas of overlap.

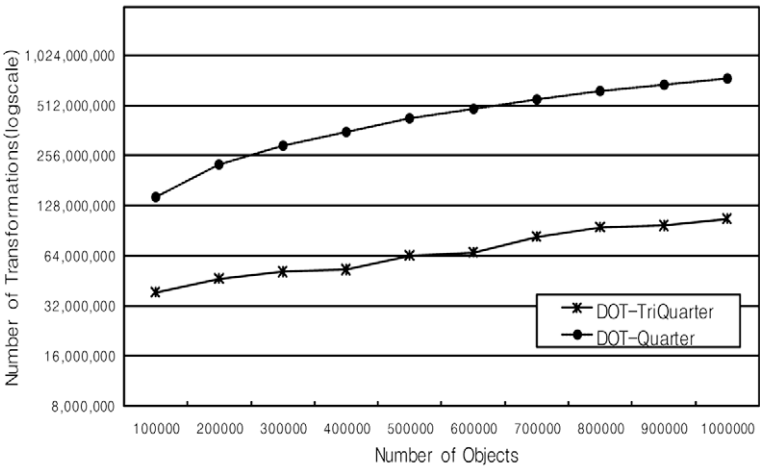
We assume that the optimal spatial join algorithm accesses every data page only once with a buffer of the minimum size. Fig. 13 shows the number of disk accesses when using DOT-TriQuarter and R*-TreeJoin. For the uniform, strip, and strip + uniform distributions, the buffer sizes required by DOT-TriQuarter when the number of disk accesses approached the optimal value were 7168 kB (about 13.7% of the total leaf pages), 4096 kB (about 7.8% of the total leaf pages), and 5120 kB (about 9.6% of the total leaf pages).

In contrast, the number of disk accesses when using R*-TreeJoin was close to the optimal value when the buffer size was 9216 kB (about 17.6% of the total leaf pages) for the uniform distribution, and when the buffer size was 10,240 kB (about 19% of the total leaf pages) for the strip and strip + uniform distributions.

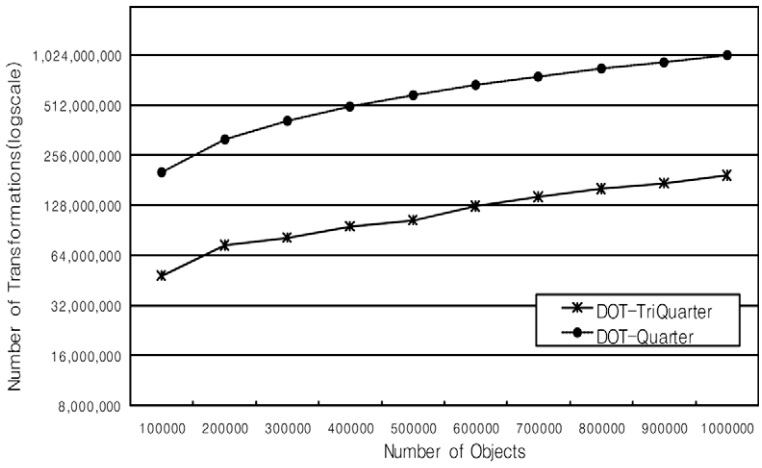
We have compared the performance of the DOT-TriQuarter and R*-TreeJoin methods in terms of the index construction time and the index size. The index construction time for DOT-TriQuarter consists of the transformation time, the sort time, and the insertion time. The transformation time is spent transforming the MBRs of spatial objects in two-dimensional space



(a) uniform distribution

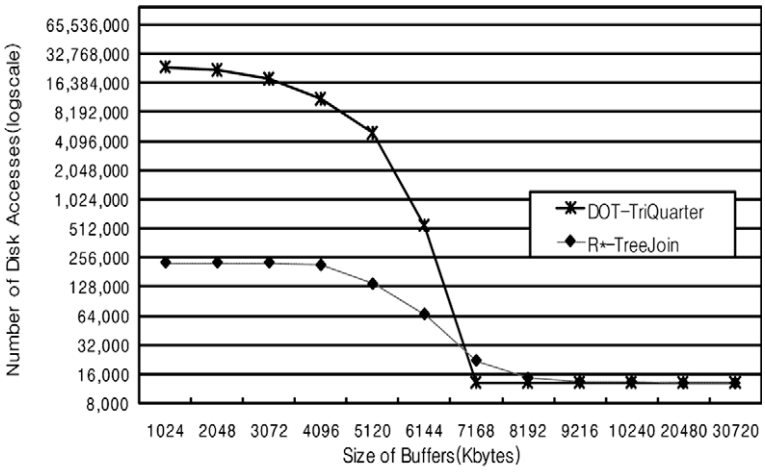


(b) strip distribution

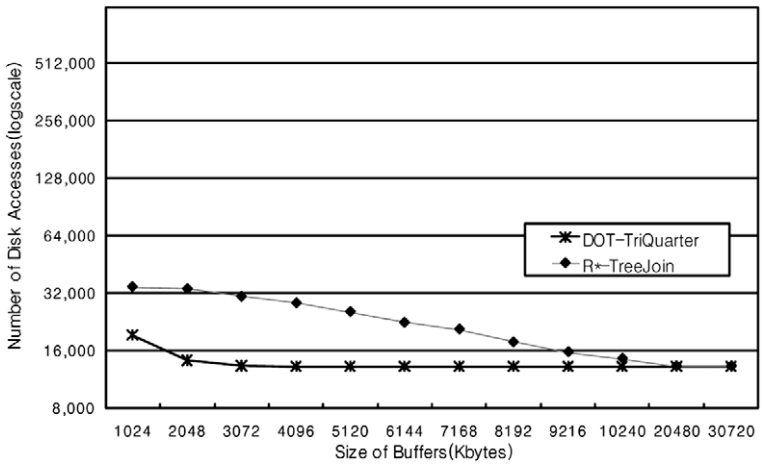


(c) strip+uniform distribution

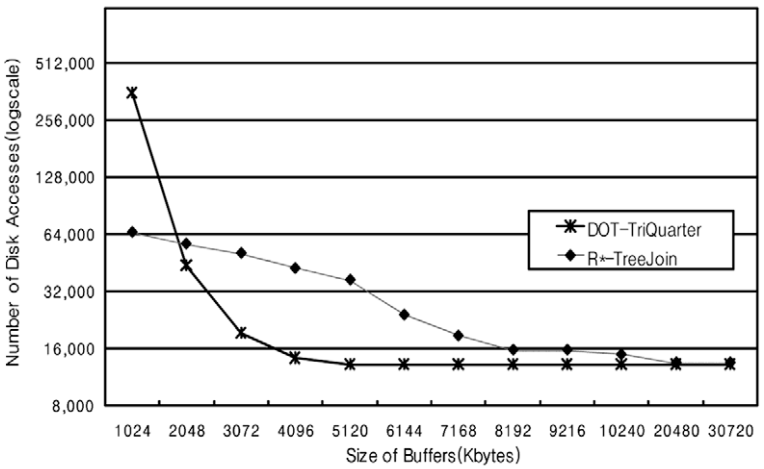
Fig. 12. Number of spatial transformation operations with different numbers of objects.



(a) uniform distribution



(b) strip distribution



(c) strip+uniform distribution

Fig. 13. Number of disk accesses with different buffer sizes.

Table 2
Index size and construction time with various spatial object distributions.

Data Set	DOT-TriQuarter					R*-TreeJoin	
	Index size (Bytes)	Construction time (ms)				Index size (Bytes)	Construction time (ms)
		Sort_Time	Insert_Time	Total time			
Uniform distribution	26,771,456	4945	1517	415	6877	26,771,456	114,874
Strip distribution	27,000,832	2308	1527	377	4212	27,025,408	118,656
Strip + uniform distribution	27,242,496	4622	1497	341	6460	27,267,072	119,489

into X values in one-dimensional space using a space-filling curve. The sort time is spent sorting spatial objects according to their X values. The insertion time is spent inserting spatial objects with X values into the DOT index using the B^+-tree structure.

Table 2 shows the results obtained with various spatial object distributions. As shown in Table 2, DOT-TriQuarter and R*-TreeJoin have almost the same index size for all three distributions. However, in terms of the index construction time, DOT-TriQuarter was 16.7 times faster for the uniform distribution, 28.2 times faster for the strip distribution, and 18.5 times faster for the strip + uniform distribution when compared with R*-TreeJoin.

In Experiment 3, we compared the performance of DOT-TriQuarter, DOT-Quarter, and R*-TreeJoin in terms of the total elapsed time for spatial join processing. The total elapsed time for DOT-TriQuarter and DOT-Quarter consisted of the extraction time, the search time, and the post-processing time. The extraction time is spent obtaining a set of line segments for a query region (spatial candidate join region) by spatial transformation using the quarter division technique. The search time is spent finding spatial objects in a DOT index. The post-processing time is spent verifying that the searched spatial object intersects with a query region.

Fig. 14 shows the results obtained across a varying number of objects. For the uniform, strip, and strip + uniform distributions, buffer sizes were set to 9216 kB, 10,240 kB, and 10,240 kB, respectively, where the numbers of disk accesses required for join processing with DOT-TriQuarter and R*-TreeJoin appeared almost the same in Experiment 2. The Y-axis denotes the elapsed time in msec on a log-scale.

The results show that DOT-TriQuarter performed slightly better than DOT-Quarter for all three distributions. DOT-TriQuarter was 1.2–2.8 times faster than R*-TreeJoin for the uniform distribution. We also see that spatial join processing for the uniform distribution required more processing time than for the strip and strip + uniform distributions. Because spatial objects of various sizes were uniformly distributed over the entire space, all of the methods produced a large result set. Compared with R*-TreeJoin, DOT-TriQuarter was 1.4–3.3 times faster for the strip distribution, and 1.3–2.8 times faster for the strip + uniform distribution.

In Experiment 4, we compared the performance of DOT-TriQuarter and R*-TreeJoin in terms of the index size, the index construction time and the total elapsed time for spatial join processing with real road network data sets. For all four data sets, the buffer sizes were set to 10,240 kB according to the result of Experiment 2.

Table 3 shows the results of comparison of the index size and the index construction time for the four real data sets. We measured the total elapsed time for DOT-TriQuarter using the method explained in Experiment 2. As shown in Table 3, DOT-TriQuarter and R*-TreeJoin have almost the same index size for all four data sets. However, in terms of the index construction time, DOT-TriQuarter was 5.45–13.2 times faster than R*-TreeJoin.

Fig. 15 shows the result of comparison of the spatial join processing time for the four data sets. The Y-axis denotes the elapsed time in msec on a log-scale. The results show that DOT-TriQuarter was 1.5 to 4.8 times faster than R*-TreeJoin for all four data sets.

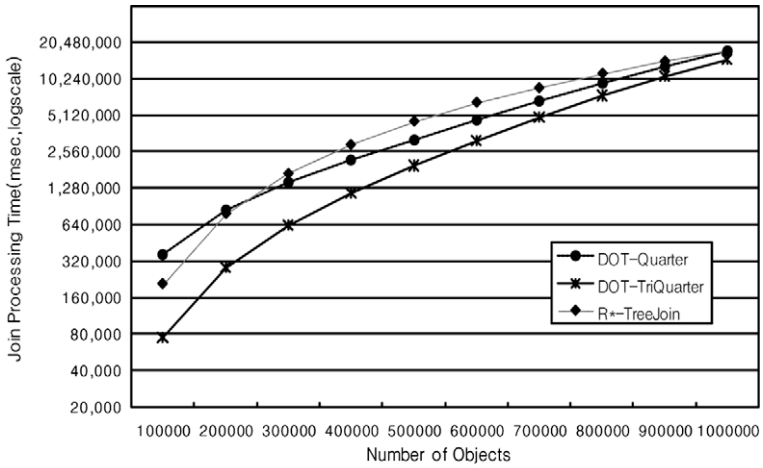
In summary, DOT-TriQuarter showed the best performance regardless of object distribution and was shown to run up to three times faster than R*-TreeJoin.

7. Concluding remarks

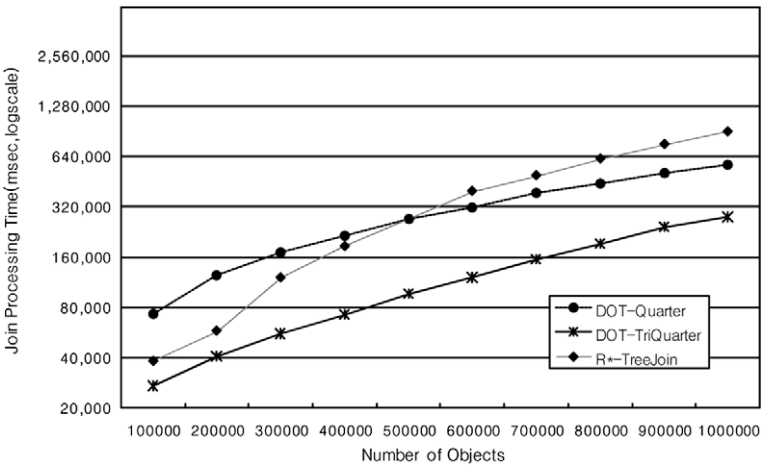
A *spatial join* is an operation that finds a set of pairs of spatial objects satisfying a spatial condition such as overlap or containment. It accesses every spatial object multiple times, and requires a large number of disk accesses as well as high CPU processing times [4,13]. In this paper, we have addressed the efficient processing of spatial joins by using underlying spatial index structures [34].

DOT [10,11] transforms every minimum bounding rectangle in D-dimensional original space into a point in one-dimensional space, and then stores all of those points using a well-known B^+-tree structure. DOT is easily integrated into existing DBMSs because the B^+-tree is a ubiquitous structure. In this paper, we have proposed a spatial join method that exploits DOT-indexing.

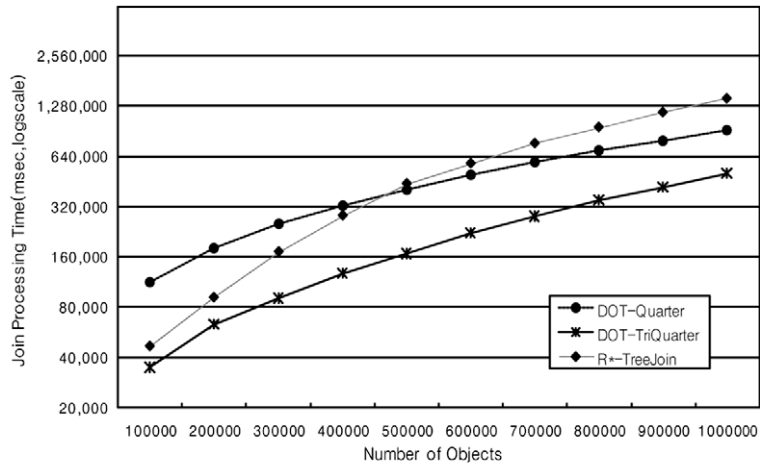
First, we extended the original DOT proposed for one-dimensional original space [10,11] to two-dimensional space. Next, we suggested a way of representing objects in four-dimensional intermediate space obtained from the first transformation of DOT. Then, we analyzed the regularity of the space-filling curve used in DOT and, based on the result of the analysis, we proposed the *quarter division* and *tri-quarter division* techniques to reduce the number of space transformation operations. We



(a) uniform distribution



(b) strip distribution



(c) strip+uniform distribution

Fig. 14. Elapsed time for processing a spatial join with different numbers of objects.

Table 3
Index size and construction time with road network data sets.

Data set	DOT-TriQuarter					R*-TreeJoin	
	Index size (Bytes)	Construction time (ms)				Index size (Bytes)	Construction time (ms)
		Trans_Time	Sort_Time	Insert_Time	Total time		
CAS	4,485,120	225	109	81	415	4,542,464	2263
RR	5,877,760	394	170	85	649	6,320,128	8564
TCB	25,341,952	1130	576	230	1936	26,419,200	11,958
CAR	102,391,808	2952	2587	774	6313	103,161,856	62,985

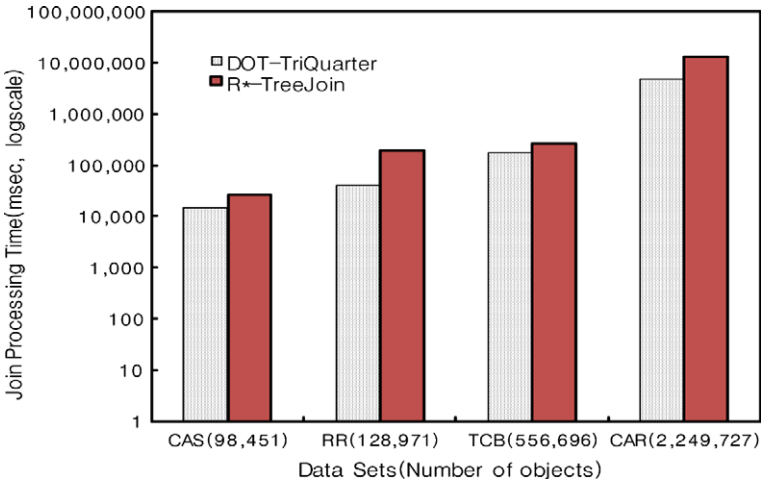


Fig. 15. Elapsed time for processing a spatial join with road network data sets.

then proposed a spatial join algorithm for two-dimensional original space. In processing spatial joins, we determine the access order of the pages containing the spatial objects to considerably reduce the number of disk accesses.

We verified the effectiveness of the proposed method via experiments using data sets of various sizes and distributions. The results show that the proposed method outperforms the standard R*-tree-based join method by up to three times.

Our approach currently targets spatial databases that store geographic objects in two-dimensional original space. As future work, we plan to perform in-depth study on high-dimensional cases to enable the proposed method to be applied in high-dimensional applications such as multimedia information retrieval.

Acknowledgements

This work was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD, Basic Research Promotion Fund) (KRF-2007-314-D00221 and KRF-2007-313-D00651) and also partially supported by the MKE (Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA (Institute of Information Technology Advancement) (IITA-2009-C1090-0902-0040).

References

[1] H. Back, J. -I. Won, J. -H. Yoon, S. Park, An efficient spatial join method using DOT index, Journal of Korean Institute of Information Scientists and Engineers, KISS 34 (5) (2007) 420–436.
[2] N. Beckmann, H. Kriegel, R. Schneider, The R*-tree: an efficient and robust access method for points and rectangles, in: Proc. Int'l. Conf. on Management of Data, ACM SIGMOD, 1990, pp. 322–331.
[3] C. Bohm, S. Beerchtold, D. Keim, Searching in high dimensional spaces – index structures for improving the performance of multimedia databases, ACM Computing Surveys 33 (3) (2001) 322–373.
[4] T. Brinkhoff, H.P. Kriegel, B. Seeger, Efficient processing of spatial joins using R-trees, in: Proc. Int'l. Conf. on Management of Data, ACM SIGMOD, 1993, pp. 237–246.
[5] T. Brinkhoff, H.P. Kriegel, R. Schneider, B. Seeger, Multi-step processing of spatial joins, in: Proc. Int'l. Conf. on Management of Data, ACM SIGMOD, 1994, pp. 197–208.
[6] D. Comer, The ubiquitous B-Tree, ACM Computing Surveys 11 (2) (1979) 121–137.
[7] A. Corral, J.M. Almendros-Jimeneza, A performance comparison of distance-based query algorithms using R-trees in spatial databases, Information Sciences 177 (11) (2007) 2207–2237.
[8] H. Dai, H. Su, Approximation and analytical studies of inter-cluster performances of space-filling curves, Discrete Mathematics and Theoretical Computer Science (2003) 53–68.

[9] S. Du, Q. Qin, Q. Wang, H. Ma, Evaluating structural and topological consistency of complex regions with broad boundaries in multi-resolution spatial databases, Information Sciences 178 (1) (2008) 52–68.
[10] C. Faloutsos, S. Roseman, Fractals for secondary key retrieval, in: Proc. Int'l. Symp. on Principles of Database Systems, ACM PODS, 1989, pp. 247–252.
[11] C. Faloutsos, Y. Rong, DOT: a spatial access method using fractals, in: Proc. Int'l. Conf. on Data Engineering, IEEE ICDE, 1991, pp. 152–159.
[12] O. Gunther, The Cell Tree: An Index for Geometric Data, Memorandum No. UCB/ERL M86/89, Univ. of California, Berkeley, December 1986.
[13] O. Gunther, Efficient computation of spatial joins, in: Proc. Int'l. Conf. on Data Engineering, IEEE ICDE, 1993, pp. 50–59.
[14] A. Guttman, R-trees: a dynamic index structures for spatial searching, Proc. Int'l. Conf. on Management of Data, ACM SIGMOD, 1984, pp. 47–57.
[15] R.H. Guting, An introduction to spatial database systems, Special Issue on Spatial Database Systems of the VLDB Journal 3 (4) (1994) 357–399.
[16] A. Henrich, H.-W. Six, P. Widmayer, The LSD tree: spatial access to multidimensional point and non point objects, in: Proc. Int'l. Conf. on Very Large Data Bases, VLDB, 1989, pp. 45–53.
[17] Y.-W. Huang, N. Jing, E.A. Rundensteiner, Rundensteiner: spatial joins using R-trees: breadth-first traversal with global optimizations, in: Proc. Int'l. Conf. on Very Large Data Bases, VLDB, 1997, pp. 396–405.
[18] P.W. Huang, P.L. Linb, H.Y. Lina, Optimizing storage utilization in R-tree dynamic index structure for spatial databases, Journal of Systems and Software 55 (3) (2001) 291–299.
[19] H. Kriegel, M. Schiwietz, R. Schneider, B. Seeger, Performance comparison of point and spatial access methods, in: Proc. Int'l. Symp. on Design and Implementation of Large Spatial Databases, SSD, 1989, pp. 89–114.
[20] J. Lawder, P. King, Querying multi-dimensional data indexed using the Hilbert space-filling curve, ACM SIGMOD Record 30 (1) (2001) 19–24.
[21] M. Lee, K. Whang, W. Han, I. Song, Transform-space view: Performing spatial join in the transform space using original-space indexes, IEEE Transactions on Knowledge and Data Engineering 18 (2) (2006) 245–260.
[22] B. Moon, H.V. Jagadish, C. Faloutsos, J. Saltz, Analysis of the clustering properties of Hilbert space-filling curve, IEEE Transactions on Knowledge and Data Engineering 13 (1) (2001) 124–141.
[23] H.K. Ng, H.W. Leong, Multi-point queries in large spatial databases, in: Proc. Int'l. Conf. on Advances in Computer Science and Technology, IASTED ACST, 2007, pp. 408–413.
[24] J. Nievergelt, H. Hinterberger, K.C. Sevcik, The grid file: an adaptable, symmetric Multikey file structure, ACM Transactions on Database Systems 9 (1) (1984) 38–71.
[25] J. Orenstein, Spatial query processing in an object-oriented database system, in: Proc. Int'l. Conf. on Management of Data, ACM SIGMOD, 1986, pp. 326–336.
[26] J. Orenstein, F. Manola, PROBE spatial data modeling and query processing in an image database applications, IEEE Transactions on Software Engineering 14 (5) (1988) 611–629.
[27] B.-U. Pagel, H.-W. Six, H. Toben, The transformation technique for spatial objects revisited, in: Proc. Int'l. Symp. on Advances in Spatial Databases, SSD, 1993, pp. 73–88.
[28] D. Pfoser, Y. Theodoridis, C.S. Jensen, Indexing Trajectories in Query Processing for Moving Objects, Chorochronos Technical Report, CH-99-3, 1999.
[29] D. Pfoser, C.S. Jensen, Y. Theodoridis, Novel approaches in query processing for moving objects, in: Proc. Int'l. Conf. on Very Large Data Bases, VLDB, 2000, pp. 395–406.
[30] J.T. Robinson, The K–D–B-tree: a search structure for large multidimensional dynamic indexes, in: Proc. Int'l. Conf. on Management of Data, ACM SIGMOD, 1981, pp. 10–18.
[31] <http://www.rtreeportal.org>.
[32] B. Seeger, H.-P. Kriegel, Techniques for design and implementation of efficient spatial access methods, in: Proc. Int'l. Conf. on Very Large Data Bases, VLDB, 1988, pp. 360–371.
[33] T. Sellis, N. Roussopoulos, C. Faloutsos, The R^+ tree: a dynamic index for multidimensional objects, in: Proc. Int'l. Conf. on Very Large Data Bases, VLDB, 1987, pp. 507–518.
[34] J. Song, K. Whang, Y. Lee, M. Lee, S. Kim, Spatial join processing using corner transformation, IEEE Transactions on Knowledge and Data Engineering 11 (4) (1999) 688–695.
[35] Y. Tao, D. Papadias, MV3R-tree: a spatio-temporal access method for timestamp and interval queries, in: Proc. Int'l. Conf. on Very Large Data Bases, VLDB, 2001, pp. 431–440.
[36] Y. Theodoridis, M. Vazirgiannis, T.K. Sellis, Spatio-temporal indexing for large multimedia applications, in: Proc. Int'l. Conf. on Multimedia Systems, IEEE ICMCS, 1996, pp. 441–448.
[37] J. Xiao, Comparison of heuristics for scheduling spatial clusters to reduce I/O cost in spatial join processing, in: Proc. Int'l. Conf. on Machine Learning and Cybernetics, ICMLC, 2006, pp. 2455–2460.