An NA-Tree-Bit-Patterns-based Method for Continuous Range Queries over Moving Objects[1]

Hue-Ling Chen[†], and Ye-In Chang

[†]Dept. of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan
Republic of China
{E-mail: chen.hueling@gmail.com}
{Tel: 886-3-4244395}
{Fax: 886-3-4245281}

Abstract

A continuous range query is defined to periodically re-evaluated to locate moving objects that are currently inside the boundary of the range query and is widely used to support the location-based services. However, the query processing becomes complicated due to frequent locations update of moving objects. The query indexing relies on incremental evaluation, building the index on range queries instead of moving objects, and exploiting the relation between locations objects and queries. The cell-based query indexing method has been proved to have the better performance of query processing than that of the R^*-tree-based query indexing method with the overlapping problem in internal nodes. However, it takes a lot of space and time for the cell-based method to maintain the index structure, when the number of range queries increases. The NA-tree has been proved to solve the overlapping problem in the R^*-tree to minimize the number of disk accesses during a tree search for the range queries. In this paper, we propose the *NA-Tree-Bit-Pattern-Based* (*NABP*) query indexing method based on the NA-tree. We use the bit-patterns to denote the regions and to preserve the locality of range queries and moving objects. Therefore, our *NABP* method can incrementally local update the affected range queries over moving objects by bit-patterns operations, especially with the increase of the number of range queries. From our simulation study, we show that our *NABP* method requires less CPU time and storage cost than the cell-based method for large number of range queries update. We also show that our *NABP* method requires less CPU time than the R^*-tree-based method for large number of moving objects update.

(**Key Words:** bit-pattern, continuous range queries, moving objects, NA-tree, query index)

[1]This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-95-2221-E-110-101.

1 Introduction

A continuous range query is a fundamental type of a continuous query to periodically re-evaluated to locate moving objects that are currently inside query ranges [10, 11]. For example, "Find all people who are moving close to the department store, and then send e-coupons to the PDAs or cell-phones of potential customers who are interested in the big sale" is one continuous range query. It can be answered by placing a boundary around the location of the building to retrieving all moving objects that are currently located within querying boundaries [7, 21]. Therefore, monitoring continuous range queries is required in numerous applications such as logistics, transportation, and location-based services. Figure 1 shows a typical monitoring system, which consists of a base station, a database server, application servers, and a large number of moving objects. The database server manages range queries and moving objects. The application server requests or updates spatial range queries to the database server, and obtains the latest query results from the database server periodically. Moving objects periodically report their locations by GPS to the database server through the base station. Only when locations of range queries or moving objects update, the query results will be updated and sent to the related range queries and moving objects [7, 21].

[Figure 1 about here.]

Since the range queries and objects move with time, we focus on the problem of range queries over moving objects in which the ranges of queries are changed less frequently than the locations of moving objects. Let us take Figure 2 as an example to briefly describe continuous range queries. A range query is regarded as a Minimal Boundary Rectangle (MBR) and a moving object is regarded as single point. At time T_0 in Figure 2-(a), query Q_1 contains no object and query Q_2 contains object P_2. These queries do not change their locations during time T_0 to T_1. At time T_1 in Figure 2-(b), query Q_1 contains object P_1, because object P_1 changes its locations at time T_0 and moves to query Q_1. At time T_2 in Figure 2-(c), query Q_1 changes its range and does not contain object P_1. Therefore, the result of range queries should be updated continuously when the locations of moving objects or the range of queries are changed with time.

[Figure 2 about here.]

Efficient evaluation of continuous range queries is critical on the acceptable response time to search for range queries over the moving objects. Many methods on indexing for the continuous range queries [7, 9, 10, 11, 15, 17, 21, 22, 24, 26] have been discussed and can be classified into two categories: the object indexing and the query indexing, as shown in Figure 3. However, the brute force method or the object indexing method [2, 12, 16, 18] has poor performance due to frequent objects update. Because locations of range queries change less frequently than those of moving objects [16], it costs less time to maintain an index for range queries than for moving objects. In order to take less time to perform the periodic query evaluation, the query indexing has been proposed to rely on incremental evaluation, reversing the role of queries and data, and exploiting the related locations of objects and queries [7, 13, 14, 17, 22, 23, 24].

[Figure 3 about here.]

As shown in Figure 3, the methods of query indexing can be classified into two categories, tree-based methods and grid based methods, respectively. In the tree-based methods, the R^*-tree-based method [1, 6, 17, 19] (the variation of the R-tree) was first proposed to use a safe region, the shortest range between the object and a query boundary, to avoid excessive location updates. The object does not have to report the location until it moves outside the safe region. However, the performance of the R^*-tree-based method is degenerated by intensive computation to determine a safe region due to the overlapping in regions of range queries or internal nodes. In the grid indexing methods, the space is partitioned into the equal sized grids which are usually regarded as cells. In [10, 11, 22, 23, 24], the Kalashnikov *et al.*'s cell-based method was proposed to have the better performance than the R^*-tree based-method. It uses query lists for each partitioned cell to help answer the continuous range queries. The shingle-based indexing method [22] is proposed to deal with the drawback of accessing unnecessary lists which affects the performance the incremental updating. However, it costs a lot of space and time to build and maintain the index, while the number of range queries increases or changes. It also costs a lot of time to search for all queries in query lists, while objects change locations frequently.

Since the grid-based query indexing method has the better performance of query processing than that of the tree-based query indexing method, we adopt the the Nine-Areas

tree (denoted NA-tree) which is one of grid-based indices as the index structure for range queries. The NA-tree uses the bucket-numbering scheme [5] to partition the space into buckets, where a bucket means the partitioned space equivalent to a grid. The NA-tree is also one of tree-based indices which defines nine regions to store a range query in only one node. Therefore, the NA-tree can solve the problem of overlaps between MBR's of internal nodes in the R*-tree [1, 4, 27] to minimize the number of disk accesses during a tree search for the range queries [5]. In other words, it needs less search cost (in terms of the number of visited nodes) than the R*-tree. Therefore, in this paper, we propose a query indexing method based on the NA-tree, the *NA-Tree-Bit-Pattern-Based* query indexing method (the *NABP* method). We observe that the related bit-patterns denoting for regions based on NA-tree can be used to answer the range queries. Since the bit-patterns are used to preserve the locality of range queries and moving objects, our *NABP* method can incrementally local update the affected range queries over moving objects by bit-patterns operations. Moreover, since the number of range queries increases with time, our *NABP* method can update the bit-patterns only for the increase of the number of range queries, instead of rebuilding the NA-tree. From our simulation study, we show that our *NABP* method requires less CPU time and storage cost than the cell-based method for large number of range queries update. We also show that our *NABP* method requires less CPU time than the R^*-tree-based method for large number of moving objects update.

The rest of the paper is organized as follows. In section 2, we discuss related work and briefly describe the NA-tree structure which is used in our *NABP* method. In Section 3, we present our *NABP* method for continuous range queries processing. In Section 4, we evaluate the performance among our *NABP* method, the Kalashnikov *et al.*'s cell-based method [10, 11], and R^*-tree-based method [1, 4, 27]. Finally, we give the conclusion.

2 Related Work

The problem of evaluation on continuous range queries is: given a set of range queries and a set of moving objects, continuously determine the set of objects that are contained within each query [10, 11]. The goal of the method for the continuous range queries is to re-evaluate all queries in as short time as possible. The storage cost should also be

considered. In this section, we briefly describe two well-known indexing method for the continuous range queries, including the cell-based method and the R^*-tree-based method. We also describe the NA-tree index structure used in our $NABP$ method.

2.1 The Cell-based Method

In the Kalashnikov *et al.*'s cell-based method [10, 11], the space is partitioned into cells. Each cell maintains two query lists: full and partial, as shown in Figure 4. The full list stores the IDs of the queries that completely cover the cell, while the partial list keeps those partially cover the cell. During the query reevaluation, these lists are used to find all the queries that cover an object location. Take Figure 4 as an example. For point P in cell(0,1) in Figure 4-(a), range queries Q_2 and Q_4 in the list are checked. However, the cell-based method requires the large storage to store the duplicate range queries in query lists of the related cells. For example, range queries Q_2 and Q_4 are stored twice in partial list of cell(0,0) and cell(0,1), as shown in Figure 4-(b). When the number of range queries is larger than the capacity of the cell, the monitor area should be re-partitioned into cells of new size. It takes a lot of time to rebuild and maintain query lists of cells.

[Figure 4 about here.]

2.2 The R^*-Tree-based Method

The R^*-tree-based method [1, 6, 17, 19] uses a safe region, the shortest range between the object and a query boundary, to avoid excessive location updates. Take Figure 5 as an example. Since there is the overlapping between internal nodes $E1$ and $E2$ in Figure 5-(a), the R^*-tree-based method should travel one path from node $E1$ to compute the safe region in range query Q_2 for point P. It also travels another path from node E_2 to compute the safe region in query Q_4 for point P, as shown in Figure 5-(b). The safe regions are different and required to be updated while the point P moves away. Due to the overlapping in regions of range queries or internal nodes, the R^*-tree-based method has to travel more than one tree path and access many partitions of nodes to compute the range of a safe region for the moving object.

[Figure 5 about here.]

2.3 The NA-Tree (Nine-Area Tree)

An NA-tree is an index structure based on the bucket-numbering scheme [5]. The space is decomposed into buckets. A bucket is numbered as a binary bit-string $x_1y_1..x_ny_n$ of 0's and 1's, the so-called DZ expression. Symbols '0' and '1' correspond to left (lower) and right (upper) half regions, respectively, for each binary division along the X axis (Y axis). The leftmost two bits x_1y_1 correspond to the first binary division, and the nth two bits x_ny_n correspond to the nth binary division along the X and Y axes of the space made by the (n-1)'th division. Figure 6-(a) and (b) shows examples of buckets after one and two binary divisions along the X and Y axes, respectively. The DZ expression of the gray bucket in Figure 6-(b) is $x_1y_1x_2y_2$='0110*'. We convert the bucket number from binary to decimal form, as shown in the legend alongside Figure 6-(b). The uptrend of bucket numbers increases from southwest to northeast, as shown in Figure 7-(a), which is called the N-order Peano curve of order n(=2). Therefore, the number of buckets after the nth binary division along the X and Y axes are ordered by the N-order Peano curve of order n. A variable, *Max_bucket* ($= 2^n \times 2^n - 1$), is used to record the maximum bucket number (in decimal form) of this area. In Figure 7-(b), the maximum bucket number is 15 (1111), *i.e.*, $Max\_bucket = 15$.

[Figure 6 about here.]

[Figure 7 about here.]

In an NA-tree [5], a spatial object is specified by its bounding rectangle and represented by two points, $L(X_l, Y_b)$ and $U(X_r, Y_t)$, where L is the lower left coordinate and U is the upper right coordinate of the bounding rectangle. The spatial number $O(l, u)$ is computed for the spatial object O, where l is the bucket number of $L(X_l, Y_b)$ and u is the bucket number of $U(X_r, Y_t)$. For example, the spatial number of spatial object O in Figure 7-(b) is (1, 6). Moreover, two bucket numbers l and u can be represented as the binary form: $x_1y_1...x_ny_n$ and $x'_1y'_1x'_ny'_n$, respectively, where $x_i \leq x'_i$, $y_i \leq y'_i$, and $1 \leq i \leq n$.

Based on the bucket-numbering scheme, there are four equal-sized regions defined for four ranges of the bucket numbers, as shown in Figures 8-(a) and (b). According to ranges of two spatial numbers: l and u, for a spatial object O, there are nine child nodes defined for different sized regions in an NA-tree, as shown in Figures 8-(c) and 9. An NA-tree

structure can handle two types of nodes, internal nodes and leaf nodes, for spatial objects. An internal node in the NA-tree can have nine, four, three, or two child nodes. Since a leaf node has no child node, it is a terminal node. A spatial object can only be stored in a leaf, not in an internal node. Because the NA-tree does not split the spatial space, it spatially organizes these spatial data objects depending on their spatial numbers and locations. Take Figure 10 as an example. Our *NABP* method uses the NA-tree to store one range query in one node. Range queries Q_2 and Q_4 are stored once in the $5th$ child node of the NA-tree. The storage cost would be less than the cell-based method. In order to answer the continuous range queries over the moving point P at time t, our *NABP* method accesses in only one path from the root to the $5th$ child node to search for range queries Q_2 and Q_4. The search cost would be less than the R^*-tree-based method.

[Figure 8 about here.]

[Figure 9 about here.]

[Figure 10 about here.]

3 The NA-Tree-Bit-Patterns-based Method (*NABP*)

In this section, we first describe bit-patterns for nine regions in the NA-tree. We then present our *NABP* method for continuous range queries over moving objects by checking bit-patterns based on the NA-tree. Finally, we give an example to illustrate our *NABP* method.

3.1 The Bit-Pattern of the Region in the NA-Tree

Since one region in the NA-tree contains a range of bucket numbers, we can get a region number by the bit-pattern derived from the bit-strings of bucket numbers. Take Figure 11 as an example. Region 2 contains a range of four bucket numbers from 4 to 7 . These four bucket numbers have bit-pattern '01' in the prefix two bits of their bit-strings, as shown in the dotted diagram. Thus, we can get region 2 by the bit-pattern '01'.

[Figure 11 about here.]

Figure 12 shows regions with the corresponding bit-patterns at level i ($i > 0$). The bucket number with the bit-string ($x_1y_1...x_iy_i$) which is ordered by the N-Order Peano

curve of order i is stored in the node with bit-pattern $x_i y_i$, x_i, or y_i at level i of the NA-tree. The $2nd_$child at level 1 represents region 2 with the bit-pattern $\underline{01}$. The symbol '$\underline{\ }$' denotes the checked bit-pattern which means that either or both bits x_i, y_i should be checked in the bit-strings of bucket numbers to get region number RN at level i of the NA-tree. The $5th_$child at level 1 represents region 5 which contains two small regions 1 and 2. Its bit-pattern is shown as $\underline{0*}$ by combining two bit-patterns: $\underline{00}$ and $\underline{01}$, of two regions 1 and 2, respectively. The symbol '$*$' denotes the arbitrary bit (0 and 1) which means that the region ranges across two half parts along the X or Y axis. It is similar to the other node which represent regions 6, 7, or 8. The difference is the checked bit-pattern. The $9th_$child at level 1 represents region 9 which contains four small regions from 1 to 4. Its bit-pattern is shown as $\underline{**}$ by combining four bit-patterns: $\underline{00}$ and $\underline{01}$, $\underline{10}$, and $\underline{11}$ of four regions 1, 2, 3, and 4 , respectively.

[Figure 12 about here.]

Since the internal node at level $(i-1)$ $(i \geq 1)$ can have nine, four, three, or two child nodes at level i, the checked bit-pattern of each child node is determined by its region number or the region number of its parent node. For the internal node with the region number ranging from 1 to 4 at level $(i-1)$, it has nine child nodes with the checked bit-pattern $x_i y_i$ at level i. For example, the $2nd_$child with region number 2 at level 1 in Figure 12 has nine child nodes at level 2 with corresponding two underlined bits in the checked bit-patterns.

For the internal node with region number 5 or 7 at level $(i-1)$, it has three child nodes with the checked bit-pattern x_i at level i. For example, the $5th_$child with region number 5 at level 1 in Figure 12 has three child nodes with region numbers 5, 7 and 9 at level 2. The region of the child node always ranges across two half parts along the Y axis and is represented by the bit '$*$'. But, the region of the child node may range in one or across two of left and right parts along the X axis and is represented by one underlined bit ('$\underline{0}$' or '$\underline{1}$') or '$\underline{*}$' in the checked bit-pattern. For the internal node with region number 6 or 8 at level $(i-1)$, it has three child nodes with the checked bit-pattern y_i at level i. For example, the $8th_$child with region number 8 at level 1 in Figure 12 has three child nodes with region numbers 6, 8, and 9 at level 2. The region of the child node always ranges

across two half parts along the X axis and is represented by the bit '$*$'. But, the region of the child node may range in one or across two of lower and upper parts along the Y axis and is represented by the underlined bit ('$\underline{0}$' or '$\underline{1}$') or '$\underline{*}$' in the checked bit-pattern.

For the internal node with region number 9 at level $(i-1)$, the number of its child nodes and the bit-patterns of its child nodes are determined by the region number of its parent node at level $(i-2)$. For its parent node with the region number ranging from 0 to 4, it has four child nodes with the checked bit-pattern $x_i y_i$ at level i. For its parent node with the region number ranging from 5 to 8, it has two child nodes with one bit x_i or y_i in the checked bit-pattern at level i. For example, the $9th_$child with region number 9 at level 1 in Figure 12, its parent node is root with region number 0 at level 0. It has four child nodes 1, 2, 3 and 4 at level 2 with two bits $x_2 y_2$ in the checked bit-patterns. For the other one example, the $9th_$child with region number 9 at level 2 whose parent node is with region number 5 at level 1 in Figure 12 has two child nodes with reigion number 5 and 7 at level 3. There is only one bit x_3 in the checked bit-patterns of these two child nodes.

Therefore, in our *NABP* method for the continuous query processing, we check the bit-pattern to get the region of a range query or a point based on the NA-tree. Then, we can retrieve the ralted range queries and points from the region and get the relation bwtween the range query and the moving object.

3.2 Continuous Query Processing

We assume that the space is a R-kilometer by R-kilometer region. Based on the bucket numbering scheme in the NA-tree [5], the space is divided n times along the X and Y axes to obtain the number of $2^n \times 2^n$ equal-sized buckets. The length w of a bucket along one axis is $(R/2^n)$-kilometer. In the continuous query processing, a range query Q would be inserted into or deleted from the space while a moving object would change the location over time. The server deals with two events: *range query update* and *point update*.

In the event of *range query update*, we perform four steps $A1$ to $A4$ in our *NABP* method which are shown in the left side of Figure 13. A range query is stored in the NA-tree as a *Minimal Boundary Rectangle* (MBR) with two pairs of the coordinates: (X_l, Y_b) and (X_r, Y_t). In Step $A1$, we transform two pairs of coordinates into the two bucket numbers which

is the spatial number for a range query. In Step $A2$, we check the bit-pattern in two bucket numbers to get a global region number *QNA_RN*, which denotes the region that the range query is located in the NA-tree. In Step $A3$, we decide the related region *PNA_RN* which is contained in region *QNA_RN*. In Step $A4$, if a moving object exists in region *PNA_RN*, we decide the relation between the range query and the moving object and broadcast the message about the relation at the server. Otherwise, we return the answer 'No Point'.

[Figure 13 about here.]

In the event of *point update*, we also perform four steps $B1$ to $B4$ in our *NABP* method which are shown in the right side of Figure 13, which are similar to steps $A1$ to $A4$ in the left side, respectively. However, a moving object is stored only as a point with one pair of the coordinates (X_p, Y_p) in the *point update*, instead of two pairs of coordinates in the *range query update*. Therefore, in Step $B4$, we check the the existence of the range query in the region which is related with the moving point in the relation decision.

3.2.1 A Range Query Update

While updating the range query Q with two pairs of coordinates: the lower left coordinate (X_l, Y_b) and upper right coordinate (X_r, Y_t), we perform four steps $A1$ to $A4$ as follows. In Step $A1$, we compute the spatial number $Q(l, u)$ as follows, where l is the bucket number of coordinate (X_l, Y_b) and u is the bucket number of coordinate (X_r, Y_t). We first divide two pairs of coordinates of range query Q: (X_l, Y_b) and (X_r, Y_t), by width w to get two pairs of bucket coordinates: LB (X_L, Y_B) and RT (X_R, Y_T). Then, we transform the decimal values of two pairs of bucket coordinates: LB (X_L, Y_B), RT (X_R, Y_T), into two pairs of bit-strings: $LB(x_1x_2...x_n, y_1y_2...y_n)$, $RT(x'_1x'_2...x'_n, y'_1y'_2...y'_n)$, respectively. The length of a bit-string is n bits depending on the N-Order Peano curve of order n used in the NA-tree. Finally, the bucket numbers l and u are obtained by interleaving x_i and y_i bits in two pairs of bit-strings: LB and RT as two DZ expressions: $l(x_1y_1x_2y_2...x_ny_n)$ and $u(x'_1y'_1x'_2y'_2...x'_ny'_n)$, respectively. We also compute the bucket number $c(x_{c1}y_{c1}...x_{cn}y_{cn})$ for the bucket coordinate CT (X_C, Y_C) of central point (X_c, Y_c) in range query Q.

In Step $A2$, we check the bit-pattern in three bit-strings: l, u, and c, following the related flowchart in Figure 14. Then, we can get the global region number *QNA_RN* for the location of range query Q in the NA-tree. We start at level 0 of the root in the NA-

tree and assume that the region number QRN of the whole space is 0. Since the region number QRN at level i in the NA-tree only ranges from decimal values 0 to 9, we compute the global region number QNA_RN by the equation QNA_RN=QNA_RN + $QRN * 10^{i-1}$ $(0 < i \leq n)$ once while updaing the range query Q. Thus, we can get the region number QRN at level i by retrieving the ith digit from the rightmost digit of the global region number QNA_RN, *i.e.*, QRN=$QNA\_RN(i)$, instead of re-checking the bit-patterns. The initial value of $QNA\_RN(0)$ is assumed to be 0.

[Figure 14 about here.]

Then, we first consider whether or not range query Q has ever ranged across two parts along the X and Y axes of the region for the root or the internal node at level (i-1) ($i \geq 1$). We use the variable *flag_9*=1 to denote for the above condition in which the range query should have ever been stored in the internal node which represents region $QNA\_RN(i)$=9 at level i. We should check the bit-pattern in bucket number c after level i of the internal node, which is *Case_9*. Otherwise, We should check both two bucket numbers: l and u in the other three cases: *Case_04*, *Case_57*, and *Case_68*.

In *Case_9*, we first consider whether or not the region number of the internal node is 9, *i.e.*, $QNA\_RN(i\text{-}1)$=9. If the condition of $QNA\_RN(i\text{-}1)$=9 is true, then the region number of child node is determined by the region number of its parent node at level (i-2), *i.e.*, $QNA\_RN(i\text{-}2)$. Otherwise, the region number of child node is determined by its region number $QNA\_RN(i\text{-}1)$. There are three cases used to check for the region number $QNA\_RN(i\text{-}1)$ or $QNA\_RN(i\text{-}2)$. In these three cases: *Case_9_04*, *Case_9_57*, and *Case_9_68*, we use *Condition_XY*, *Condition_X*, and *Condition_Y* to check the bit-string x_iy_i, x_i, or y_i in bucket number c, respectively. The checked bit-pattern x_iy_i, x_i, or y_i underlined in Figure 14 is used to check the region number of the child node where the centeral point CT of range query Q locates at level i.

In *Case_04*, we check the bit-pattern in two bit-strings: $x_ix'_i$ and $y_iy'_i$ from two bucket numbers l and u of range query Q, by two conditions: *Condition_H* (*i.e.*, the horizontal condition) and *Condition_V* (*i.e.*, the vertical condition), respectively.

$$Condition\_H(x_i, x_i') : R_x = \begin{cases} 0, & \text{if } x_i x_i'\text{=00, } i.e.,\ x_i = x_i' = 0\ (L); \\ 1, & \text{if } x_i x_i'\text{=11, } i.e.,\ x_i = x_i' = 1\ (R); \\ *, & \text{if } x_i x_i'\text{=01, } i.e.,\ x_i < x_i'\ (* : \text{arbitrary bit 0 or 1}). \end{cases}$$

$$Condition\_V(y_i, y_i') : R_y = \begin{cases} 0, & \text{if } y_i y_i'\text{=00, } i.e.,\ y_i = y_i' = 0\ (B); \\ 1, & \text{if } y_i y_i'\text{=11, } i.e.,\ y_i = y_i' = 1\ (T); \\ *, & \text{if } y_i y_i'\text{=01, } i.e.,\ y_i < y_i'\ (*\text{: arbitrary bit 0 or 1}). \end{cases}$$

In *Condition_H* (*Condition_V*), the condition of two equal bits: $x_i = x_i'$ ($y_i = y_i'$) means that both two bucket coordinates: X_L and X_R (Y_B and Y_T), are located in the same part: left(L) or right(R) part along the X-axis (the lower(B) or upper(T) part along the Y-axis). Otherwise, two bucket coordinates are located in the different parts along the X axis (Y axis) which is denoted by the symbol $*$. Then, we combine R_x and R_y as the checked bit-pattern to get the region number of the child node at level i according to Figure 15.

[Figure 15 about here.]

The *Case_57* (*Case_68*) is similar to *Case_04*. The difference is that we only have to check bit-string $x_i x_i'$ ($y_i y_i'$) to get the checked bit-pattern R_x (R_y) by condition *Condition_H* (*Condition_V*). The reason is that the internal node with region number *QNA_RN*(i-1) 5 or 7 (6 or 8) only has three child nodes with their region numbers: 5, 7, and 9 (6, 8, and 9), as shown in Figure 12. The range query Q in one of these child nodes always ranges across two parts along the Y axis (X axis), which results in $R_y = *$ ($R_x = *$). The bit-pattern checking for the global region number of range query Q stops at the leaf node of the NA-tree.

Let us take Figure 16 as an example. At level 1, the variable *flag_9*=0, region number *QRN*=0, global region number *QNA_RN*(0)=0 of the whole space satisfy *Case_04*. For range query Q in Figure 16-(a), we check two bit-strings: $x_1 x_1' = 00$ and $y_1 y_1' = 11$ by two conditions: *Condition_H* and *Condition_V*, respectively, to get the checked bit-pattern $R_x R_y = \underline{01}$. According to Figure 15, we get region number *QRN*=2 and global region number *QNA_RN*=$\underline{2}$, as shown in Figure 16-(b). At level 2, the variable *flag_9*=0 and region number *QNA_RN*(1)=2 of level 1 satisfy *Case_04*. For range query Q in Figure 16-(c), we check two bit-strings: $x_2 x_2' = 01$ and $y_2 y_2' = 11$ by two conditions: *Condition_H* and *Condition_V*, respectively, to get the checked bit-pattern $R_x R_y = \underline{*1}$. According to Figure

15, we get region number QRN=8 and the global region number QNA_RN=$\underline{8}2$, as shown in Figure 16-(d). At level 3, the variable *flag_9*=0 and region number $QNA\_RN(2)$=8 of level 2 satisfy *Case_68*. For range query Q in Figure 16-(e), we only have to check one bit-string $y_3y_3' = 00$ by *Condition_V* to get the checked bit-pattern $R_xR_y = *\underline{0}$. According to Figure 15, we get region number $QRN = 6$ and global region number $QNA\_RN = \underline{6}82$, as shown in Figure 16-(f).

[Figure 16 about here.]

In Step $A3$, we decide the related region $PNA\_RN(i)$, which is contained in region $QNA\_RN(i)$ at level i. The moving objects in region $PNA\_RN(i)$ may be contained or partially overlapped the range query Q in region $QNA\_RN(i)$. In the NA-tree, region $QNA\_RN(i)$ may contain one, two, or four small regions, as shown in Figure 12. In Step $A4$, we check the existence of the moving object in region $PNA\_RN(i)$. If there exists no moving object in region $PNA\_RN(i)$, the message of "No Point" will be output. Otherwise, each moving object P with one pair of bucket coordinates (X_P, Y_P) is retrieved from region $PNA\_RN(i)$. We decide the relation between range query Q and moving object P by making two distance comparisons on bit-patterns along the X and Y axes.

We find three relations between one bucket coordinate (X_P) of moving object P and two bucket coordinates (X_L, X_R) of range query Q along the X axis: (1) *Meet*; (2) *Contain*; (3) *Disjoin*. We use *Condition_*P_X with X_L, X_R, and X_P as the values of three parameters L, R, and P_X, respectively, to check these three relations, as shown in Figure 17. The one of first three equation ($L = P_X$, $P_X = R$ or $L = P_X = R$) means that P_X is in the same bucket as either or both L and R of range query Q and represents the relation 'Meet'. The forth equation ($L < P_X < R$) means that P_X is in the different bucket and the bucket is inside the range between L and R of range query Q and represents the relation 'Contain'. The one of last two equations ($L > P_X$ or $P_X > R$) means that P_X is in the different bucket but the bucket is outside the range between L and R of range query Q. The diagrams for these equations are shown in Figure 17. The processing is similar to get the relation along the Y axis . The difference is that we use *Condition_*P_Y along the Y axis to find three relations between one bucket coordinate (Y_P) of moving object P and two bucket coordinates (Y_B, Y_T) of range query Q along the Y axis.

[Figure 17 about here.]

Then, We make the combination of equations in two conditions: $Condition\_P_X$ and $Condition\_P_Y$ to help decide the relation between moving object P and range query Q in the two dimensional space. We derive seventeen kinds of $InType$'s, and conclude three relations: (1) *Partial Overlap*; (2) *Inside*; (3) *Outside*, as shown in Figure 18. For example, the relation '*Partial_Overlap* of '$InType = 1$ means that moving object P and range query Q are in the same bucket along the X and Y axes. The relation '*Partial_Overlap* of $InType = 7$ means that moving object P is in the same bucket as two bucket coordinates: B and T of the range query Q along the Y axis. But, it is in the different bucket and the bucket between two bucket coordinates: L and R of range query Q along the X axis. The relation '*Inside*' of $InType = 16$ means that moving object P is contained in the range query Q.

[Figure 18 about here.]

When the number of range queries increases over the bucket capacity (*i.e.*, the defined number of range queries in a bucket), we should make another one division along the X and Y axes to get the large number of buckets. The order n of the N-Order Peano curve should become large to be $(n+1)$ to order the increasing number of buckets used in the NA-tree. Then, the length of bit-strings for the original number of range queries becomes long. However, the prefix bits are not changed. Our *NABP* method only have to incrementally check the bit-pattern on those new added bits $x_{n+1}y_{n+1}$.

3.2.2 The Moving Object Update

While updating the moving object P with a pair of coordinates (X_p, Y_p), we perform four steps $B1$ to $B4$ in Figure 13. The processing of Steps $B1$ to $B4$ is similar to that of Steps $A1$ to $A4$ for the event of the range query update. However, in Step $B1$, we compute only one bucket number b for the moving object P. In Step $B2$, we check the bit-pattern in the bit-string x_iy_i of bucket number b by *Condition_XY*. Then, we get the global region number *PNA_RN* which represents the region that the moving object P locates in the NA-tree. In Step $B3$, we decide the related region *QNA_RN(i)* which contains region *PNA_RN(i)* at level i $(0 < i \leq n)$ of the NA-tree. The region with *QNA_RN(i)* may have range queries which contain or partially overlap moving object P. For example, the region

QNA_RN(i)(=3, 6, 7 or 9) contains the region *PNA_RN(i)*(=3). In Step $B4$, we retrieve each range query from the region *QNA_RN(i)* at level i of the NA-tree and compare it with the moving object P. Then, we can determine whether or not object P is inside or partially overlapped the range query.

3.3 An Example

Let us take Figure 19 as an example to illustrate our *NABP* method for continuous range queries over moving objects. The space is assumed as a 1 kilometer by 1 kilometer region and is partitioned into $2^3 \times 2^3 = 64$ equal-sized buckets. These buckets are linearly organized by the N-Order Peano curve of order $n = 3$. The width w of each bucket is equal to $1000/2^3 = 125$ meters. At time T_0 in Figure 19-(a), one range query Q_2 and two moving objects P_1 and P_2 initially exist in the space.

Then, a new range query Q_1 with two pairs of coordinates: $LB(75, 795)$ and $RT(350, 850)$, is inserted into the space. The processing of range query Q_1 insertion is shown in Figure 20. After Step $A4$, there is no object in node 2 at level 1 of the NA-tree with range query Q_1, as shown in Figure 19-(b).

[Figure 19 about here.]

[Figure 20 about here.]

At time T_1 in Figure 19-(c), moving object P_1 moves to (260, 800). The processing of moving object P_1 update is shown in Figure 21. After Step $B4$, range query Q_1 is found in node 2 at level 1 of the NA-tree which may have the relation with moving object P_1, as shown in Figure 19-(d). We decide the relation between range query Q_1 ($LB(X_L,Y_B)$=(0,6), $RT(X_R,Y_T)$=(2,6)) and moving object $P_1(X_P,Y_P)$=(2,6). After checking by two conditions: $Condition\_P_X(X_P = X_R)$ and $Condition\_P_Y(Y_B = X_P = Y_T)$, which is $Intype = 6$ in Figure 18, the relation between range query Q_1 and moving object P_1 is '*Partial Overlap*'.

[Figure 21 about here.]

4 The Performance Study

In this section, we compare the performance of the continuous range query processing among our *NABP* method, the Kalashnikov *et al.*'s cell-based method [10, 11], and R^*-

tree-based method (the variation of the R-tree) [1, 4, 27].

4.1 The Performance Model

The issue of obtaining the updated locations of objects is independent of the technique used for evaluating the queries. Since we focus on the efficient evaluation of queries, we assume that the network is stable for the communication such that updated information is available at the server, without considering how exactly it was made available [10, 11]. We concern about the performance of the server to deal with the large number of updated information in the period of time, including the updating locations of moving objects and the evaluation of range queries. New locations objects are generated and distributed uniformly in each cycle. The location of an object can change greatly from one cycle to the next without having any impact on the performance. While some index structures for moving objects rely upon restricted models of movement, the query indexing allows objects to move arbitrarily. It also happened that the number of range queries increases and range queries changes their ranges. At this point, the range query index should be rebuilt. Therefore, since the overall object distribution chosen for the experiment is maintained, the objects can move anywhere and are not stored based on the index structure. Moving objects were represented as points and the number of objects ranges from 10^5 to 10^6. Range queries were represented as squares with the average size 0.000025% and 0.0001%. The number of range queries ranges from 10000 to 15000. Moving objects and range queries are uniformly distributed on the space 1000×1000.

In our *NABP* method, the range queries were assigned to the corresponding bucket based on the N-order Peano Curve in the NA-tree. The range queries in the same bucket were linked. The number of spatial objects in a bucket, denoted as *bucket_capacity*, was assigned to be ten range queries. Because of the limitation of *bucket_capacity*, an N-order Peano curve of order 6 was required for 10000 to 15000 range queries. Then, the number of 10^6 data points were randomly updated to evaluate the range queries. For the cell-based method, the space was chosen to consist of 1000×1000 cells. Range queries and moving objects were generated and put into arrays. The index was initialized and the range queries were added to it [10, 11]. The capacity of each cell was assigned to be the number of ten

range queries. For the R^*-tree-based method, the capacity of each node was assigned to be the number of ten range queries or nodes. Each entry in the leaf node contained the coordinates of a range query. Since the sequence of inserting range queries affects the structure of the R^*-tree, we sort and group the range queries by using the Hilbert curve on building and processing an R^*-tree for datasets and use the pruning metrics in [4, 27].

To compare the performance of three methods above, we took 1000 averages of the above results required to the evaluation. Since storage usage is related with the CPU time for the range queries update and moving objects update, we consider the CPU time and the storage cost as performance measures. The CPU time is used to measure the time which takes for the location computation and comparison for the moving objects updating and range queries updating. The storage cost is used to measure the number of cells in the cell-based method and the number of nodes in our *NABP* method for storing all the range queries in the space. In order for the solution to be effective, it is necessary to efficiently compute the relation between large number of objects and queries. Since spatial indices built in the main memory would perform better than disk-oriented structures [10, 11], we take indices optimized for the main memory to process the efficient and scalable continuous queries. Therefore, we focus on the measure of the CPU time, instead of the I/O time.

4.2 Simulation Results

In this simulation, we consider the case of the dataset with constant number of moving objects updates (10^6) and the variable number of range queries update (10000 to 15000). Figure 22 shows the result of the storage cost for the range queries update. Our *NABP* method requires lower storage cost than the cell-based method. Since each cell in the cell-based method stores the pointers to all queries that fully or partially cover the cell, the storage cost increases as the number of the range queries increases or the average size of the range queries increases. However, our *NABP* method uses the *NA*-tree to store the range query in only one node. The storage cost of our *NABP* method increases a little as the number of range queries increases.

[Figure 22 about here.]

Figure 23-(a) shows the result of the CPU time for the range queries update. Our

NABP method requires less CPU time than the cell-based method and the R^*-tree-based method. Since the range query may ranges across more cells in the cell-based method, it requires CPU time to compute the cells which are fully or partially contained in the range query. The CPU time of the cell-based method increases as the number of range queries increases. In the R^*-tree-based method, it requires more CPU time to find, delete, and insert on the related nodes for the range query update. However, our *NABP* method uses the bit-patterns operation with order n of the N-Order Peano curve to obtain the location of the range query. The CPU time of our *NABP* method increases a little when the order of the N-Order Peano curve is changed for the increase of the number of range queries.

[Figure 23 about here.]

Figure 23-(b) shows the result of the CPU time for the number of 10000 range queries over the number of 10^5 and 10^6 moving points update. Because the cell-based method has high storage cost shown in Figure 22, the cell-based method requires less CPU time than our *NABP* method. The reason is the tradeoff between the space and time complexity. Since each cell in the cell-based method stores all the pointers to the range queries which fully or partially contains it, it requires to search the cell where the moving object locates. However, our *NABP* method uses the bit-patterns operation with order n of the N-Order Peano curve to compute all range queries which contain the moving object. Moreover, our *NABP* method requires less CPU time than the R^*-tree-based method. In the R^*-tree-based method, it requires more CPU time to search in the related and overlapped nodes and find all range queries which contain the moving object.

5 Conclusion

In this paper, we have presented the *NABP* method for the continuous range queries over the moving objects. Our *NABP* method uses the bit-patterns of regions based on the NA-tree to check the relation between the range queries and moving objects. Our method searches only one path in the NA-tree for the range query, instead more than one tree paths in the R^*-tree-based method due to the overlapping problem. When the number of range queries increases with time, our *NABP* method can incrementally update the affected range queries by bit-patterns checking, instead of rebuilding the index like the cell-based method.

From our simulation study, we have shown that our *NABP* method requires less CPU time and less storage cost than the cell-based method for large number of range queries update. We also have shown that our *NABP* method requires less CPU time than the R^*-tree-based method for large number of moving objects update.

Acknowledgement

The authors like to thank National Sun Yat-Sen University, "Aim for Top University Plan" project of NSYSU and Ministry of Education, Taiwan, for partially supporting the research.

References

[1] N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles,"*Proc. of the ACM SIGMOD Int. Conf. on Management of Data,* 1990, pp. 322–331.

[2] M. Cai and P. Revesz, "Parametric R-Tree: An Index Structure for Moving Objects," *Proc. of the Int. Conf. on Management of Data*, 2000, pp. 57–64.

[3] Y. Cai, K. A. Hua, and G. Cao, "Processing Range-Monitoring Queries on Heterogeneous Mobile Objects," *Proc. of the Int. Conf. on Mobile Data Management*, 2004, pp. 27–38.

[4] Y. Chen and J. M. Patel, "Efficient Evaluation of All-Nearest-Neighbor Queries," *Proc. of the 23rd Int. Conf. on Data Eng.,* 2007, pp. 1056–1065.

[5] Ye-In Chang, Cheng-Huang Liao and Hue-Ling Chen, "*NA*-Trees: A Dynamic Index for Spatial Data," *Journal of Information Science and Eng.,* 2003, 19, (1), pp. 103–139.

[6] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. of the ACM SIGMOD Int. Conf. on Management of Data,* 1984, pp. 47–57.

[7] Y. L. Hsueh, R. Zimmermann, H. Wang, and W. S. Ku, "Partition-Based Lazy Updates for Continuous Queries over Moving Objects," *Proc. of the 15th Int. Symp. on Advances in Geographic Information Systems*, 2007, pp. 1–8.

[8] H. Hu, J. Xu, H. Wang, and D. L. Lee, "PAM: An Efficient and Privacy-Aware Monitoring Framework for Continuously Moving Objects," *IEEE Trans. on Data and Knowledge Eng.*, 2009, http://doi.ieeecomputerssociety.org/10.1109/TKDE.2009.86.

[9] S. M. Jang, S. II. Song, and J. S. Yoo, "An Efficient PAB-Based Query Indexing for Processing Continuous Queries on Moving Objects," *ETRI Journal*, 2007, 29, (5), pp. 691–693.

[10] D. V. Kalashnikov, S. Prabhakar, W. G. Aref, and S. E. Hambrusch, "Efficient Evaluation of Continuous Range Queries on Moving Objects," *Proc. of the 13th Int. Conf. on Database and Expert Systems Applications*, 2002, pp. 731–740.

[11] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch, "Main Memory Evaluation of Monitoring Queries on Moving Objects," *Distributed and Parallel Databases*, 2004, 15, (2), pp. 117–135.

[12] D. Lin, C. S. Jensen, B. C. Ooi, and S. Saltenis, "Efficient Indexing of the Historical, Present, and Future Positions of Moving Objects, " *Proc. of the Int. Conf. on Mobile Data Management*, 2005, pp. 59-66.

[13] F. Liu, K. A. Hua, and F. Xie, "On Reducing Communication Cost for Distributed Moving Query Monitoring Systems," *Proc. of the 9th Int. Conf. on Mobile Data Management*, 2008, pp. 156–164.

[14] M. F. Mokbel, X. Xiong, and W. G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases," *Proc. of the ACM Int. Conf. on Management of Data*, 2004, pp. 623–634.

[15] Y. H. Park, K. S. Bok, and J. S. Yoo, "Continuous Range Query Processing over Moving Objects," *IEICE Trans. on Information and Systems*, 2008, E91-D, (11).

[16] M. Pelanis, S. Saltenis, and C. Jensen, "Indexing the Past, Present, and Anticipated Future Positions of Moving Objects," *ACM Trans. on Database Systems*, 2006, 31, (1), pp. 255–298.

[17] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch, "Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects," *IEEE Trans. on Computers*, 2002, 51, (10), pp. 1124–1140.

[18] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the Positions of Continuously Moving Objects," *ACM SIGMOD Record*, 2002, 29, (2), pp. 331–342.

[19] T. Sellis, N. Roussopoulos and C. Faloutsos, "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. of the 13th Int. Conf. on VLDB,* 1987, pp. 507–518.

[20] Y. Tao and D. Papadias, "Spatial Queries in Dynamic Environments," *ACM Trans. on Database Systems*, 2003, 28, (2), pp. 101–139.

[21] H. Wang, R. Zimmermann, and W. S. Ku, "Distributed Continuous Range Query Processing on Moving Objects," *Proc. of the 17th Int. Conf. on Database and Expert Applications*, 2006, pp. 655–665.

[22] K. L. Wu, S. K. Chen, and P. S. Yu, "Shingle-Based Query Indexing for Location-Based Mobile E-Commerce," *Proc. of the IEEE Int. Conf.on E-commerce Technology*, 2004, pp. 16–23.

[23] K. L. Wu, S. K. Chen, and P. S. Yu, "Efficient Processing of Continual Range Queries for Location-Aware Mobile Services," *Infomation Systems Frontiers*, 2005, 7, (5), pp. 435–448.

[24] K. L. Wu, S. K. Chen, and P. S. Yu, "Incremental Processing of Continual Range Queries over Moving Objects," *IEEE Trans. on Knowledge and Data Eng.*, 2006, 18, (11), pp. 1560–1575.

[25] X. Xiong, M. F. Mokbel, W. G. Aref, S. E. Hambrusch, and S. Prabhakar, "Scalable Spatio-temporal Continuous Query Processing for Location-aware Services," *Proc. of the 16th Int. Conf. on Scientific and Statistical Database Management*, 2004, pp. 317–326.

[26] X. Xiong, M. F. Mokbel, and W. G. Aref, "LUGrid: Update-tolerant Grid-based Indexing for Moving Objects," *Proc. of the 7th Int. Conf. on Mobile Data Management*, 2006, pp. 1–15.

[27] J. Zhang, N. Mamoulis, D. Papadias and Y. Tao, "All-Nearest-Neighbors Queries in Spatial Databases," *Proc. of the 16th Int. Conf. on Scientific and Statistical Database Management,* 2004, pp. 297–306.

List of Figures

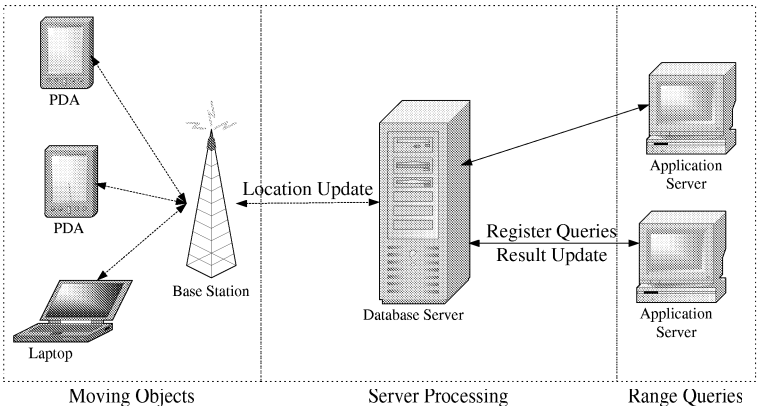


Figure 1: The system architecture for continuous range queries

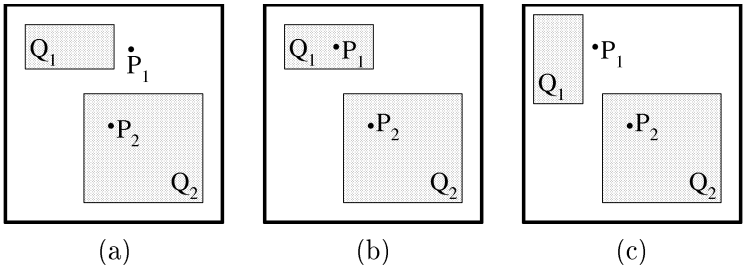


Figure 2: An example of continuous range queries over moving objects: (a) at time T_0; (b) at time T_1; (c) at time T_2.

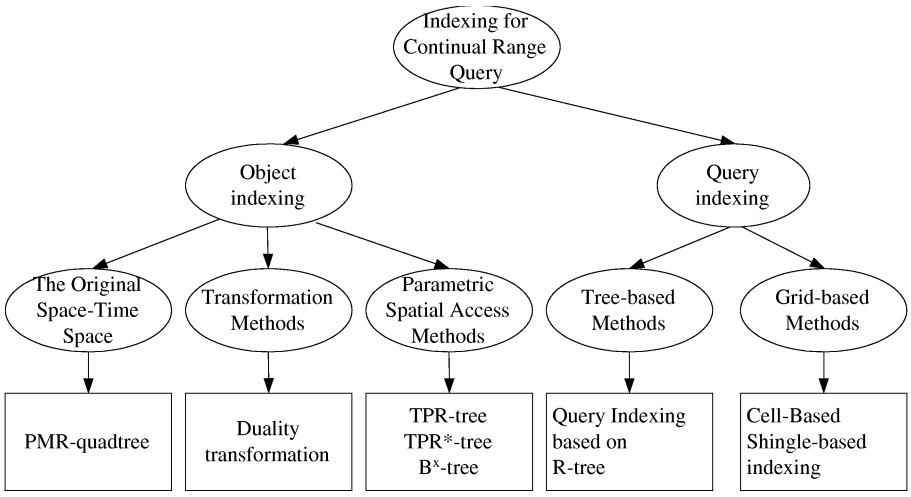


Figure 3: The indexing methods for continuous range queries

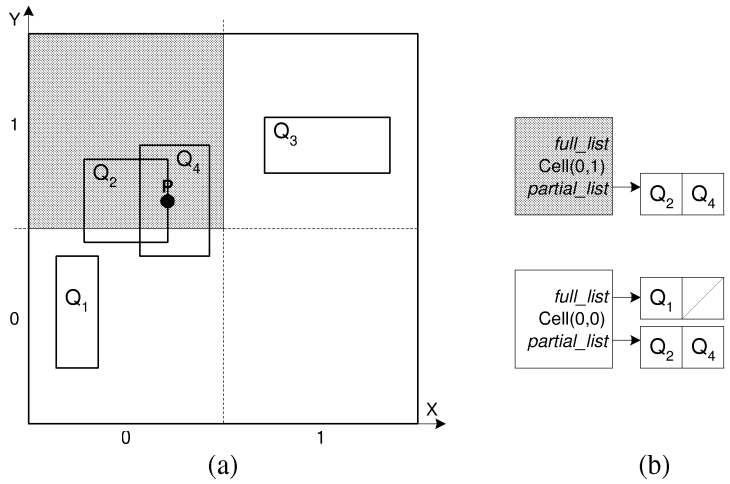


Figure 4: The cell-based method

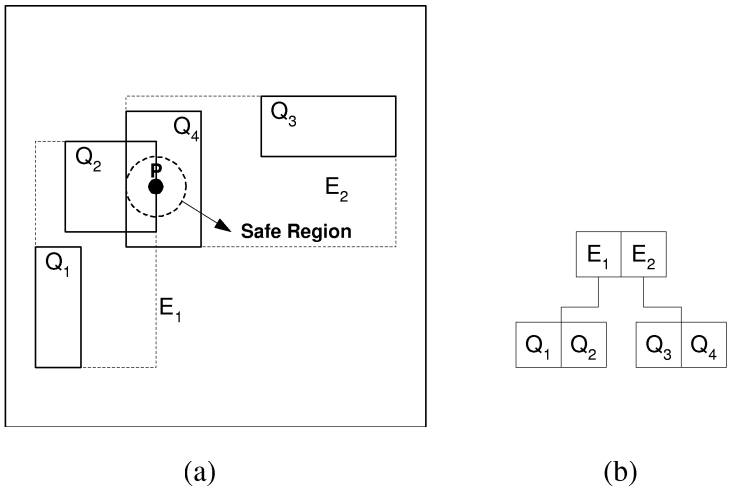


Figure 5: The R^*-tree-based method: (a) a point and range queries; (b) the structure of R^*-tree.

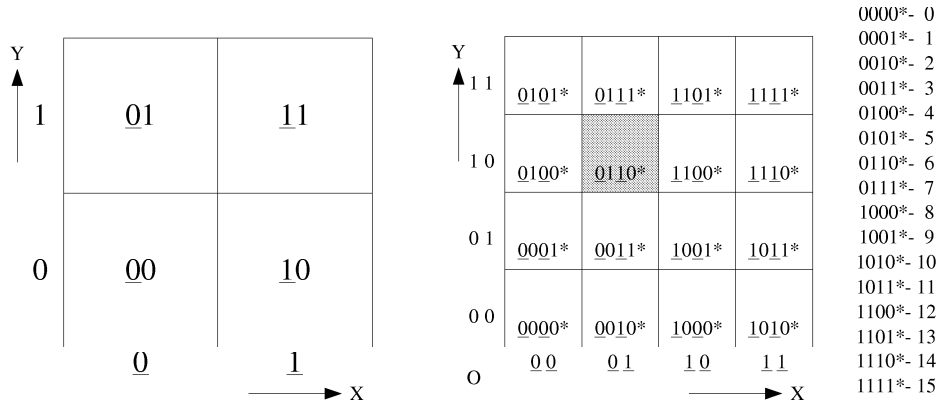


Figure 6: Space decomposition and DZ Expression: (a) one division with four regions; (b) two divisions with sixteen regions.

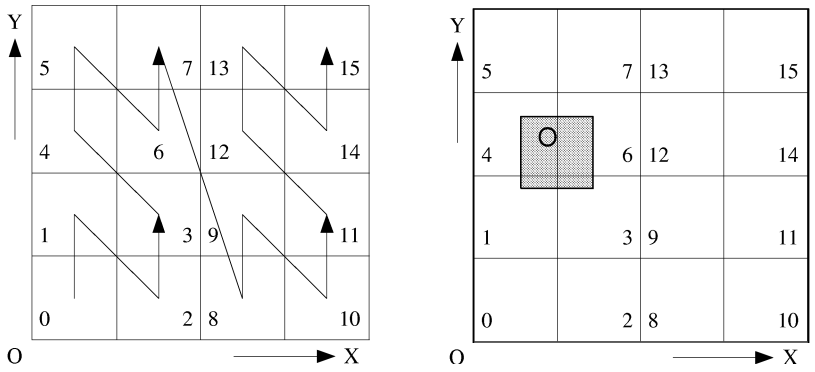


Figure 7: An example of the bucket numbering scheme: (a) the N-Order Peano Curve of order 2; (b) bucket numbers of spatial object O.

Region	Bucket number b
I	$0 \le b \le \frac{1}{4}(Max\_bucket+1)-1$
II	$\frac{1}{4}(Max\_bucket+1) \le b \le \frac{1}{2}(Max\_bucket+1)-1$
III	$\frac{1}{2}(Max\_bucket+1) \le b \le \frac{3}{4}(Max\_bucket+1)-1$
IV	$\frac{3}{4}(Max\_bucket+1) \le b \le (Max\_bucket+1)-1$

(a)

Region II	Region IV
Region I	Region III

(b)

Four Regions				*child*
I	***II***	***III***	***IV***	***node***
l, u				***1***
	l, u			***2***
		l, u		***3***
			l, u	***4***
l	***u***			***5***
l		***u***		***6***
		l	***u***	***7***
	l		***u***	***8***
l			***u***	***9***

(c)

Figure 8: Regions in the NA-tree: (a) four ranges of bucket numbers; (b) four regions; (c) nine nodes.

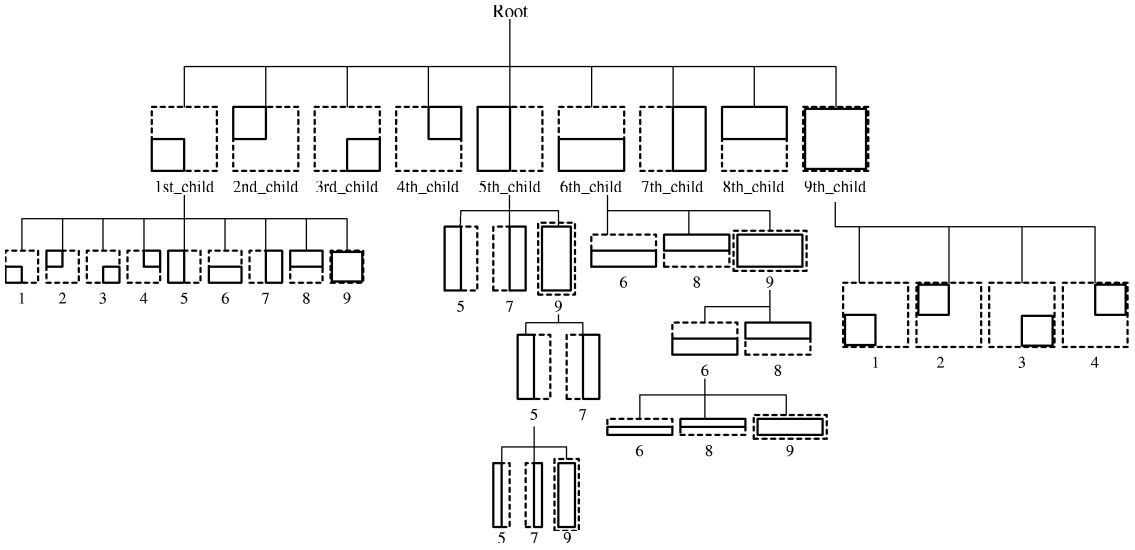


Figure 9: The basic structure of an NA-tree

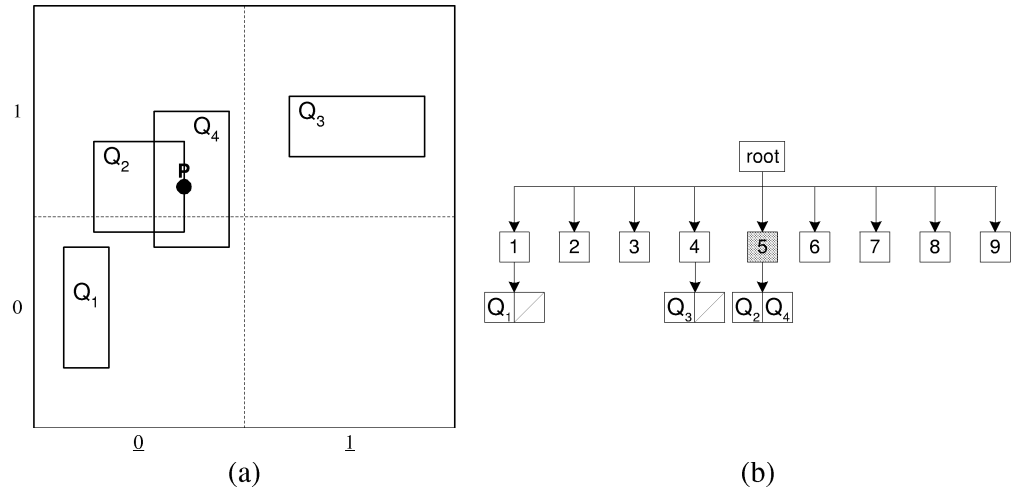


Figure 10: Our $NABP$ method: (a) the point P and range queries; (b) the corresponding NA-tree.

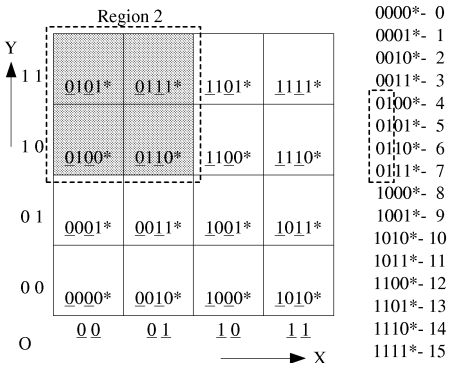


Figure 11: Region 2 for a range of bucket numbers from 4 to 7 with the bit-pattern '01'.

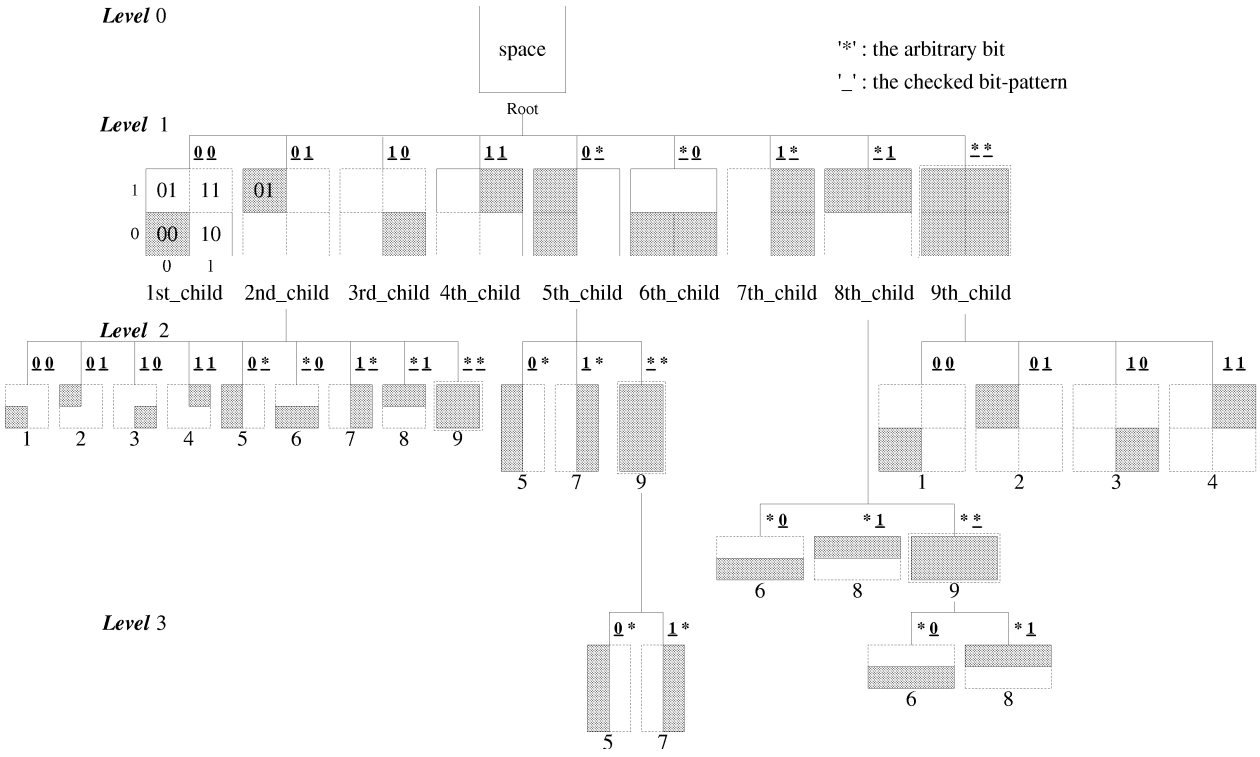


Figure 12: The checked bit-patterns based on the NA-tree

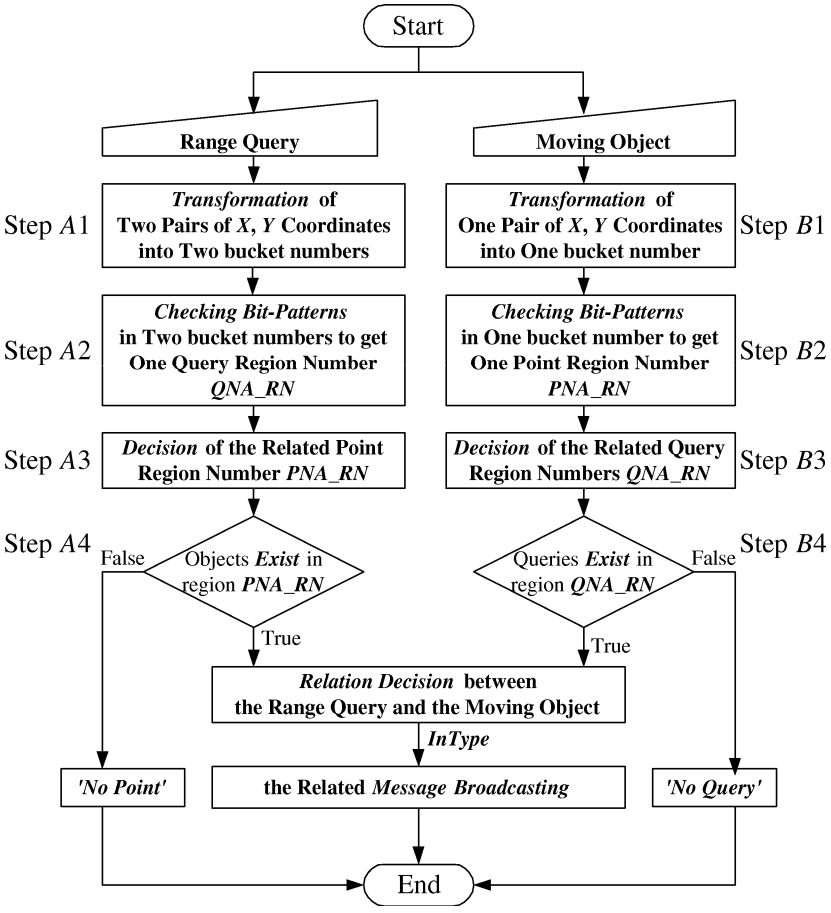


Figure 13: Steps of continuous query processing

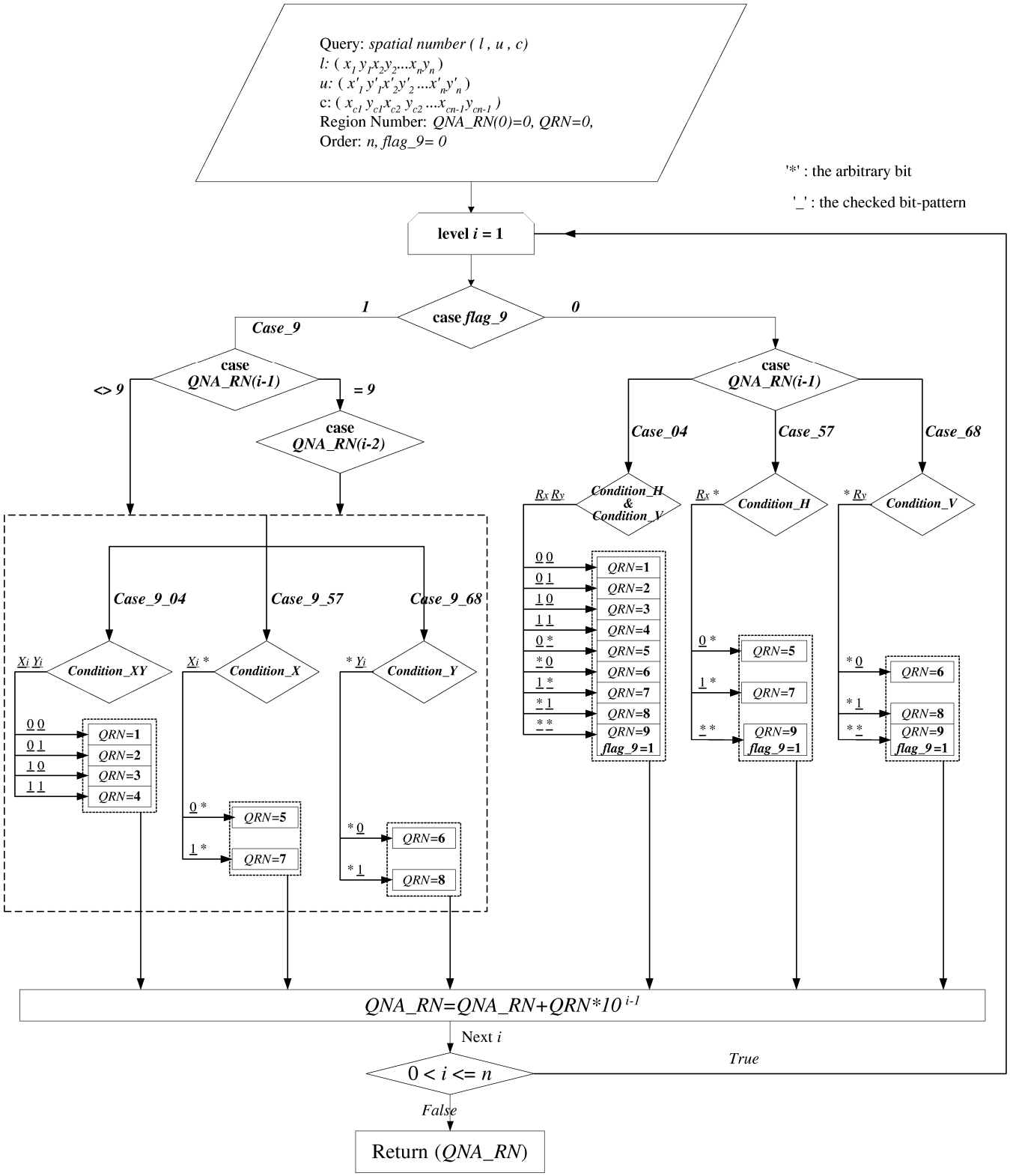


Figure 14: The flowchart to get the global region number QNA_RN for range query Q

Condition_H		Condition_V		Checked Bit-Pattern	Region Number
X_i	X'_i	Y_i	Y'_i	$Rx\ Ry$	QRN
0 (L)	0 (L)	0 (B)	0 (B)	0 0	1
		1 (T)	1 (T)	0 1	2
		0 (B)	1 (T)	0 *	5
1 (R)	1 (R)	0 (B)	0 (B)	1 0	3
		1 (T)	1 (T)	1 1	4
		0 (B)	1 (T)	1 *	7
0 (L)	1 (R)	0 (B)	0 (B)	* 0	6
		1 (T)	1 (T)	* 1	8
		0 (B)	1 (T)	* *	9

Figure 15: The checked bit-pattern for the region number QRN at level i of the NA-tree

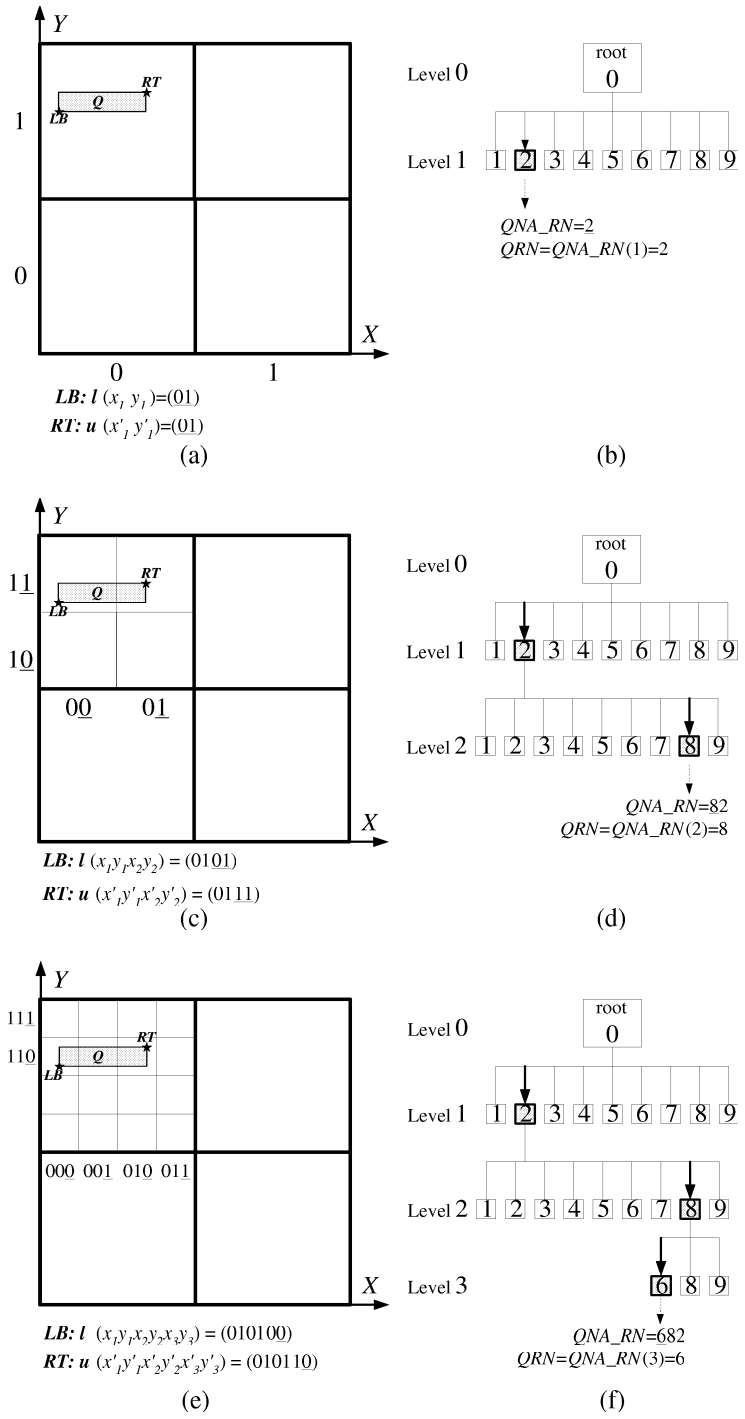


Figure 16: The location of query Q: (a) the N-order Peano Curve of order 1; (b) the level 1 of the NA-tree; (c) the N-order Peano Curve of order 2; (d) the level 2 of the NA-tree; (e) the N-order Peano Curve of order 3; (f) the level 3 of the NA-tree.

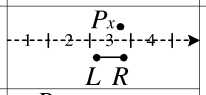
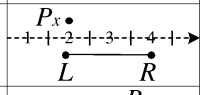
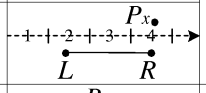
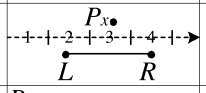
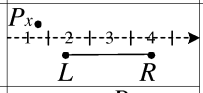
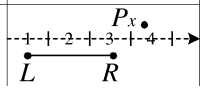
Condition_Px	Diagram	Relation
$L=P_x=R$		*Meet*
$L=P_x$		
$P_x=R$		
$L<P_x<R$		*Contain*
$P_x<L$		*Disjoin*
$P_x>R$		

Figure 17: Relations along one dimensional axis

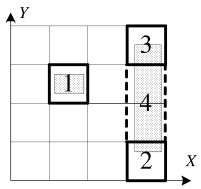
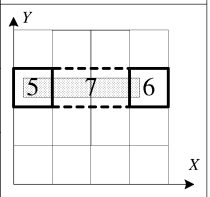
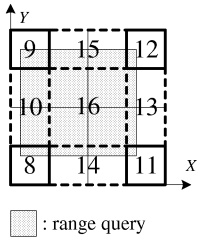
Conditions		Result		
Condition_Px	*Condition_Py*	*InType*	Diagram	Relation
$L=P_x=R$	$B=P_y=T$	1		*Partial Overlap*
	$B=P_y$	2		
	$P_y=T$	3		
	$B<P_y<T$	4		
$L=P_x$	$B=P_y=T$	5		
$P_x=R$		6		
$L<P_x<R$		7		
$L=P_x$	$B=P_y$	8		
	$P_y=T$	9		
	$B<P_y<T$	10		
$P_x=R$	$B=P_y$	11		
	$P_y=T$	12		
	$B<P_y<T$	13		
$L<P_x<R$	$B=P_y$	14		
	$P_y=T$	15		
	$B<P_y<T$	16		*Inside*
Others		0	*Outside*	

Figure 18: Relations in the two dimensional space

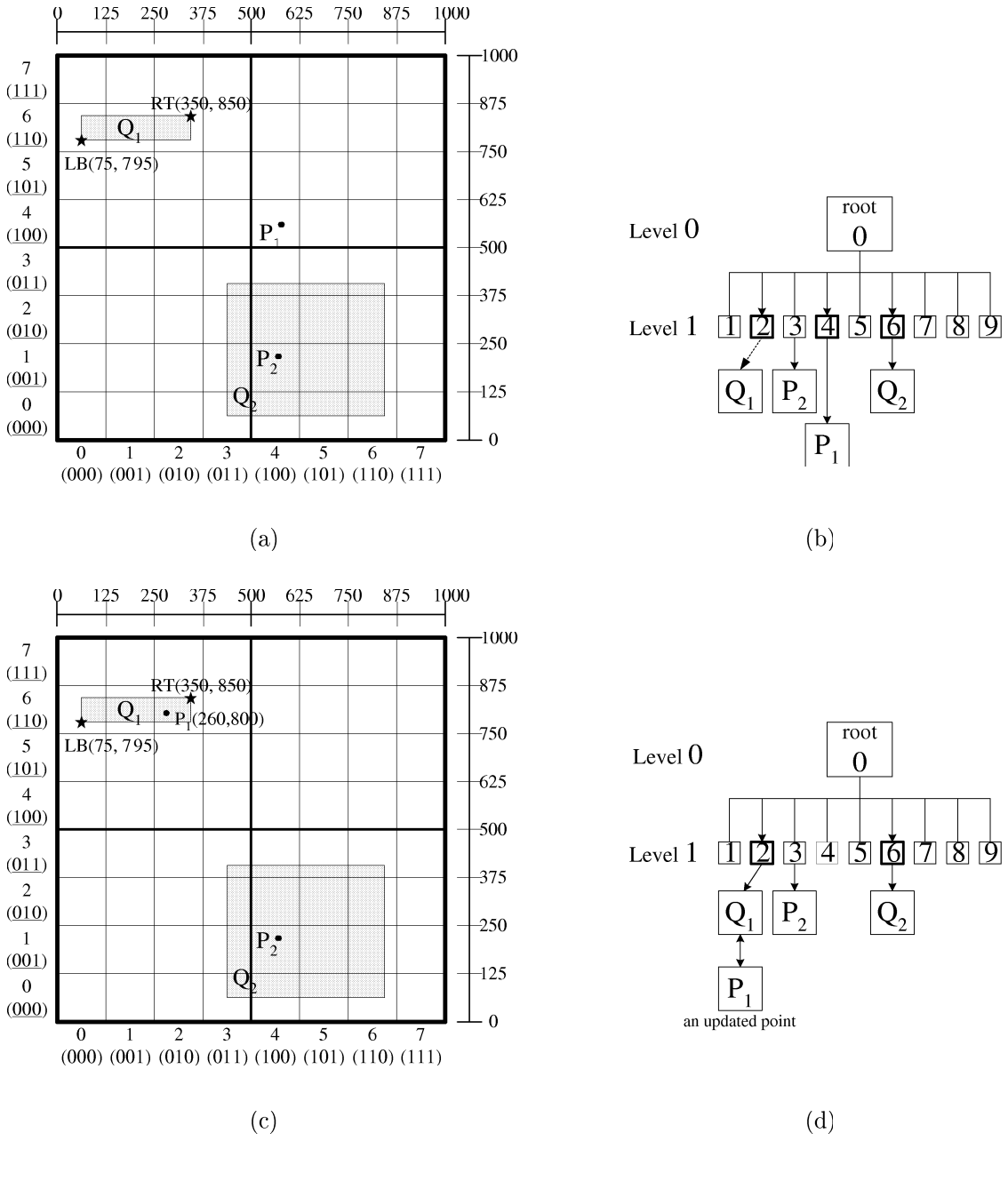


Figure 19: An example based on the N-Order Peano Curve of order 3: (a) the dataset at time T_0; (b) the NA-tree at time T_0; (c) the dataset at time T_1; (d) the NA-tree at time T_1.

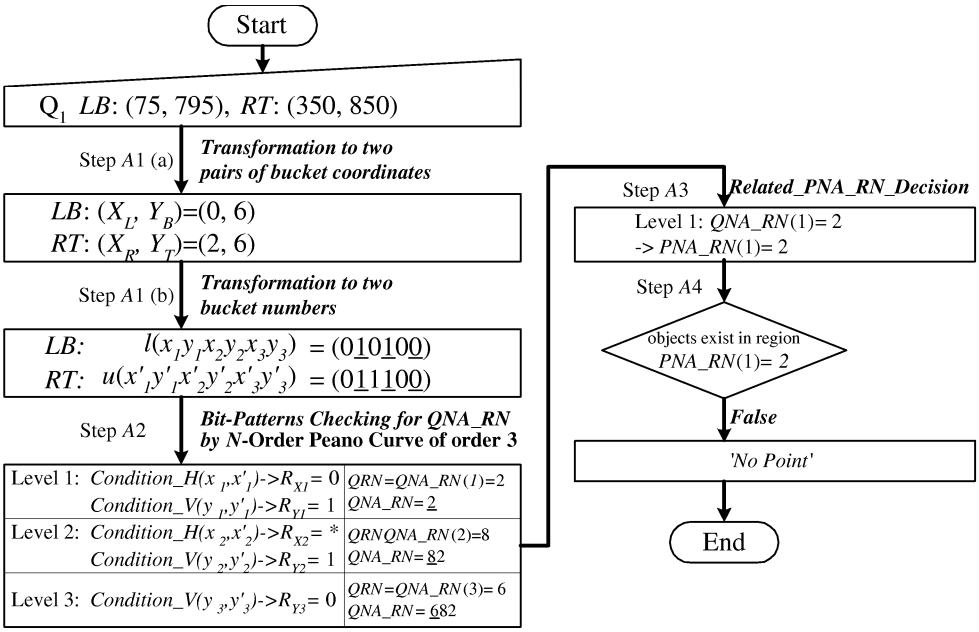


Figure 20: The process of query Q_1 update at time T_0

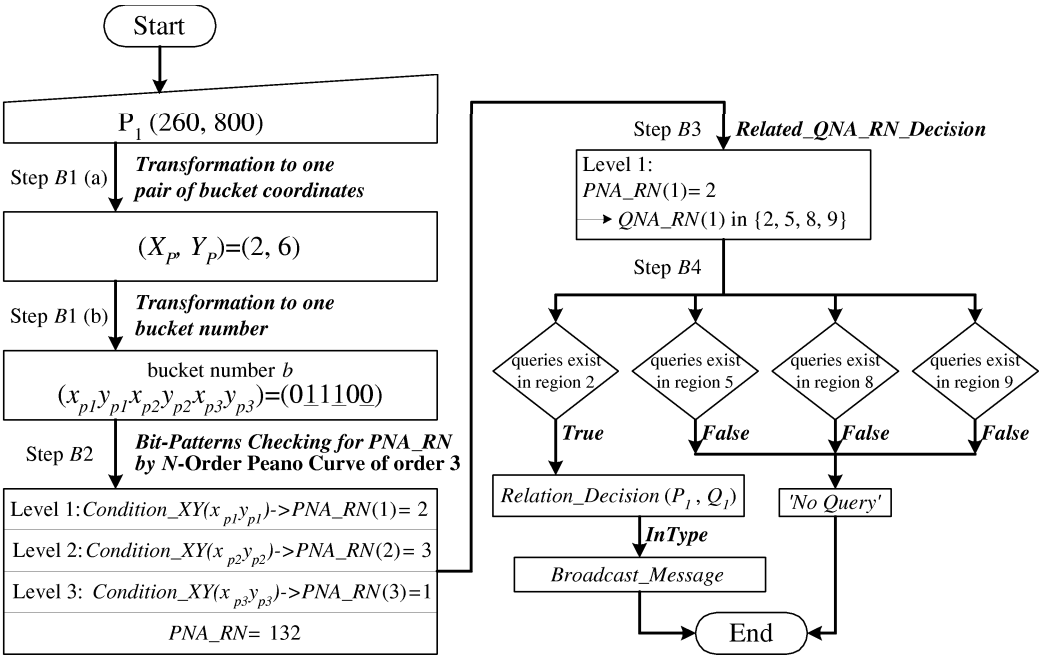


Figure 21: The process of point P_1 update at time T_1

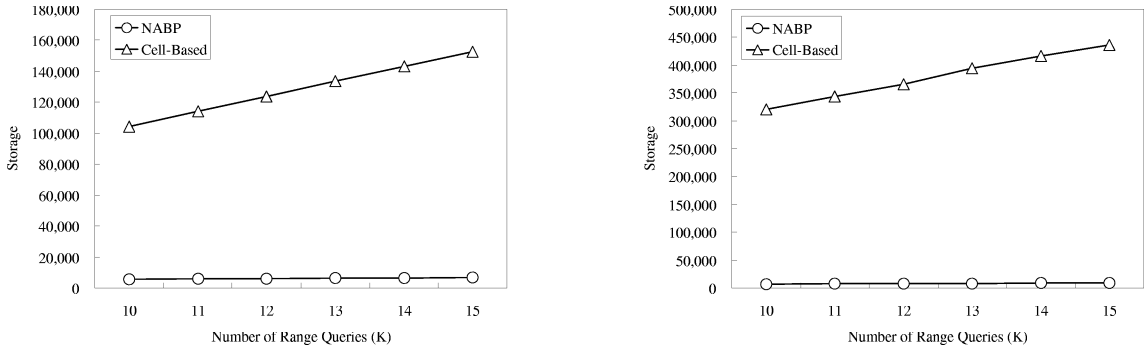


Figure 22: A comparison of storage cost ($K = 10^3$): (a) average size of range queries: 0.000025%; (b) average size of range queries: 0.0001%.

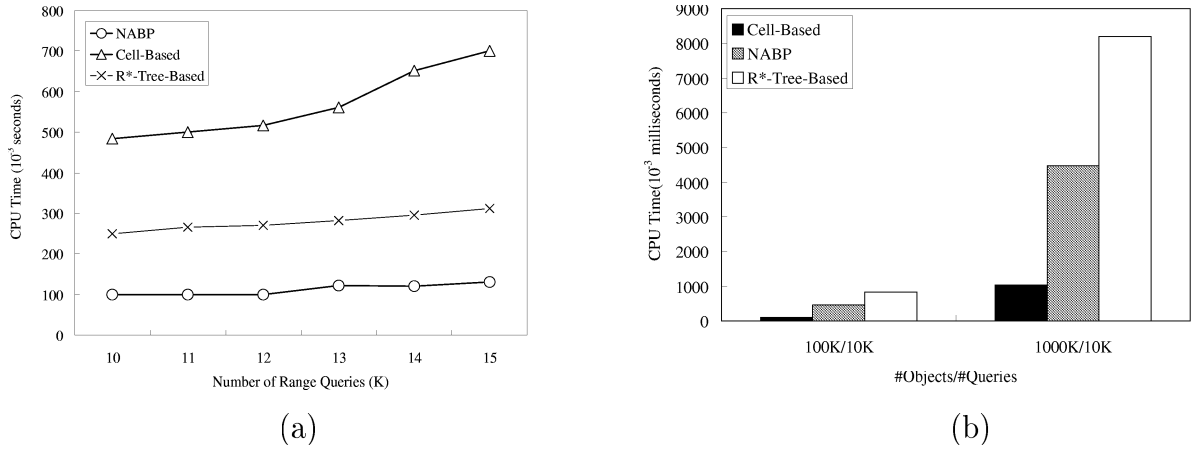


Figure 23: A Comparison of CPU time ($K = 10^3$): (a) range queries update; (b) moving objects update.