RC23635 (W0506-084) June 21, 2005
Computer Science

IBM Research Report

On Incremental Processing of Continual Range Queries for Location-Aware Services and Applications

Kun-Lung Wu, Shyh-Kwei Chen, Philip S. Yu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

On Incremental Processing of Continual Range Queries for Location-Aware Services and Applications

Kun-Lung Wu, Shyh-Kwei Chen, and Philip S. Yu
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
$\{$*klwu, skchen, psyu*$\}$*@us.ibm.com*

Abstract

A set of continual range queries, each defining the geographical region of interest, can be periodically re-evaluated to locate moving objects. Processing these continual queries efficiently and incrementally hence becomes important for location-aware services and applications. In this paper, we study a new query indexing method, called CES-based indexing, for incremental processing of continual range queries over moving objects. A set of *containment-encoded squares* (CES) are predefined, each with a unique ID. CES's are *virtual constructs* (VC) used to decompose query regions and to store indirectly pre-computed search results. Compared with a prior VC-based approach, the number of VC's visited in an index search in CES-based indexing is reduced from $(4L^2 - 1)/3$ to $\log(L) + 1$, where L is the maximal side length of a VC. Search time is hence significantly lowered. Moreover, containment encoding among the CES's makes it easy to identify all those VC's that need not be visited during an incremental query reevaluation. We study the performance of CES-based indexing and compare it with a prior VC-based approach.

Keywords: Query Indexing, Moving Objects, Mobile Computing, Location-Aware Applications, and Continual Range Queries.

1 Introduction

With the advances in mobile computing and location-sensing technologies, location-aware services and applications have become possible. Such applications can be used to deliver relevant, timely and engaging content and information to targeted customers. For example, a retail store in a mall can send timely e-coupons to the PDA's or cell-phones of potential customers who are close to the store.

To provide location-aware services and applications, one must first know where moving objects are currently located. A set of continual range queries, each defining the geographical regions of interest, can be repeatedly re-evaluated to locate moving objects. For example, we can place a square or a circle around the location of a hotel, an apartment building, or a subway exit. Efficient processing of a large number of continual range queries over moving objects is hence critically important.

Depending on whether or not queries move, the processing of continual range queries over moving objects can be roughly classified into two categories: (a) stationary queries over moving objects, e.g., [3,

11, 17, 25], and (b) moving queries over moving objects, e.g., [6, 13]. When the query region is associated with a moving object, such as a moving vehicle, it is a moving query. For example, a range query associated with a taxi cab can be used to find the customers who are close to the taxi at a given moment. In this paper, we focus on stationary queries where a query region is associated with a stationary object, such as a landmark.

Query indexing has been used to speed up the processing of continual static range queries over moving objects [11, 17, 25]. Periodically, each object position is used to search the query index to find all the range queries that contain the object. Once the containing range queries are identified, the object ID is inserted into the results associated with the identified queries. After every object position is searched against the query index, the most up-to-date results for all the continual range queries are available.

In [25], a VCR-based query indexing approach was proposed for incremental processing of continual range queries over moving objects. It was main memory based and was shown to outperform other query indexing approaches. It uses a set of predefined *virtual construct rectangles* (VCR) to decompose query regions and to store indirectly pre-computed search results. However, many of the VCR's are redundant, unnecessarily slowing down index search time and query evaluation time.

In this paper, we propose a new query indexing method, called CES-based indexing, for incremental processing of continual static range queries over moving objects. CES stands for *containment-encoded squares*. Similar to VCR's, CES's are *virtual constructs* (VC) used to decompose query regions and to store pre-computed search results. However, the set of predefined virtual constructs, the decomposition algorithm and the search algorithm are all different.

There are fewer CES's defined than VCR's. More importantly, the number of CES's visited during an index search in CES-based indexing is only $\log(L) + 1$, much smaller than $(4L^2 - 1)/3$ in a square-only VCR-based indexing, where L is the maximal side length of a VC. Search time is hence significantly lowered. There are containment relationships between the virtual constructs in CES-based indexing. These relationships are encoded in their ID's. Containment encoding makes index search operations very efficient because from the encoding of the smallest CES covering an object location, the encoding of all the other covering CES's can be easily derived. Moreover, containment encoding makes it easy to identify those VC's that need not be visited during an incremental query reevaluation. Simulations are conducted to evaluate CES-based indexing and compare it with a square-only VCR-based indexing for periodic query reevaluation. The results show that CES-based indexing substantially outperforms the square-only VCR-based indexing.

The paper is organized as follows. Section 2 describes related work. Section 3 briefly summarizes a prior VCR-based indexing method. Section 4 describes CES-based indexing method. We show the

definition of CES's, the decomposition and the search algorithms, and the incremental query reevaluation with containment-encoded squares. Section 5 shows the performance evaluation. Section 6 summarizes our paper.

2 Related Work

Query indexing was not used in the moving object environment until recently [11, 17, 25]. In [17], an R-tree-based query indexing method was first proposed for continual range queries over moving objects. A safe region for each mobile object was defined, allowing an object not to report its location as long as it has not moved outside its safe region. However, determining a safe region requires intensive computation. In [11], a cell-based query indexing scheme was proposed. It was shown to perform better than the R-tree-based query index [11]. The monitoring area is partitioned into cells. Each cell maintains two query lists: full and partial. The full list stores the IDs of the queries that completely cover the cell, while the partial list keeps those that partially intersect with the cell. However, using partial lists has a drawback. The object locations must be compared with the range query boundaries in order to identify those queries that truly contain an object. Because of that, it cannot allow query reevaluation to take advantage of the incremental changes in object locations.

In [25], a VCR-based query indexing method was presented for incremental processing of continual range queries over moving objects. It was shown to outperform the cell-based approach [11]. It is similar to the CES-based indexing method presented in this paper in that both use one or more virtual constructs to decompose the query regions. However, VCR-based query indexing is less efficient in query reevaluation. Many of the VCR's defined are redundant, unnecessarily degrading the index search and query reevaluation times. Without containment-encoding, it is also less efficient in identifying computations that can be avoided in incremental query reevaluation.

Although range queries can be treated as rectangles, traditional spatial indexing methods, such as R-trees [8], are not effective because they are mostly disk-based approach. As shown in [11], R-tree-based query indexing is not as effective as the cell-based approach, which is main-memory-based, even if it is modified for main memory access. Moreover, the performance of an R-tree quickly degenerates when the regions of range queries start to overlap one another [5, 9].

There are research papers focusing on other issues of moving object databases. For example, various indexing techniques on moving objects have been proposed [1, 2, 10, 12, 16, 20, 22]. The trajectories, the past, current, and the anticipated future positions of the moving objects and the parameters of the motion functions of the moving objects all have been explored for indexing. Different constraints are usually imposed to reduce the overhead caused by location updates. The data modeling issues of representing

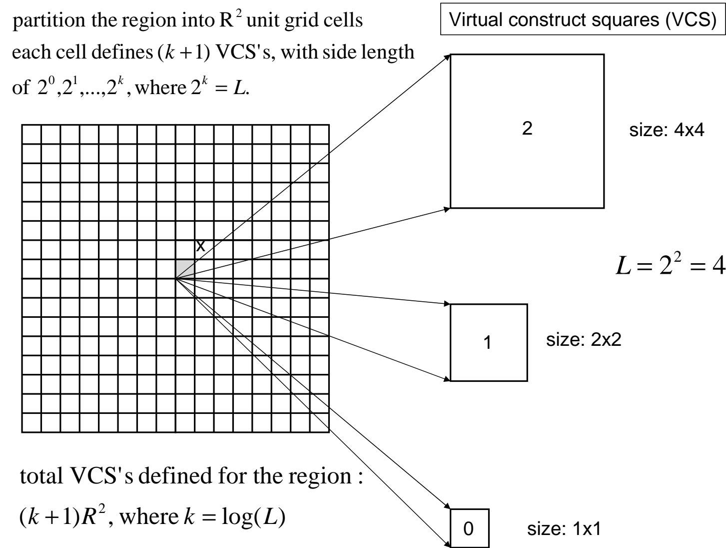

Figure 1: An example of virtual construct squares (VCS).

and querying moving objects were discussed in [4, 7, 19, 24]. Uncertainty in the positions of the moving objects was dealt with by controlling the location update frequency [23, 24], where objects report their positions when they have deviated from the last reported positions by a threshold. Partitioning the monitoring area into domains (cells) and making each moving object aware of the query boundaries inside its domain was proposed in [3] for adaptive query processing. Objects must report to the server when they move across query boundaries or domain boundaries. The more complex problem of locating moving objects when the continual range queries also move around was studied in [6, 13].

3 A VCR-based indexing approach

For comparison, we briefly describe a prior VCR-based indexing method that uses square-only virtual constructs [25]. Because it uses *virtual construct squares*, VCS, we refer it here as VCS-based indexing. For each integer grid point (a, b), where $0 \leq a, b < R$, a set of $k + 1$ virtual construct squares, or VCS's, are defined, where $L = 2^k$ is the maximal side length of a VCS. These $k + 1$ VCS's share the common bottom-left corner at (a, b) but have different sizes. For an R^2 monitoring region, the total number of VCS's defined is hence $(k + 1)R^2$. In contrast, there are $4R^2/3 - R^2/(3 * 4^k)$ CES's (see Property 1). More VCS's than CES's are defined. Fig. 1 shows an example of VCS's. There are 3 different sizes of VCS's: 1×1, 2×2 and 4×4.

Decomposition is relatively easy in VCS-based indexing, similar to covering a floor with square-only tiles of different sizes [25]. A CES with the largest possible size can be used to cover the query region,

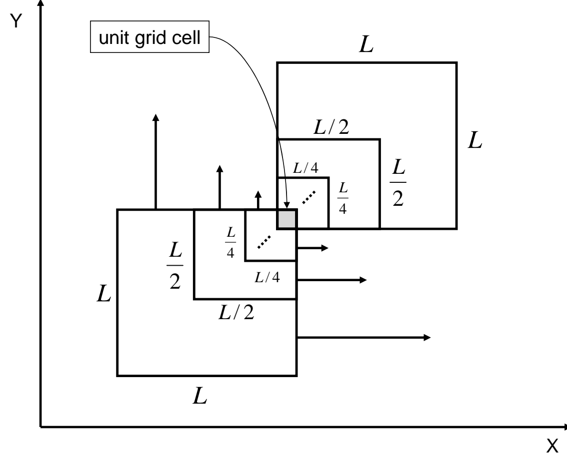

Figure 2: Covering VCS's that contain an object location within a unit grid cell.

beginning from the bottom-left corner and moving towards east and north.

The average index search time is slower in VCS-based indexing than in CES-based indexing. This is because the number of VCS's that can cover any object location in VCS-based indexing is $(4L^2-1)/3$, or $(4^{k+1}-1)/3$, significantly larger than $k+1$ for the CES-based indexing. This can be derived as follows. Consider the bottom-left VCS with size $L \times L$ that covers the unit grid cell in Fig. 2. We can move this $L \times L$ VCS eastwards along the X-axis and/or upwards along the Y-axis. There are a total of L^2 positions where the $L \times L$ VCS can be placed such that it still covers the unit grid cell. Similarly, for the VCS with size $L/2 \times L/2$, the number of positions is $(L/2)^2$. Hence, the number of covering VCS's is

$$L^2 + (L/2)^2 + \cdots + 1 = \sum_{i=0}^{k} (L/2^i)^2 = (4L^2 - 1)/3.$$

Now, we describe the incremental reevaluation algorithm using a VCS-based query index. Query results are maintained in an array of object lists, one for each query. Assume that $OL(q)$ denotes the object list for q. $OL(q)$ contains the IDs of all objects that are inside the boundaries of q. Periodically all $OL(q)$'s, $\forall q \in Q$, must be recomputed, taking into account the changes in object locations since the last reevaluation.

The pseudo code for Algorithm VCS_IR is described in Fig. 3. IR stands for Incremental reevaluation. The object locations used in the last reevaluation are assumed to be available. These locations are referred to as the *old* locations in contrast to the *new* locations for the current reevaluation. For each $o_i \in O$, if the location of o_i, denoted as $L(o_i)$, has not been updated since the last reevaluation, nothing needs to be done for this object. For an object whose location has been updated, we compute two covering-VCS sets: $CV_{new}(o_i)$ with the new location data and $CV_{old}(o_i)$ with the old location data.

```
Algorithm VCS_IR
  for (i = 0; o_i ∈ O; i++) {
    if (L(o_i) has not been updated) { continue; }
    compute CV_new(o_i); compute CV_old(o_i);
    for (k = 0; v_k ∈ CV_new(o_i) − CV_old(o_i); k++) {
      q = QL(v_k);
      while (q ≠ NULL) {
        insert(o_i, OL(q)); q = q → next; }
    }
    for (k = 0; v_k ∈ CV_old(o_i) − CV_new(o_i); k++) {
      q = QL(v_k);
      while (q ≠ NULL) {
        delete(o_i, OL(q)); q = q → next; }
    }
  }
```

Figure 3: Pseudo code for Algorithm VCS_IR.

When an object has moved, three cases need to be considered: (1) It has moved into a new VCS; (2) It has moved out of an old VCS; (3) It has remained inside the same old VCS. With both $CV_{new}(o_i)$ and $CV_{old}(o_i)$, we can easily identify the VCS's under each case. For any VCS v_k that is in the new covering VCS set but not the old, i.e., $v_k \in CV_{new}(o_i) - CV_{old}(o_i)$, we insert an instance of o_i to the $OL(q)$ list, $\forall q \in QL(v_k)$. Here, $QL(v_k)$ is the query list associated with VCS v_k. This accounts for the case that o_i has moved into these VCS's. On the other hand, for a VCS v_j that is in the old covering VCS set but not the new, i.e., $v_j \in CV_{old}(o_i) - CV_{new}(o_i)$, we delete an instance of o_i from $OL(q)$ list, $\forall q \in QL(v_j)$. This accounts for the case that o_i has moved out of these VCS's. For any VCS that is in both covering VCS sets, nothing needs to be done. It accounts for the case that o_i has remained inside the boundaries of these VCS's.

Note that both $CV_{new}(o)$ and $CV_{old}(o)$, $\forall o \in O$, must be completely computed in VCS_IR. This makes VCS_IR less efficient than algorithm CES_IR to be described in Fig. 9.

4 CES-based query indexing

4.1 System model

Similar to [25], we assume that there is a monitoring region where moving objects are tracked. The region is partitioned into $R_x R_y$ virtual grids. Without loss of generality, we assume $R_x = R_y = R$. The grid coordinates are used to specify range queries and moving objects in terms of positions and boundaries. Range queries are specified as rectangles defined along the grid lines. Namely, query boundaries are specified with integer grid coordinates [25]. However, object locations can be anywhere. We assume that

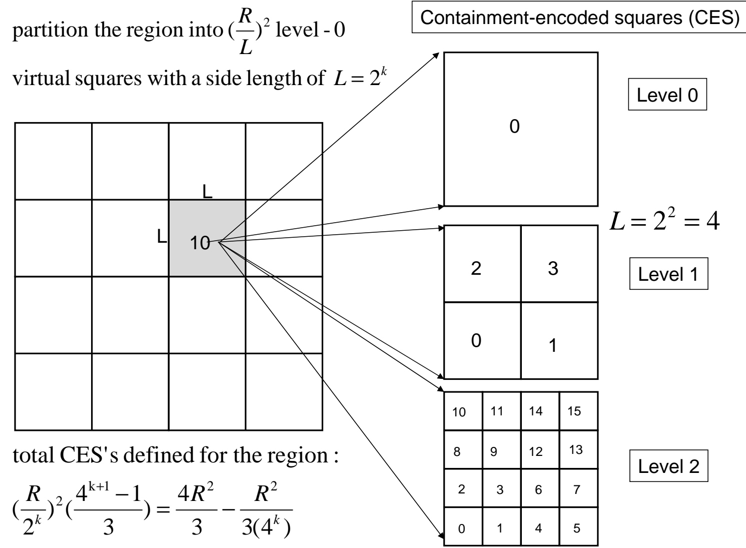

Figure 4: An example of containment-encoded squares (CES).

continual range queries are stationary, but they can be inserted or deleted dynamically. Objects move continuously. An object may report its position back to the query processor periodically or when the position change is greater than a threshold [24]. Alternatively, position-sensing device may be employed to track object positions.

4.2 Containment-encoded squares (CES)

Fig. 4 shows an example of virtual containment-encoded squares and their ID labeling. Without loss of generality, we assume that $R = 2^r$, where r is some integer. The CES's are defined as follows. First, we partition the entire $R \times R$ monitoring area into $(R/L)^2$ virtual square partitions, each of size $L \times L$. Here, we assume that $L = 2^k$ and L is the maximal side length of a CES. The $L \times L$ squares are called level-0 virtual squares. Then, we create k additional levels of virtual squares. Level-1 virtual squares are created by partitioning each level-0 virtual square into 4 equal-sized $L/2 \times L/2$ virtual squares . Level-2 virtual squares are created by partitioning each level-1 virtual squares into 4 equal-sized $L/4 \times L/4$ virtual squares. Level-k virtual squares have unit side length, i.e., 1×1.

The total number of CES's defined within each level-0 virtual square, including itself, is $\sum_{i=0}^{i=k} 4^i = (4^{k+1}-1)/3$. These virtual squares are defined to have containment relationships among them in a special way. Every unit-sized CES is contained by a CES of size 2×2, which is in turn contained by a CES of size 4×4, which is in turn contained by a CES of size 8×8, $\cdots$, and so on.

Property 1 *The total number of CES's defined in a $R \times R$ monitoring region is* $(\frac{R}{L})^2 \sum_{i=0}^{i=k} 4^i = \frac{4R^2}{3} - \frac{R^2}{3(4^k)}$.

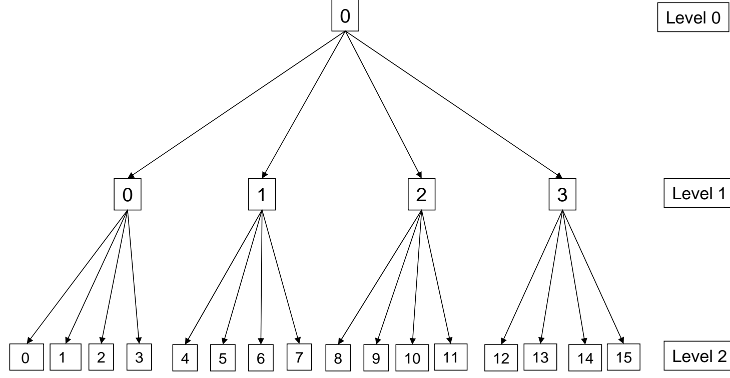

Figure 5: An example of a perfect quaternary tree and its containment-encoded labeling.

Within each level, the ID of a virtual square consists of two parts: a partition ID and the local ID within the partition. If a virtual square has a partition ID p and local ID z_i, then its unique ID c_i at level i, where $0 \leq i \leq k$, can be computed as follows:

$$c_i = 4^i p + z_i.$$

This is because there are 4^i CES's within each partition at level i. The partition ID can be computed as the row scanning order of the level-0 CES's starting from the bottom row and moving upwards. For example, for a level-0 CES (a, b, L, L), where (a, b) is the bottom-left corner and L is the side length, its partition ID can be computed as follows:

$$P(a, b, L, L) = \frac{a}{L} + (\frac{b}{L})(\frac{R}{L}).$$

The labeling of local CES IDs within a partition follows that of a perfect quaternary tree as shown in Fig. 5, where the IDs of the four child squares are $4s, 4s + 1, 4s + 2$ and $4s + 3$ if the parent has a local ID s. In order to preserve containment relationships between virtual squares at different levels, the CES IDs within the same partition at each level follow the z-ordering space-filling curve, or Morton order [15, 14, 18]. For example, in Fig. 4, the IDs for the 16 level-2 virtual squares for partition 10 follow the z-ordering space-filling curve. In general, the local IDs of $4s, 4s + 1, 4s + 2$ and $4s + 3$ are assigned to the southwest, southeast, northwest and northeast children, respectively, of a parent virtual square with a local ID s.

Property 2 *For any CES at level i with a local ID z_i, where $0 < i \leq k$, the local ID of its parent can be computed by $\lfloor z_i/4 \rfloor$, or a logical right shift by 2 bits of the binary representation of z_i.*

Property 3 *The total number of CES's that can possibly cover/contain any given data point within the monitoring region is $k + 1$, or $\log(L) + 1$.*

Note that we can also view the entire predefined CES's as $k+1$ levels of overlapping square grid cells where each cell at level i contains exactly 4 cells at level $i+1$, where $0 \leq i < k$. Hence, there are exactly $k+1$ CES's that cover any given data point within the monitoring area.

4.3 Decomposition algorithm

Fig. 6 shows the pseudo code for decomposing a rectangle range query $q = (a, b, w, h)$, where (a, b) is the bottom-left corner and w and h are the width and height, respectively, of the range query, into one or more CES's. It is a modification of a strip-splitting-based optimal algorithm for decomposing a query window into maximal quad-tree blocks [21]. The difference is that the algorithm in [21] allows m to be as large as $\log(R)$, assuming that $R = 2^r$, r is some integer, and R is the side length of the monitoring area. In contrast, we only allow m to be as large as $L = 2^k$, the maximal side length of a CES. The decomposition algorithm performs multiple iterations of 4 strip-splitting processes. During each iteration it tries, if possible, to strip away from q a column strip or a row strip of width or height of $m = 2^i$, where $0 \leq i < k$, from each of the four outside layers of q, starting with $i = 0$. The column strip or row strip is then split or decomposed into multiple $m \times m$ square blocks. The goal is to use minimal number of maximal-sized CES's to decompose q. The entire strip-splitting process is like peeling a rectangular onion from the outside. The width of each layer at each successive iteration is doubled until it reaches L. After that, it decomposes the remaining q using $L \times L$ CES's.

During each iteration, the rule to determine if there is any strip of width or height 2^i that can be removed from the remaining q is based on the bottom-left corner, width and height of q [21]. Assume that the current remaining q is denoted as (a', b', w', h'), if $(a' \mod 2^{i+1}) \neq 0$, then a column strip of width 2^i, where $0 \leq i < k$, can be removed from the leftmost of q. If $((b' + h') \mod 2^{i+1}) \neq 0$, then a row strip of height 2^i can be removed from the topmost of q. If $((a' + w') \mod 2^{i+1}) \neq 0$, then a column strip of width 2^i can be stripped from the rightmost of q. Finally, if $(b' \mod 2^{i+1}) \neq 0$, then a row strip of height 2^i can be removed from the bottommost of q.

As an example, Fig. 7 shows the step-by-step decomposition of a range query q defined as $(5, 2, 8, 12)$. (1) A column strip $(5, 2, 1, 12)$ of width 1 is removed from the leftmost outside of q because $(5 \mod 2) \neq 0$. The remaining q becomes $(6, 2, 7, 12)$. The column strip is split into 12 CES's of size 1×1. (2) A column strip $(12, 2, 1, 12)$ of width 1 is removed from the rightmost outside of the remaining q because $((6 + 7) \mod 2) \neq 0$. This column strip is split into 12 CES's of size 1×1. The remaining q becomes $(6, 2, 6, 12)$. (3) A column strip $(6, 2, 2, 12)$ of width 2 is removed from the leftmost outside of q because $(6 \mod 4) \neq 0$. The remaining q becomes $(8, 2, 4, 12)$. (4) A row strip $(8, 12, 4, 2)$ of height 2 is removed from the topmost outside of q because $((2 + 12) \mod 4) \neq 0$. The remaining q becomes

```
Decomposition (a, b, w, h) {
  m = 1; q = (a, b, w, h) ;
  while ((q ≠ NULL) ∧ (m < L)) {
    strip from q the leftmost column strip with width m,
    if any, and split the column strip with m × m CES's;

    strip from q the topmost row strip with height m,
    if any, and split the row strip with m × m CES's;

    strip from q the rightmost column strip with width m,
    if any, and split the column strip with m × m CES's;

    strip from q the bottommost row strip with height m,
    if any, and split the row strip with m × m CES's;

    m = m × 2;
  }
  if (q ≠ NULL) {
    decompose q with CES's of size L × L;
  }
}
```

Figure 6: Pseudo code for decomposition algorithm with CES's.

$(8, 2, 4, 10)$. (5) A row strip $(8, 2, 4, 2)$ of height 2 is removed from the bottommost outside of q because $(2 \mod 4) \neq 0$. The remaining q becomes $(8, 4, 4, 8)$. (6) Finally, $(8, 4, 4, 8)$ is decomposed into two 4×4 CES's and the remaining q becomes NULL.

4.4 Search algorithm

After decomposition, the query ID is inserted into the ID lists associated with decomposed CES's. Assume that $QL(l, c)$ denotes the query ID list associated with a level-l CES with a local ID c. These query ID lists contain indirectly pre-computed search results. Namely, the queries containing a CES are all stored in the associated query ID list. To find the queries covering an object location, we first find the covering CES's and then the covering queries.

For a given data point (x, y), the search algorithm finds the $k + 1$ CES's that contain or cover (x, y). Fig. 8 shows the pseudo code for a bottom-up search algorithm. It first finds the partition ID and the local ID of the level-k CES that contains (x, y). Let p denote the partition ID and z denote the local ID of the covering CES at level k. The unique ID of the covering CES at level k is $4^k p + z$. From Property 2, the local ID at level $k - 1$ can be easily computed by dividing z by 4 because of containment encoding. This can be implemented by a logical right shift by 2 bits. As a result, the entire search operation is extremely

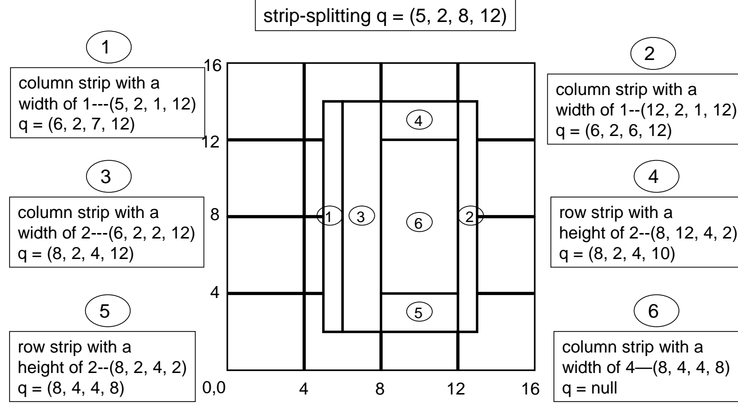

Figure 7: An example of strip-splitting-based decomposition with CES's.

Bottom-up Search(x, y) {
  $I_x = \lfloor x \rfloor$; $I_y = \lfloor y \rfloor$;
  $P_x = \lfloor I_x/L \rfloor$; $P_y = \lfloor I_y/L \rfloor$;
  // (LP_x, LP_y) is the partition bottom-left corner
  $p = P_x + P_y(R/L)$; // partition ID
  $z = Z(I_x - LP_x, I_y - LP_y, 2^0)$;
  // local ID of CES $(I_x, I_y, 1, 1)$
  for $(l = k; l \geq 0; l = l - 1)$ {
    $c = 4^l p + z$; // covering CES ID at level l
    if ($QL(l, c) \neq$ NULL) { output($QL(l, c)$); }
    $z = z/4$; // right shifts by 2 bits
  }
}

Figure 8: Pseudo code for a bottom-up search algorithm with CES-based indexing.

efficient.

Note that even though we partition each virtual square at level i into 4 equal-sized quadrants at level $i + 1$, similar to the quad-tree space partition, the bottom-up search algorithm described in Fig. 8 makes the CES-based query index unique. It achieves efficient search by taking advantage of the containment encoding embedded in the local IDs of virtual squares at different levels.

4.5 Query reevaluation with CES-based indexing

Because many objects might not have moved outside some CES boundaries since the last evaluation, the computation should be done incrementally. Containment encoding in the CES's makes it easy to identify the CES's that need not be visited during an incremental re-computation.

```
Algorithm CES_IR
  for (j = 0; o_j ∈ O; j++) {
    if (L(o_j) has not been updated) { continue; }
    p_old = P(L_old(o_j)); p_new = P(L_new(o_j));
    z_old = local ID of the unit CES covering L_old(o_j));
    z_new = local ID of the unit CES covering L_new(o_j));
    if (p_old ≠ p_new) {
      for (l = k; l ≥ 0; l--) {
        c_new = 4^l * p_new + z_new;
        insert o_j into OL(q), ∀q ∈ QL(l, c_new);
        c_old = 4^l * p_old + z_old;
        remove o_j from OL(q), ∀q ∈ QL(l, c_old);
        z_old = z_old/4; z_new = z_new/4; } }
    else {
      for (l = k; l ≥ 0; l--) {
        c_new = 4^l * p_new + z_new;
        c_old = 4^l * p_old + z_old;
        if (c_new ≠ c_old) {
          insert o_j into OL(q), ∀q ∈ QL(l, c_new);
          remove o_j from OL(q), ∀q ∈ QL(l, c_old);
          z_old = z_old/4; z_new = z_new/4; }
        else break;
      }
    }
  }
```

Figure 9: Pseudo code for Algorithm CES_IR.

The pseudo code for Algorithm CES_IR is described in Fig. 9. For each $o_j \in O$, denoting the set of all moving objects, if the location of o_j, denoted as $L(o_j)$, has not been updated since the last reevaluation, nothing needs to be done for this object. For an object whose location has been updated, we first compute the partition ID's of the old and new locations, denoted as p_{old} and p_{new}, respectively.

Depending on whether or not p_{new} and p_{old} are the same, some computation can be saved. If they are not the same, the object has since moved into a different partition. In this case, no computation can be saved. We need to insert o_j into and remove o_j from all the $OL(q)$'s for queries contained in the query ID lists associated with the CES's that cover the new and old locations, respectively. On the other hand, if p_{new} and p_{old} are the same, some CES's in the same partition may contain both the old and new locations. Hence, no action is needed for these CES's. Due to containment encoding, these CES's that contain both the old and the new locations can be easily identified by their local ID's. If z_{old} equals z_{new} for the level-l CES, then the computation can be saved for CES's from level-0 to level-l.

5 Performance evaluation

5.1 Simulation studies

Simulations were conducted to evaluate and compare CES-based indexing with the VCS-based indexing for periodic reevaluations of continual range queries over moving objects. Since it has been shown in [25] that the VCS-based indexing approach outperforms other query indexing schemes, such as the cell-based approach in [11], we focus in this paper on comparing CES-based indexing with the VCS-based indexing.

For the simulations, the monitoring region was defined by $R_x = R_y = 512$ grid units. A continual range query was represented as a rectangle with width of W_x and height W_y. Both W_x and W_y were randomly and independently chosen between 1 and W. W were varied from 30 to 80. The bottom-left corner of a range query was chosen uniformly within the monitoring area. The maximum side length of a VCS or CES $L = 2^k$ and k is an integer.

A total number of $|Q|$ continual range queries were inserted into the query index. A total of $|O|$ objects were generated. The initial locations of these objects were uniformly distributed within the monitoring area. Their subsequent locations were calculated based on the following rule. We define M as the maximal horizontal or vertical movement in terms of virtual grids between two consecutive reevaluations. The new location of a moving object was calculated based on its old location and the horizontal and vertical movements, which were independently chosen for directions and magnitudes. Namely, if an object was at (x, y), then its new location at the next reevaluation was at $(x + d_x \triangle_x, y + d_y \triangle_y)$, where d_x and d_y were equally likely to be 1 or -1 and $\triangle_x$ and $\triangle_y$ were independently and uniformly chosen from $[0, M]$. Query results were first computed with the initial object locations. Then, the locations were updated based on the movements defined by M. Afterwards, a query reevaluation was performed. We measured the time it took to complete the reevaluation and the total storage cost for the query index. We assumed that there were no changes to the query index between two query reevaluations. We conducted our simulations on an IBM ThinkPad T30 model (CPU 2.4 GHz; memory size 512 Mbytes) running cygwin under Windows XP.

5.2 Impact of L on index storage cost and query reevaluation time

The maximal side length L of a virtual construct impacts both the index storage cost and continual query reevaluation time. Here we examine the impacts of different L's, ranging from 4 to 64, under both a CES-based and a VCS-based index. For this experiment, $W = 80$, $|Q| = 8,000$, $|O| = 50,000$ and $M = 1$.

From Fig. 10(a), as L increases, the index storage cost steadily decreases for the CES-based indexing. In contrast, it decreases first and then increases for the VCS-based indexing. Total index storage costs are

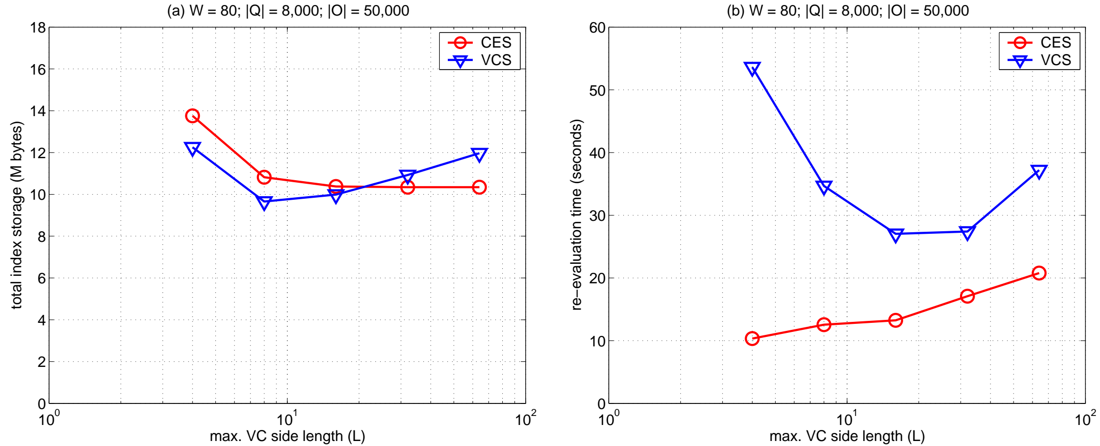

Figure 10: The impact of L on (a) total index storage and (b) reevaluation time.

comparable for both indexing schemes. The index storage cost consists of two components: (a) the query ID lists, one for each VCS or CES, and (b) an array list of pointers to the query ID lists. For both schemes, component (a) generally decreases as L increases because fewer VC's are needed to cover a range query. As L increases, Component (b) increases much faster for the VCS-based indexing than the CES-based indexing. This is because the total number of VC's defined is $(k+1)R^2$ for the VCS-based indexing, but it is $4R^2/3 - R^2/(3 * 4^k)$ for the CES-based indexing, where $k = \log(L)$.

Fig. 10(b) shows the the average query reevaluation time. Here, the performance advantage of the CES-based indexing over the VCS-based indexing is clearly observed for the entire range of L's. This is because the number of VC's visited during an index search is at most $\log(L) + 1$ for the CES-based indexing, compared with $(4L^2 - 1)/3$ for the VCS-based indexing.

5.3 Comparisons of CES and VCS

Now we compare CES-based with VCS-based indexing under various numbers of continual queries and moving objects. We focus on the query reevaluation time because the total index storage costs are comparable for both schemes.

Figs. 11(a) shows the impact of $|Q|$, the number of continual range queries, on the reevaluation time. For this experiment, $W = 50$, $L = 16$ and $|O| = 50,000$. We varied $|Q|$ from 1,000 to 16,000. Both $M = 1$ and $M = 10$ were used. $M = 1$ represents the scenario where most objects have not moved too far from their old locations since the last evaluation. In contrast, $M = 10$ represents the scenario where most objects have moved far away from their old locations since the last evaluation. More computation can be saved for the case of $M = 1$. CES-based indexing outperforms VCS-based indexing for all cases.

Figs. 11(b) shows the impacts of $|O|$, the number of moving objects, on the query reevaluation time.

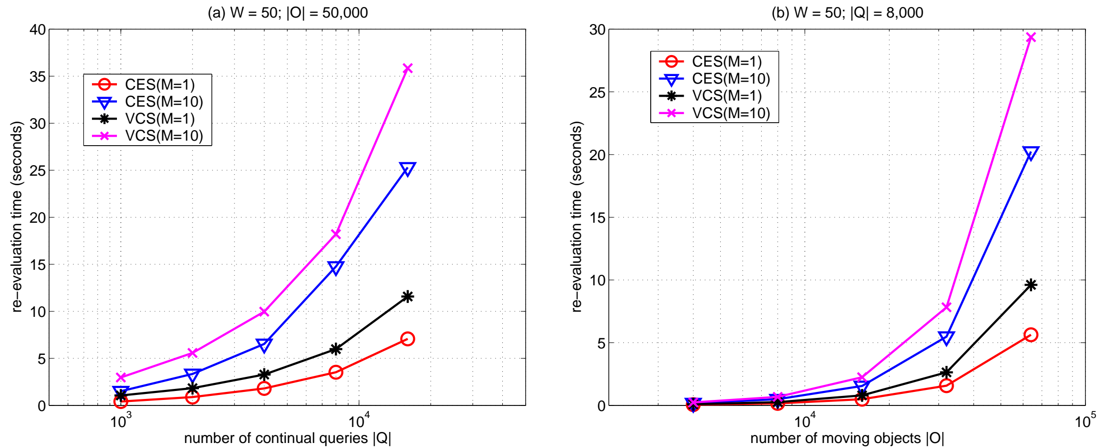

Figure 11: The impact of (a) $|Q|$ and (b) $|O|$, respectively, on reevaluation time.

For this experiment, $W = 50, L = 16$ and $|Q| = 8000$. $|O|$ was varied from 4,000 to 64,000. Again, CES-based indexing outperforms VCS-based indexing in query reevaluation time. Such performance advantage becomes more prominent as the number of moving objects increases.

6 Summary

Efficiently locating moving objects is critically important in supporting many location-based services and applications. We have presented a new CES-based query index for incremental processing of continual range queries over moving objects to locate up-to-date locations of these moving objects. A set of containment-encoded squares (CES) is predefined, each with a unique ID. CES's are virtual constructs used to cover each query region and to store indirectly pre-computed query results. The use of CES's provides fast search operations. More importantly, it makes it easy to identify the moving objects that need not be evaluated during a periodic query reevaluation. As a result, incremental processing of continual range queries is efficient. Simulations have been conducted to evaluate and compare CES-based indexing with a prior VCS-based indexing scheme. The results show that (1) the CES-based indexing has comparable storage cost with the VCS-based indexing and (2) the CES-based indexing substantially outperforms the VCS-based indexing in query reevaluation time, particularly if the number of moving objects is large.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving objects. In *Proc. of ACM PODS*, 2000.

[2] C. C. Aggarwal and D. Agrawal. On nearest neighbor indexing of nonlinear trajectories. In *Proc. of ACM PODS*, 2003.

[3] Y. Cai and K. A. Hua. An adaptive query management technique for real-time monitoring of spatial regions in mobile database systems. In *Proc. of Int. Performance, Computing and Communication Conf.*, 2002.

[4] L. Forlizzi, R. H. Guting, E. Nardelli, and M. Scheider. A data model and data structures for moving objects. In *Proc. of ACM SIGMOD*, 2000.

[5] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[6] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Motion adaptive indexing for moving continual queries over moving objects. In *Proc. of ACM CIKM*, 2004.

[7] R. H. Guting, M. H. Bohlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. on Database Systems*, 25(1):1–42, Mar. 2000.

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD*, 1984.

[9] E. Hanson and T. Johnson. Selection predicate indexing for active databases using interval skip lists. *Information Systems*, 21(3):269–298, 1996.

[10] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B+-tree based indexing of moving objects. In *Proc. of VLDB*, 2004.

[11] D. V. Kalashnikov, S. Prabhakar, W. G. Aref, and S. E. Hambrusch. Efficient evaluation of continuous range queries on moving objects. In *Proc. of DEXA*, 2002.

[12] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. of ACM PODS*, 1999.

[13] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proc. of ACM SIGMOD*, 2004.

[14] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. of ACM SIGMOD*, 1986.

[15] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proc. of ACM PODS*, April 1984.

[16] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *Proc. of VLDB*, 2000.

[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*, 51:1124–1140, Oct. 2002.

[18] H. Samet. *Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[19] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proc. of IEEE ICDE*, 1997.

[20] Y. Tao, D. Papadias, and Q. Shen. The TPR*-Tree: An optimized spatio-temporal access method for predictive queries. In *Proc. of VLDB*, 2003.

[21] Y.-H. Tsai, K.-L. Chung, and W.-Y. Chen. A strip-splitting-based optimal algorithm for decomposing a query window into maximal quadtree blocks. *IEEE TKDE*, 16(4):519–523, Apr. 2004.

[22] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc. of ACM SIGMOD*, 2000.

[23] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *Proc. of IEEE ICDE*, 1998.

[24] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[25] K.-L. Wu, S.-K. Chen, and P. S. Yu. Processing continual range queries over moving objects using VCR-based query indexes. In *Proc. of IEEE MobiQuitous*, Aug. 2004.