Incremental Processing of Continual Range Queries over Moving Objects

Kun-Lung Wu, *Senior Member*, *IEEE*, Shyh-Kwei Chen, *Member*, *IEEE*, and Philip S. Yu, *Fellow*, *IEEE*

Abstract—Efficient processing of continual range queries over moving objects is critically important in providing location-aware services and applications. A set of continual range queries, each defining the geographical region of interest, can be periodically (re)evaluated to locate moving objects that are currently within individual query boundaries. 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. CESs are *virtual constructs* (VC) used to decompose query regions and to store indirectly precomputed search results. Compared with a prior VC-based approach, the number of VCs visited in a search operation 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 CESs makes it easy to identify all those VCs that need not be visited during an incremental query (re)evaluation. We study the performance of CES-based indexing and compare it with a prior VC-based approach.

Index Terms—Query indexing, moving objects, mobile computing, location-aware applications, continual range queries.

1 Introduction

Location-aware services and applications have become possible with the advances in mobile computing and location-sensing technologies, such as the global positioning systems (GPS) and RFID systems. Essentially, any object that moves or is moved can be equipped with a location-awareness device and monitored. Such location-aware applications can be used to deliver to target customers relevant, timely, and engaging content and information. For example, a retail store in a mall can send timely e-coupons to the PDAs or cell-phones of potential customers who are close to the store. A taxi company can quickly dispatch a moving taxi cab to a nearby customer who needs a service once the company receives the service request.

To provide location-based services and applications, one must first know where moving objects are currently located. Object locations can be periodically reported by the mobile objects individually back to a central server, or when the current locations deviate from the last reported positions by a threshold [30], [31]. Alternatively, position-sensing devices can be employed to tract and scan object positions, such as the GPS system or RFID devices.

With object positions, a set of continual range queries, each defining the geographical regions of interest, can be repeatedly (re)evaluated to locate moving objects that are within the query boundaries. For example, we can define a continual range query by placing a square or a circle around the location of a hotel, an apartment building, or a subway exit. The query result then contains all the moving objects that are currently located within its boundaries. Efficient processing of a large number of such continual range queries over moving objects is critically important to provide location-aware services and applications.

• *The authors are with the IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532. E-mail: {klwu, skchen, psyu}@us.ibm.com.*

Manuscript received 25 July 2005; revised 4 Jan. 2006; accepted 23 June 2006; posted online 19 Sept. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0283-0705.

Depending on whether or not queries also move, the processing of continual range queries over moving objects can be roughly classified into two categories. The first category deals with stationary queries over moving objects, e.g., [3], [15], [22], [32], [33], [34], and the second category deals with moving queries over moving objects, e.g., [8], [7], [17]. 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 moving taxi cab can be used to find the customers who are close to the taxi at a given moment. In this paper, however, we focus on stationary queries where a query region is associated with a stationary object, such as a landmark. However, queries can be added or removed dynamically. The periodically (re)evaluated continual query results are used by service providers to offer location-aware services and applications.

Query indexing has been used to speed up the processing of continual range queries over moving objects [15], [22], [32], [33], [34]. Because queries change less frequently than object positions do, it is less costly to maintain a query index than an object index. 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.

Note that, because objects continue to move nonstop while range queries are (re)evaluated only periodically, there are inevitable inaccuracies in the reported query results. The number of errors in the query results generally increases as the time interval between two consecutive (re)evaluations lengthens. In order to minimize the inherent inaccuracies, it is paramount that the time it takes to perform the periodic query (re)evaluation be as brief as possible so that query (re)evaluations can be performed as frequently as possible.

In [22], 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 [15], a cell-based query indexing scheme was proposed for evaluating continual range queries over moving objects. It was shown to perform better than the R-tree-based query index [15]. 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 (re)evaluation to take advantage of the incremental changes in object locations, making it more difficult to conduct frequent query (re)evaluations.

To allow for incremental query (re)evaluation, various kinds of virtual constructs (VCs) have been proposed for building query indexes, including shingles [34], covering tiles [33] and virtual construct rectangles (VCRs) [32]. The VCR-based query indexing was main memory based and was shown to outperform other query indexing approaches, such as the cell-based [15], the shingle-based [34], and the covering tile-based [33] approaches, in terms of total query (re)evaluation time [32]. It uses a set of predefined VCRs to decompose query regions. Search operations are conducted indirectly via VCRs. However, many of the VCRs are redundant, unnecessarily slowing down the index search time and the query (re)evaluation time.

In this paper, we propose a new query indexing method, called CES-based indexing, to further minimize the total query (re)evaluation time so that the accuracy of the query results can be additionally improved via more frequent (re)evaluations. CES stands for *containment-encoded squares*. Similar to VCRs, CESs are virtual constructs used to decompose query regions and to store precomputed search results. However, the set of predefined virtual constructs, the decomposition algorithm and the search algorithm are all different. There are fewer CESs defined than VCRs. Search time is significantly lowered. More importantly, there are containment relationships among the CESs. Containment encoding makes index search operations very efficient. Simulations are conducted to evaluate CES-based indexing. The results show that CES-based indexing substantially outperforms a square-only VCR-based indexing in terms of total query (re)evaluation time.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 describes the preliminaries, including the system model and a generic, 2D query index method based on virtual constructs for evaluating continual range queries over moving objects. Section 4 presents CES-based indexing method. Section 5 briefly summarizes a prior VCR-based indexing method. Section 6 shows performance evaluation. Section 7 concludes our paper.

2 Related Work

Although range queries can be treated as rectangles, traditional spatial indexing methods, such as R-trees [10], are not effective because they are mostly disk-based approaches. As shown in [15], 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 [6], [11].

There are many research papers focusing on other issues of moving object databases. For example, various indexing techniques on moving objects have been proposed [1], [2], [14], [16], [20], [21], [27], [25], [29]. 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 and querying moving objects were discussed in [5], [9], [24]. [31]. Uncertainty in the positions of the moving objects was dealt with by controlling the location update frequency [30], [31], 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], [4] 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 [8], [17]. Enabling mobile clients to determine if the previous query results are still valid based on the current locations was proposed in [37]. A generic framework for monitoring continuous spatial queries over moving objects was presented in [12], aiming at cutting down the communication as well as query (re)evaluation costs due to location updates. Many papers have focused on k-nearest neighbor queries [13], [26], [36], [35], which belong to another kind of spatial queries. In contrast, we focus on continual range queries in the present paper.

3 Preliminaries

3.1 System Model

Table 1 summarizes important notations and definitions used in this paper. 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 [32]. However, object locations can be anywhere. We assume that continual range queries are stationary, but they can be inserted or deleted dynamically. Objects move continuously.

Note that we can model the monitoring region with various resolutions by assigning each unit grid with

TABLE 1
Notations and Definitions

Notation	Definition
VCR	virtual construct rectangle
VCS	virtual construct square
CES	containment-encoded square
VC	virtual construct
R_x (R_y)	width (height) of the monitoring region (unit: grids; default: R)
d	physical distance for each virtual grid (unit: mile)
$o_1, o_2, \cdots$	IDs for moving objects
$q1, q2, \cdots$	IDs for continual range queries
$v1, v2, \cdots$	IDs for virtual constructs
z_i	local ID of a CES in level i
$QL(\cdot)$	query ID list (one for each VC)
$OL(\cdot)$	object ID list (one for each query)
L	maximum side length of a VCR, VCS or CES
Q	the set of all continual range queries
O	the set of all moving objects
CV_{old} (CV_{new})	the set of covering VCs for the old (new) location
M	maximum horizontal/vertical movement of an object between two consecutive (re)evaluations (unit: grids)
W	maximum width (heights) for a range query (unit: grids)
m	the width or height of a strip in the decomposition algorithm
α, β	skewness in query positions
(a, b, w, h)	a range query with bottom-left corner at (a, b), width w and height h

differing physical distances. Let d denote the physical distance for the side length of a virtual grid cell. If $d = 1/10$ mile, a 50×50 square-mile monitoring region can be represented by 500×500 virtual grids. On the other hand, if $d = 1/2$ mile, the same 500×500 grids represent a 250×250 square-mile physical region. We assume d is chosen such that all range queries can be approximately and satisfactorily specified with integer grid coordinates.[1] If query boundaries need to be specified with a higher resolution, a smaller d must be used. With a small d, $R_x R_y$ becomes large for the same physical area. In this case, the entire $R_x R_y$ grid region can be partitioned into multiple smaller regions. For each small region, one processor can then be used to monitor and process the continual range queries.

3.2 A Generic, 2D, VC-Based Query Index for Query Processing

There are three key challenges for a virtual construct-based range query index for processing continual range queries over moving objects: 1) defining and labeling a set of virtual constructs, 2) decomposing each range query into one or more virtual constructs, and 3) performing search indirectly via computing the IDs of virtual constructs containing a given object position.

Fig. 1 shows an example of processing continual range queries using a generic, 2D, virtual construct-based query index. There are three range queries, $q1$, $q2$, and $q3$ that are decomposed with five virtual constructs, or VCs, $v1$, $v2$, $v3$, $v4$, and $v5$. Each range query is first decomposed into one or more VCs, and then the query ID is inserted into the query ID lists associated with the decomposed VCs. For example, $q2$ is decomposed into $v3$ and $v4$. Hence, the ID $q2$ is inserted into the query ID lists associated with $v3$ and $v4$ in Fig. 1. The query index is implemented with a pointer array and dynamically maintained query ID lists, one for each VC. A search is conducted indirectly via the VCs. For any object position, the IDs of the VCs that cover that object position are first identified. Then, the search result can be obtained from the associated query ID lists. For example, in Fig. 1, we can compute the covering virtual constructs, $v1$ and $v3$, for an object o_2. From $v1$ and $v3$, we can find the search result of $q1$, $q2$, and $q3$. Assume that query results are maintained in an array of object lists, $OL(\cdot)$, one for each

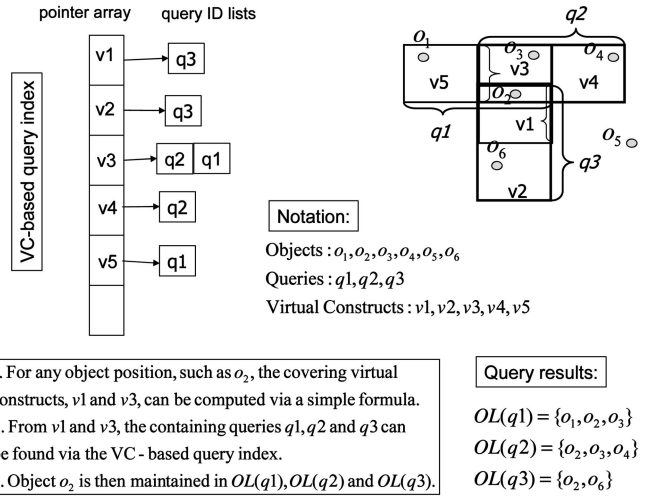


Fig. 1. Processing range queries over moving objects using a generic, 2D, VC-based query index.

1. Note that, even with a carefully chosen d, range queries might still be represented by nonintegers. In such cases, we can expand them to the nearest integers. The CES-based query index can still be used to first efficiently identify a set of candidates. Extra checking might be needed to find the final results from the candidates.

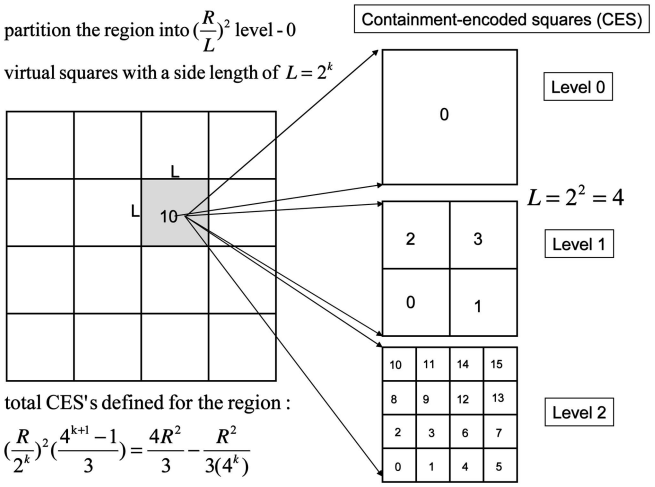


Fig. 2. An example of containment-encoded squares (CES).

query. Object ID o_2 is hence inserted into the query result lists of $OL(q1)$, $OL(q2)$, and $OL(q3)$.

4 Query Processing with a CES-Based Query Index

4.1 Containment-Encoded Squares (CES)

Fig. 2 shows an example of virtual containment-encoded squares and their ID labeling. The CESs 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 four equal-sized $L/2 \times L/2$ virtual squares. Level-2 virtual squares are created by partitioning each level-1 virtual squares into four 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 CESs 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 $8 \times 8, \cdots$, etc.

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

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 CESs within each partition at level i. The partition ID can be computed as the row scanning order of the level-0 CESs starting from the bottom row and moving upwards. For example, for a level-0 CES

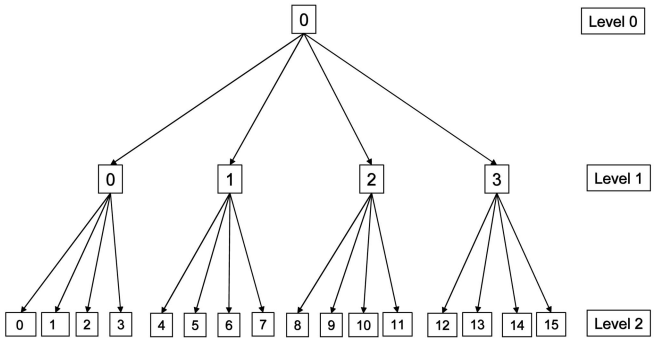


Fig. 3. An example of a perfect quaternary tree and its containment-encoded labeling.

(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:

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

The labeling of local CES IDs within a partition follows that of a perfect quaternary tree as shown in Fig. 3, 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 [19], [18], [23]. For example, in Fig. 2, 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 two bits of the binary representation of z_i.*

Property 3. *The total number of CESs 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 CESs as $k + 1$ levels of overlapping square grid cells where each cell at level i contains exactly four cells at level $i + 1$, where $0 \leq i < k$. Hence, there are exactly $k + 1$ CESs that cover any given data point within the monitoring area.

4.2 Decomposition Algorithm

Fig. 4 shows the pseudocode 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 CESs. It is a modification of a strip-splitting-based, optimal algorithm for decomposing a query window into maximal quad-tree blocks [28]. The difference is that the algorithm in [28] allows m, the width or height of a strip, to be as large as 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

```
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;
  }
}
```

Fig. 4. Pseudocode for decomposition algorithm with CESs.

of four 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 CESs 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 $L \times L$ CESs.

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 [28]. Assume that the current remaining q is denoted as (a', b', w', h'), if $(a' \bmod 2^{i+1}) \neq 0$, then a column strip of width 2^i, where $0 \leq i < k$, can be removed from the leftmost of q. If $((b' + h') \bmod 2^{i+1}) \neq 0$, then a row strip of height 2^i can be removed from the topmost of q. If $((a' + w') \bmod 2^{i+1}) \neq 0$, then a column strip of width 2^i can be stripped from the rightmost of q. Finally, if $(b' \bmod 2^{i+1}) \neq 0$, then a row strip of height 2^i can be removed from the bottommost of q. As an example, Fig. 5 shows the step-by-step decomposition of a range query q defined as (5, 2, 8, 12).

Lemma 1. *The remaining q out of the loop of the decomposition algorithm in Fig. 4 must be either null or have size $m_1 L \times m_2 L$, where m_1 and m_2 are positive integers.*

Proof. Due to the strip and splitting processes, there may be no remaining q, i.e., q is empty. Otherwise, the remaining q must have both width and height greater than or equal to L. Hence, there is no maximal virtual square smaller than $L \times L$ that can be used to decompose the remaining q. Hence, both the width and height of the remaining q must be multiples of L. □

Lemma 2. *Suppose, during an iteration i, we have a column strip with width w_i and height h_i, h_i must be a multiple of w_i. As a result, the column strip can be split into $\frac{h_i}{w_i}$ CESs of size $w_i \times w_i$.*

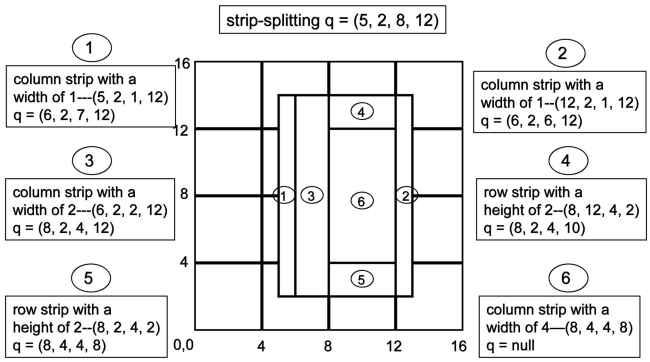


Fig. 5. An example of strip-splitting-based decomposition with CESs.

Proof. Due to successive iterations of stripping, we have $h_i \geq w_i$ and h_i can be expressed as follows:

$$h_i = \sum_{\log(w_i) \leq j < k} \theta_j 2^j + \rho L, \tag{1}$$

where $\theta_j \in \{0, 1, 2\}$ and ρ is a nonnegative integer. The second term is due to Lemma 1. From [28], there are at most two maximal strips of the same width or height at the boundaries. Hence, θ_j can be 0, 1, or 2. From the decomposition algorithm in Fig. 4, m doubles in each successive iteration until it reaches $L/2 = 2^{k-1}$; hence, $\log(w_i) \leq j < k$ in the first term. Since both terms in (1) are multiples of w_i, hence h_i is a multiple of w_i. □

Similarly, we can prove that if we have a row strip with width w_i and height h_i, then w_i must be a multiple of h_i. Hence, the row strip can be split or decomposed into $\frac{w_i}{h_i}$ CESs of size $h_i \times h_i$.

4.3 Search Algorithm

After decomposition, the query ID is inserted into the ID lists associated with decomposed CESs. Assume that $QL(l, c)$ denotes the query ID list associated with a level-l CES with a unique local ID c. These query ID lists contain indirectly precomputed 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 $k + 1$ CESs that contain or cover (x, y). Fig. 6 shows the pseudocode 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 two bits. As a result, the entire search operation is extremely efficient.

Note that, even though we partition each virtual square at level i into four equal-sized quadrants at level $i + 1$, similar to the quad-tree space partition, the bottom-up search algorithm described in Fig. 6 makes the CES-based query index unique. It achieves efficient search by taking

```
Bottom-up Search(x, y) {
  I_x = ⌊x⌋; I_y = ⌊y⌋;
  P_x = ⌊I_x/L⌋; P_y = ⌊I_y/L⌋;
  // (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 ≥ 0; l = l − 1) {
    c = 4^l p + z; // covering CES ID at level l
    if (QL(l, c) ≠ NULL) { output(QL(l, c)); }
    z = z/4; // right shifts by 2 bits
  }
}
```

Fig. 6. Pseudocode for a bottom-up search algorithm with CES-based indexing.

advantage of the containment encoding embedded in the local IDs of virtual squares at different levels.

4.4 Query (Re)Evaluation with CES-Based Indexing

Assume that $OL(q)$ denotes the object list for query 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 (re)evaluation. Because many objects might not have moved outside some CES boundaries since the last evaluation, the recomputation should be done incrementally. Containment encoding in the CESs makes it easy to identify the CESs that need not be visited during an incremental recomputation.

The pseudocode for Algorithm CES-IR is described in Fig. 7. IR stands for Incremental (re)evaluation. The object locations used in the last (re)evaluation are assumed to be available. These locations are referred to as the *old* locations in contrast to the *new* locations for the current (re)evaluation. 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 (re)evaluation, nothing needs to be done for this object. For an object whose location has been updated, we first compute the partition IDs 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 CESs that cover the new and old locations, respectively. On the other hand, if p_{new} and p_{old} are the same, some CESs in the same partition may contain both the old and new locations. Hence, no action is needed for these CESs. Due to containment encoding, these CESs that contain both the old and the new locations can be easily identified by their local IDs. 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.

As an example, Fig. 8 shows the incremental query processing using CES-based indexing under a scenario where an object moves from an old location within CES 4 to a new location within CES 7 at level 2. Both CES 4 and CES 7 belong to the same 4×4 partition at level 0. The incremental processing starts with the computation of the unit-sized CES's covering the old and new locations at level 2. In this case, the old location is inside CES 4 and the new location is inside CES 7 at level 2. Because they are not the same, the object ID is then removed from the query results of $q1$ and $q2$ associated with CES 4 at level 2 and added into the query results of $q3$ and $q4$ associated with CES 7 at level 2. Then, we find the parents of CES 4 and CES 7 at level 2, and both of them are the same CES 1 at level 1. Hence, nothing more needs to be done. The object ID will stay with the query results of $q5$ and $q6$ associated with CES 1 at level 1 and those of $q7$, $q8$, and $q9$ associated with the ancestor of CES 1.

```
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-sized CES covering L_old(o_j));
    z_new = local ID of the unit-sized 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;
      }
    }
  }
```

Fig. 7. Pseudocode for Algorithm CES-IR.

4.4.1 CES Analysis

More formally, we can analyze the total number of CESs visited during an incremental query evaluation for an object position. Assuming that both the old and the new positions are inside the same CES at level 0. In the worst case, it is $2k$. This happens when the two paths from a pair of unit-sized CESs to the root in the perfect quaternary tree do not overlap until the root node, e.g., CES 0 and CES 12 at level 2 in Fig. 8. Here, we show that the average number is also approximately $2k$, for a large k. Let A_k denote the total number of CESs visited for a $(k+1)$-level CES query index during an incremental query evaluation for an object position, accumulating from all pairs of unit-sized CESs within the same level-0 CES. The average can be computed as

$$\frac{A_k}{\binom{4^k}{2}}, \tag{2}$$

where the denominator represents the total number of distinct pairs of the 4^k unit-sized CESs at the leaf level (see Fig. 8).

We can set up a recurrence relation as follows:

$$A_k = 4A_{k-1} + \frac{3}{4}k(16^k); A_0 = 0. \tag{3}$$

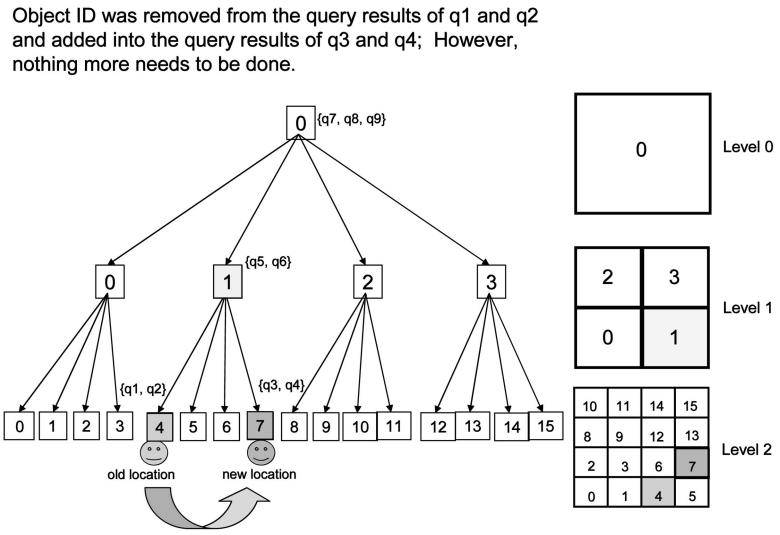


Fig. 8. An example of incremental query processing using a CES-based index.

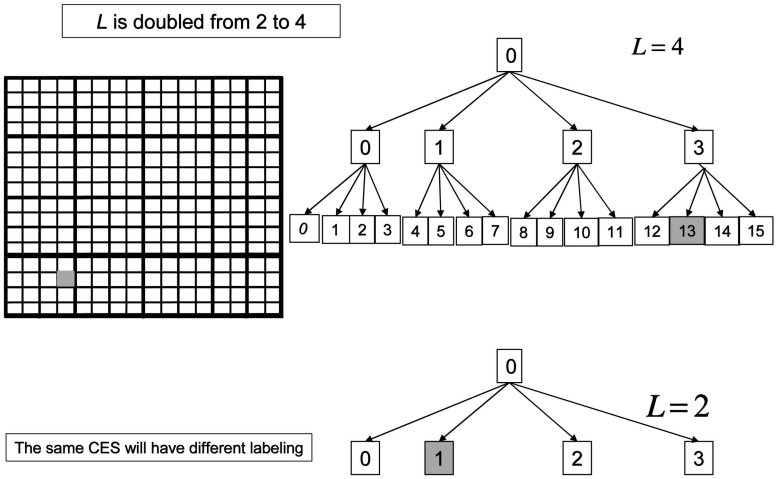


Fig. 9. An example of changes in CES labeling when L is changed.

Let us compare a k-level CES index with a $(k-1)$-level CES index. The first term ($4A_{k-1}$) is obtained because there are four $(k-1)$-level subtrees. The pairs inside each $(k-1)$-level subtree contribute the same amount since they share a subroot. The second term accounts for the case where each of the two unit-sized CESs comes from two different $(k-1)$-level subtrees. Each case contributes exactly $2k$. There are six different ways to pick two subtrees out of the four subtrees. For each pair of subtrees picked, there are $(\frac{4^k}{4}) \times (\frac{4^k}{4})$ different pairs of unit-sized CES's. Solving the recurrence relation, we have

$$A_k = k16^k + \frac{4^k}{3} - \frac{16^k}{3}. \quad (4)$$

For a large k, the average number of CESs visited is approximately $2k$.

4.5 Adapting to Changes in Query Size Distribution

So far, we have assumed that L is fixed once it is chosen. However, after a few dynamic query insertions and deletions, the distribution of query sizes might have changed. L might have to be increased or decreased accordingly. For example, if the query sizes have become larger, a larger L might be required to reduce the average number of CESs needed in query decomposition, hence reducing the index storage cost. On the other hand, if the query sizes have become smaller, then a smaller L is needed in order to reduce the index search time. To facilitate the adaptation of L, the system can maintain the statistics of the query sizes, such as the mean or median, and trigger the increase (decrease) of L when the statistics exceeds (falls behind) a threshold.

When L is changed, the ID labeling of the same CES is also changed. For example, Fig. 9 shows the different labeling of the same CES when L is increased from 2 to 4. The local labeling of the same shaded, unit-sized CES is changed from 1 to 13 when L is increased from 2 to 4. As a result, we cannot mix together in the same index queries decomposed using the new labeling and those decomposed using the old labeling.

There are generally two alternatives to handle the increase or decrease of L. The first one is to make the system quiescent for a period of time during which the entire index is completely rebuilt using the new labeling.

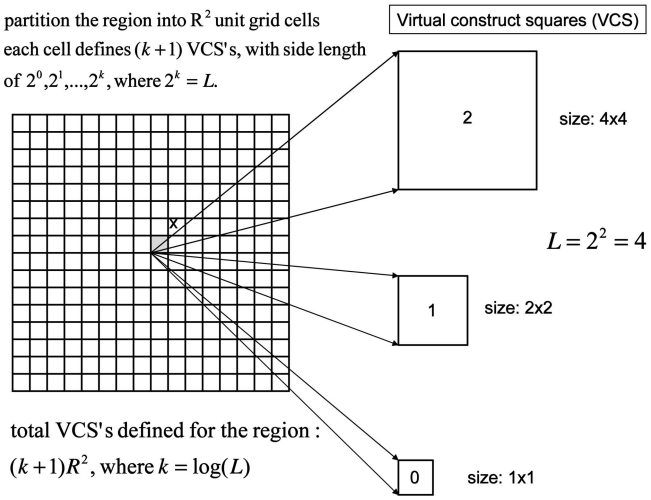


Fig. 10. An example of virtual construct squares (VCS).

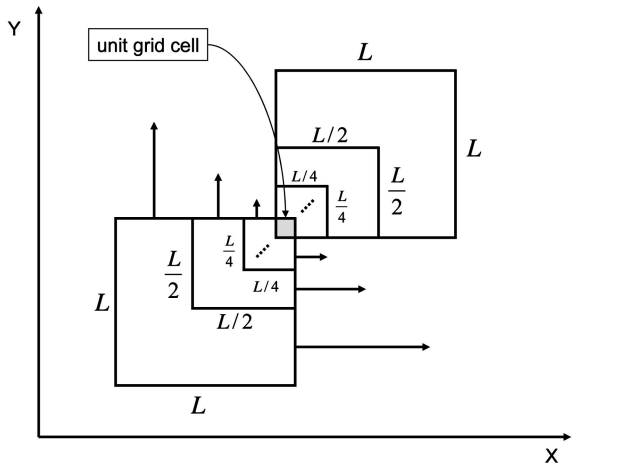


Fig. 11. Covering VCSs that contain an object location within a unit grid cell.

This approach is simple, however, it can be rather costly, especially if there are a large number of queries. The second alternative is to make the CES-based index adaptive to the change in L.

Here, we describe an adaptive solution. The basic idea is to create a second CES-based index when L is changed. Subsequent new queries will be inserted into the new index. However, subsequent searches will have to search both indexes. To avoid searching both indexes, the queries maintained in the old index can be incrementally migrated into the new index. This migration process can be done concurrently with normal search operations because the search results are always valid. In the worst-case scenario, duplicates may appear in the search results.

4.6 Dealing with a Large Number of CESs

From Property 1, when the monitoring area is large, i.e., R is large, the total number of CESs defined can be large. Hence, the total index storage cost can be large. To effectively control the index storage cost so that the entire query index can be loaded onto main memory, we can simply divide the monitoring area into multiple partitions and employ a separate computer system for each individual partition. A separate CES-based index can be easily maintained for each partition.

However, a query region may overlap with multiple partitions. In this case, it will be divided into subqueries based on the partition boundaries and will be maintained by multiple indexes. To get the total query result, we have to collect the subresults from the different indexes. When an object moves from one monitoring partition to another, the object ID is simply removed from the query results maintained in the old partition and added into the query results maintained in the new partition.

Some minor efficiency might be lost in this partitioning approach because the object may still be inside the same query but the query is maintained by two different indexes. Nevertheless, if the partitioning of the monitoring area is done properly, namely, along the places where few queries reside, then the lost efficiency can be effectively minimized.

5 Query Processing with a VCR-Based Index

For comparison, we briefly describe a prior VCR-based indexing method that uses square-only virtual constructs [32]. 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 VCSs, are defined, where $L = 2^k$ is the maximal side length of a VCS. These $k+1$ VCSs share the common bottom-left corner at (a, b) but have different sizes. For an R^2 monitoring region, the total number of VCSs defined is hence $(k+1)R^2$. In contrast, there are $4R^2/3 - R^2/(3 * 4^k)$ CESs (see Property 1). More VCSs than CESs are defined. Fig. 10 shows an example of VCSs. There are three different sizes of VCSs: 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 [32]. A VCS with the largest possible size can be used to cover the query region, 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 VCSs 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. 11. 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 VCSs 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 (re)evaluation algorithm using a VCS-based query index. The pseudocode for Algorithm VCS-IR is described in Fig. 12. For each $o_i \in O$, if the location of o_i, denoted as $L(o_i)$, has not been updated since the last (re)evaluation, nothing needs to be done for

```
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; }
    }
  }
```

Fig. 12. Pseudocode for Algorithm VCS-IR.

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.

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 VCSs 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 VCSs. 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 VCSs. 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 VCSs.

Note that in 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 described in Fig. 7.

5.1 VCS Analysis

We can also analyze the total number of VCSs calculated and examined for query reevaluation. In the worst case, the new position exceeds L away from the old position along either the x or y-dimension. In this case, all of the query ID lists from the $(4L^2 - 1)/3$ covering VCSs of the old position need to be examined for deletion; all of the query ID lists from the same amount of covering VCSs of the new position needs to be examined for insertion. Hence, a total of $2(4L^2 - 1)/3$ VCSs need to be examined.

Now, we look at the average case for small movements. The key point is to identify and count the VCSs that can cover both the new and old positions. We want to enumerate all of the possible new positions relative to the old position with a movement vector $\langle i, j \rangle$, where $-L + 1 \leq i \leq L - 1$ and $-L + 1 \leq j \leq L - 1$. Averaging individual sums of unique VCSs examined over all of the $(2L - 1)^2$ movement vectors for small movements, we can obtain the average count in an order of $O(L^2)$ with a constant factor of $\frac{32}{15}$. A detailed proof is provided in Appendix A.

6 Performance Evaluation

6.1 Simulation Studies

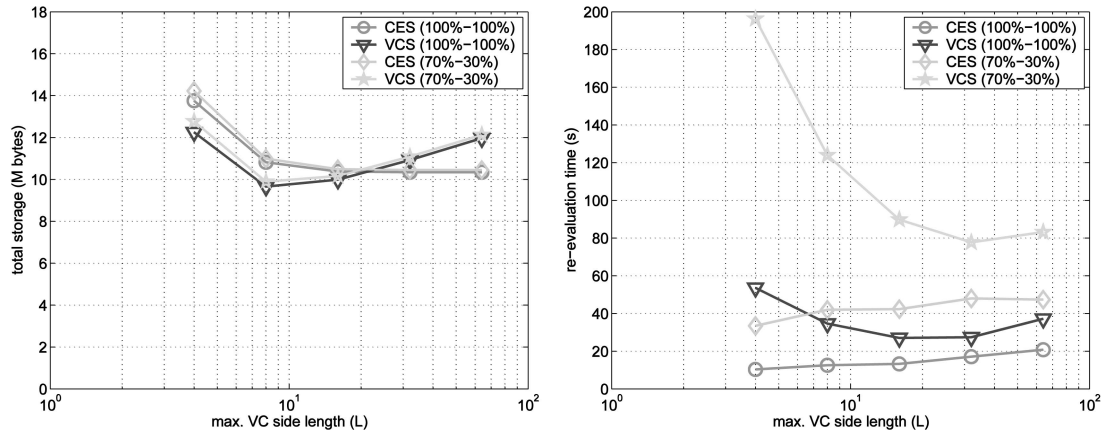
Simulations were conducted to evaluate and compare CES-based indexing with the VCS-based indexing and a grid-based approach for periodic (re)evaluation of continual range queries over moving objects. For the simulations, the monitoring region was defined by $R_x = R_y = 512$ grid units. A continual range 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 corners of range queries were distributed according to an $\alpha - \beta$ rule: α fraction of the bottom-left corners were located within β fraction of the monitoring area. For example, an 80 percent-20 percent rule means that 80 percent of the bottom-left corners of range queries are located within 20 percent of 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 distributed within the monitoring area following the same $\alpha - \beta$ rule. 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 (re)evaluation 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 (re)evaluation was performed. We measured the time it took to complete the (re)evaluation 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.

6.2 Impact of L on Index Storage Cost and Query (Re)Evaluation Time

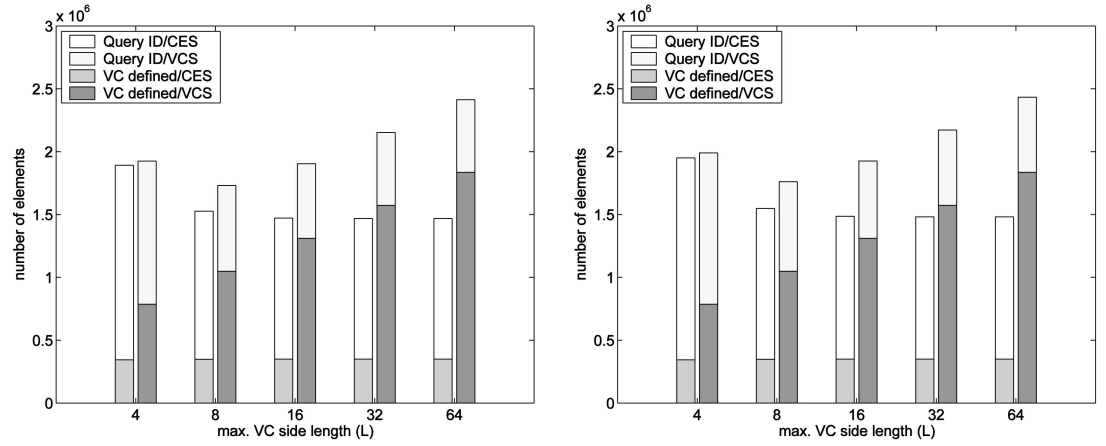
The maximal side length L of a virtual construct impacts both the index storage cost and continual query (re)evaluation time. Here, we examine the impacts of different Ls, ranging from 4 to 64, under both a CES-based and a VCS-based index. For this experiment, $W = 80$, $|Q| = 8,000$, $|O| = 50,000$, and $M = 1$. Two sets of $\alpha - \beta$ rules were used: 100 percent-100 percent and 70 percent-30 percent. They represented uniform and skewed distributions, respectively, of query regions.

From Fig. 13a, as L increases, the index storage cost steadily decreases for the CES-based index. In contrast, it decreases first and then increases for the VCS-based indexing. This is true under both uniform and skewed



(a) (b)

Fig. 13. The impact of L on (a) total index storage ($W = 80$, $|Q| = 8,000$, and $|Q| = 50,000$). (b) (Re)evaluation time ($W = 80$, $|Q| = 8,000$, and $|Q| = 50,000$).



(a) (b)

Fig. 14. Number of elements in the query ID lists and in the pointers to those ID lists, respectively, when $\alpha - \beta$ is (a) 100 percent-100 percent and (b) 70 percent-30 percent.

distributions of query regions. Nevertheless, total storage costs are comparable for both indexing schemes.

Fig. 13b shows the the average query (re)evaluation time. Here, the performance advantage of the CES-based indexing over the VCS-based indexing is clearly observed for the entire range of Ls, especially when the distribution of query regions is skewed, i.e., $\alpha - \beta = 70\% - 30\%$. This is because the number of VCs 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.

The index storage cost consists of two components: 1) the query ID lists, one for each VCS or CES defined, and 2) an array of pointers to those query ID lists. Figs. 14a and 14b show the total number of elements in each component for both indexing schemes for the uniform and skewed distributions of query regions, respectively. For both indexing schemes, the total number of elements in the query ID lists generally decreases as L increases because fewer VCs are needed to cover a range query. The CES-based indexing uses more elements in the query ID lists than the VCS-based indexing does. However, the CES-based indexing defines less CESs and hence has less elements in the array of pointers. As L increases, the total number of elements in the array of pointers increases much faster for the VCS-based indexing than the CES-based indexing. This is because the total number of VCs 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 $k = \log(L)$.

Figs. 15a and 15b show the cumulative distribution functions of the lengths of the query ID lists for both indexing schemes for the uniform and skewed distributions of query regions, respectively. Two different values of L were used, 8 and 64. For both figures, the cases of $L = 8$ and $L = 64$ for the VCS-based indexing are distinctively different but they are almost the same for CES-based indexing. Almost 60 percent and 80 percent of the query ID lists are empty for $L = 8$ and $L = 64$, respectively, for the VCS-based indexing. However, there are much less empty ID lists for the CES-based indexing, about 10 percent for the uniform distribution (Fig. 15a) and 23 percent for the skewed distribution (Fig. 15b). More importantly, the lengths of the query ID lists are generally much shorter

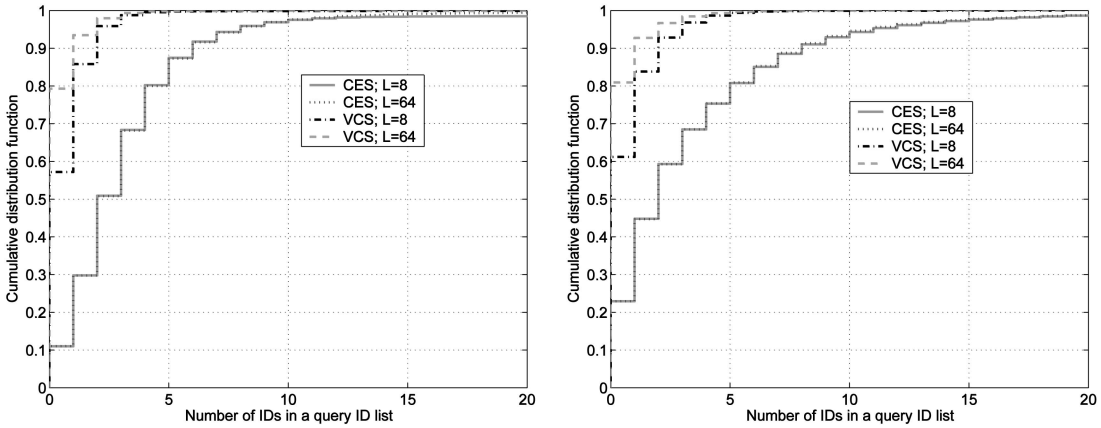


Fig. 15. Cumulative distribution functions of the length of the query ID lists when $\alpha - \beta$ is (a) 100 percent-100 percent ($W = 80$, $|Q| = 8,000$) and (b) $W = 80$, $|Q| = 8,000$ 70 percent-30 percent.

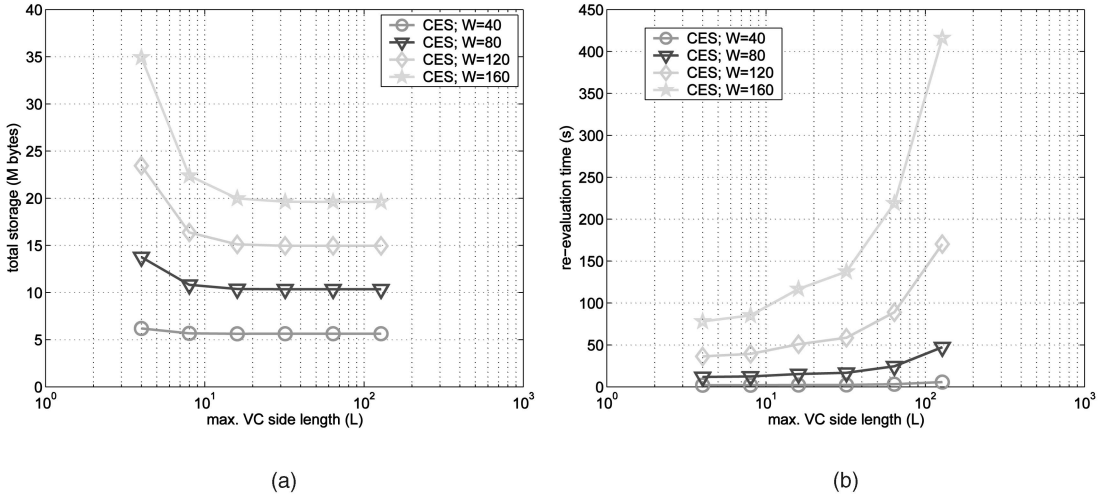


Fig. 16. The impact of query size, represented by W, on (a) total index storage ($|Q| = 8,000$, $|Q| = 50,000$) and (b) (re)evaluation time ($|Q| = 8,000$, $|Q| = 50,000$).

for the VCS-based indexing than the CES-based indexing. For example, almost all of them has less than five elements for the VCS-based indexing. In contrast, only 80 percent of them has less than five elements for the CES-based indexing. This is because there are fewer CESs than VCSs predefined.

6.3 Choosing L for a CES Index

Figs. 16a and 16b show the storage costs and total (re)evaluation times, respectively, using CES indexing with various Ls and Ws. For this experiment, $|Q| = 8,000$ and $|O| = 50,000$. Four different Ws, representing differing query sizes, were used: 40, 80, 120, and 160. For all the cases, the storage cost generally decreases as L increases from 4 to 128. When the query size is large, e.g., $W = 160$, the storage cost is substantially larger for $L = 4$ than for $L \geq 16$. When the query size is small, e.g., $W = 40$, the difference in storage cost is less significant among different Ls. For a given W, there is barely any difference in storage cost when $L \geq 16$. However, the total (re)evaluation time generally increases as L increases. From Figs. 16a and 16b, $L = 8$ or 16 is a reasonable choice in minimizing the total (re)evaluation time without incurring a substantial storage cost. For most of our experiments in this paper, we chose $L = 16$.

6.4 Comparisons of CES and VCS under Uniform Distribution

Now, we compare CES-based with VCS-based and a grid-based indexing method under various numbers of continual queries and moving objects. Using only single-sized grids, the grid-based method is a simple version of the cell-based approach [15].[2] For all the experiments, the grid size was set to L, the maximum side length of a VC in the CES-based and the VCS-based approaches. We focus on the query (re)evaluation time because it is important for determining how often query (re)evaluations can be done and how accurate query results can be.

Fig. 17a shows the impact of $|Q|$, the number of continual range queries, on the (re)evaluation time. For this experiment, $W = 50$, $L = 16$, and $|O| = 50,000$. We varied $|Q|$ from 1,000 to 16,000. Query regions were uniformly distributed. Namely, $\alpha - \beta$ was 100 percent-100 percent.

2. In general, the cell-based approach in [15] can use different-sized cells. It might partition a large cell into many small cells if the large cell happens to overlap with numerous query boundaries.

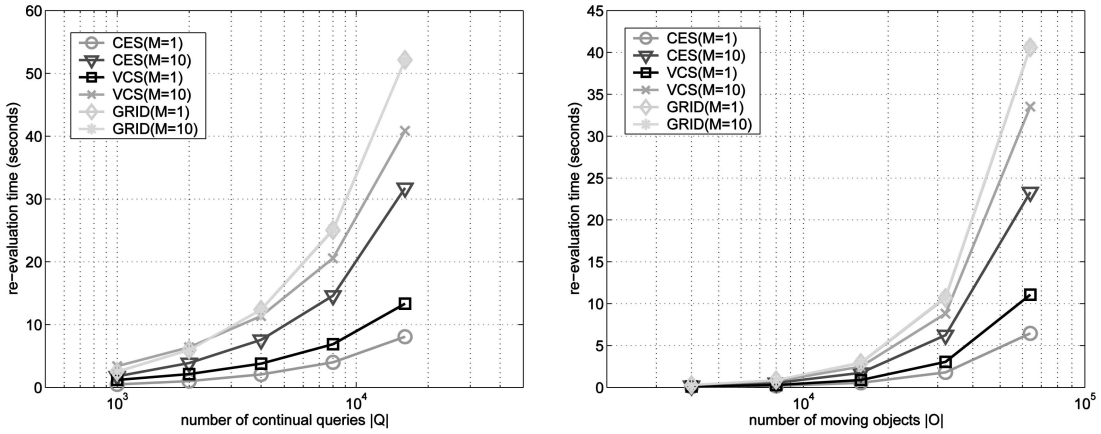


Fig. 17. The impact of (a) $|Q|$ ($W = 50$ and $|O| = 50,000$) and (b) $|O|$ ($W = 50$ and $|Q| = 50,000$), respectively, on (re)evaluation time.

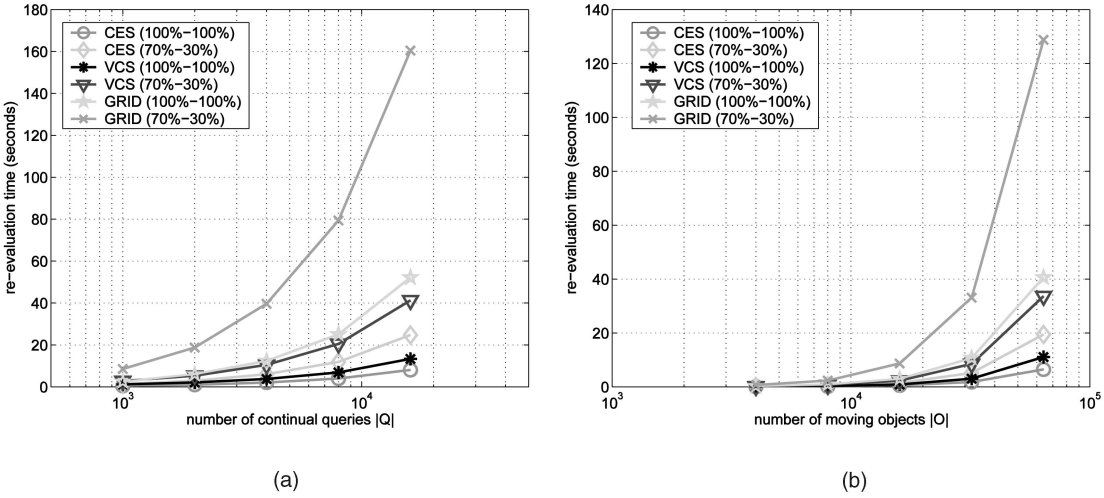


Fig. 18. The impact of (a) $|Q|$ and (b) $|O|$, respectively, on (re)evaluation time under skewed query positions.

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, which in turn outperforms grid-based indexing, for all cases. Note that, because grid-based indexing does not allow for incremental processing, there is no performance difference between the cases of $M = 1$ and $M = 10$.

Fig. 17b shows the impacts of $|O|$, the number of moving objects, on the query (re)evaluation 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, which in turn outperforms grid-based indexing, in query (re)evaluation time. Such a performance advantage becomes more prominent as the number of moving objects increases.

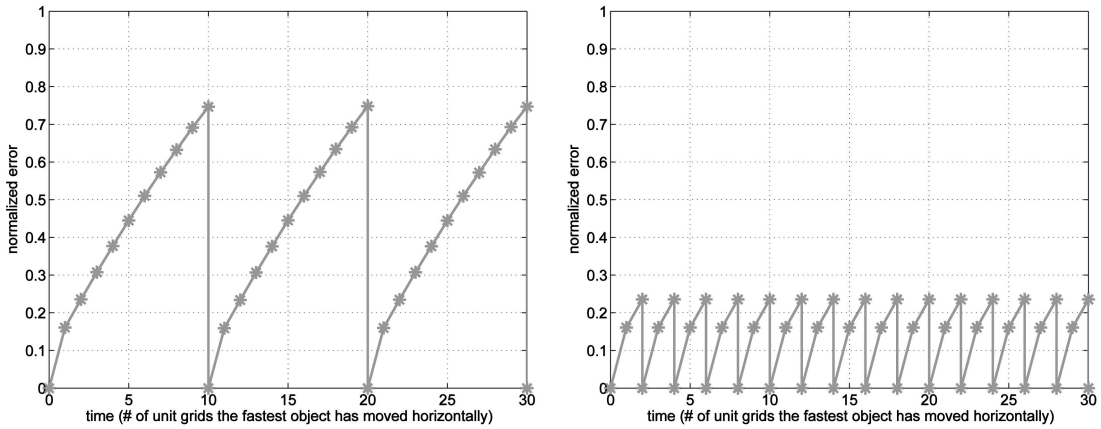
6.5 The Impact of Skew in Query Positions

Figs. 18a and 18b show the total query (re)evaluation times under the impact of skew in initial query positions. For these experiments, $M = 1$ was chosen. Two different skew cases were plotted: 100 percent-100 percent and 70 percent-30 percent. The case of 100 percent -100 percent represents the least skewed case. From these charts, it is clear that the CES-based index handles skew very well. Under all cases, the CES-based index outperforms the VCS-based index, which in turn outperforms the grid-based index. The perform advantage is more significant as the query positions become more skewed.

6.6 Inaccuracies in Query Results

So far, we have focused on the total query (re)evaluation time as the performance metric. In this section, we quantify the inherent and inevitable inaccuracies in the query results and show that these inaccuracies can be reduced by minimizing the total query (re)evaluation time.

Figs. 19a and 19b show the normalized error in query results with different time intervals between two consecutive (re)evaluations. In computing the error, we assumed that query (re)evaluation can be done instantaneously, causing the error to drop to zero the moment the queries are (re)evaluated. We also assumed that all the up-to-date object locations are available at each (re)evaluation so that we can focus on the impact of query (re)evaluation time on



(a) (b)

Fig. 19. The impact of time interval between two consecutive (re)evaluations on the inaccuracies of query results: (a) time for M to be 10 and (b) time for M to be 2.

the normalized error.[3] Due to the assumption of instantaneous (re)evaluations, the normalized error is the same for all the different approaches. We measured the normalized error as the ratio of total errors, consisting of false positives and negatives, over the total objects in the query results. False positives are those objects that are in the reported query results but should not have been, in reality, because they have since moved out of the query regions. False negatives are those objects that are not in the reported query results but should have been because they have since moved into the query regions. We measured time as the number of grids the fastest object has since moved horizontally or vertically. For this experiment, $L = 16$, $W = 50$, $R = 512$, $|O| = 50,000$, and $|Q| = 8,000$.

In general, the normalized error increases as time progresses because more objects might have moved farther away from their original positions, making the reported query results less up-to-date. However, the normalized error drops to zero once the queries are (re)evaluated. As a result, if the time interval between two consecutive (re)evaluations is larger, such as in Fig. 19a, then the reported results are less accurate, containing more errors. On the other hand, the reported results are more accurate if the time interval is shorter, such as in Fig. 19b.

From Fig. 19b, it is clear that, in order to minimize the normalized error, query (re)evaluations should be done as frequently as possible. However, because it takes a nonzero time to finish a query (re)evaluation, the best one can expect is to have a query (re)evaluation for every T seconds, where T is the total query (re)evaluation time. Hence, it is critical for a query-based indexing approach to have a minimized query (re)evaluation time so that the accuracy of the query results can be maximized.

3. In reality, some of the object locations maintained by a database server are out-of-date. This is because, as mentioned in the Introduction, object locations are typically reported periodically by the mobile objects individually back to a central server. Alternatively, position-sensing devices can be employed and the mobile object locations can be scanned periodically. In either case, certain object locations used for a query (re)evaluation are out-dated, causing potential errors in the query result even if the (re)evaluation can be done instantaneously. However, this issue is inherent in any mobile database.

6.7 The Impact of Monitoring Region Size

Finally, we study the scalability of the CES-based indexing scheme in terms of the size of the monitoring region. Figs. 20a and 20b show the impacts of R, the width of the monitoring region, on the total index storage and the total query (re)evaluation time, respectively. We varied R among 256, 512, 1,024 and 2,048. For this experiment, $W = 50$, $L = 16$, $|O| = 50,000$, and $|Q| = 8,000$ or 16,000. As R increases, more CESs are defined and, as a result, the index storage cost is higher, but only moderately. However, the total (re)evaluation time becomes significantly lower. This is because the same number of queries and objects are spread in a larger region, making most of objects covered by a small number of queries, reducing the computation cost during a query (re)evaluation.

7 Conclusion

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. CESs are virtual constructs used to cover each query region and to store indirectly precomputed query results. The use of CESs provides fast search operations. More importantly, it makes it easy to identify the moving objects that need not be evaluated during a periodic query (re)evaluation. 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 CES-based indexing substantially outperforms VCS-based indexing in query (re)evaluation time, which can improve the accuracy of the query results via more frequent (re)evaluations.

Note that, besides range queries, kNN (k nearest neighbor) queries are also important for another type of location-aware applications. Our CES-based query index is designed for efficient processing of range queries, not kNN queries. Hence, two sets of separate indexes might be needed if a system is to support both types of queries. In

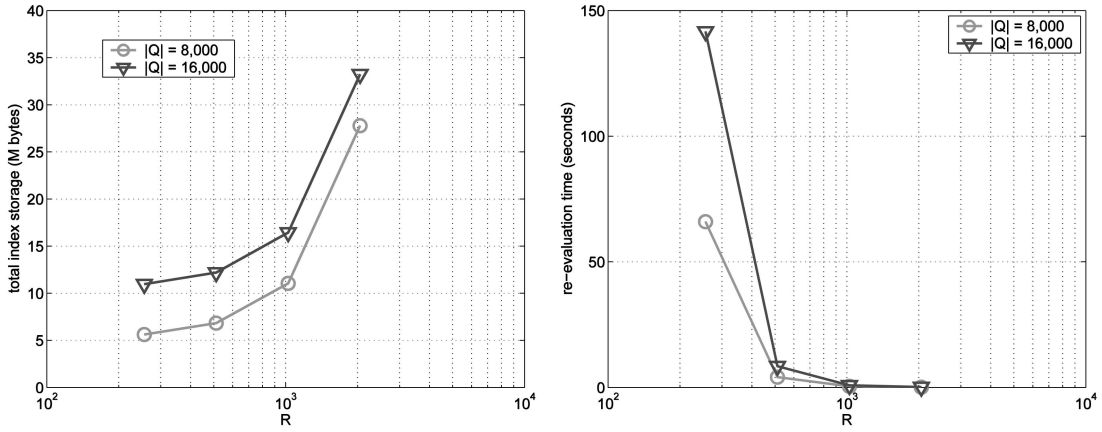


Fig. 20. The impact of R on (a) total index storage ($W = 50$, $L = 16$, and $|O| = 50,000$) and (b) total query (re)evaluation time ($W = 50$, $L = 16$, and $|O| = 50,000$).

this case, the performance advantage achieved by the CES-based index might be reduced by the increased cost of maintaining two indexes. However, the performance trade-off needs to be further studied, which is beyond the scope of this paper and can be an interesting topic for future work.

APPENDIX A
VCS ANALYSIS

Let $C_{i,j}$ denote the number of covering VCSs that cover both the old and new positions for a movement vector $\langle i, j\rangle$, as shown in Fig. 21a. Then, the number of query ID lists that need to be examined equals

$$\sum_{i=-L+1}^{L-1}\sum_{j=-L+1}^{L-1}(2(4L^2-1)/3-2C_{i,j}).$$

However, it is hard to derive a closed form for $C_{i,j}$. Consider Fig. 21b, for any given new unit-sized VCS, we have

$$C_{i,j} = (L-i)(L-j) + (L/2-i)(L/2-j) + (L/4-i)(L/4-j) + \cdots,$$

until either $(\frac{L}{2^x} - i) \le 0$ or $(\frac{L}{2^y} - j) \le 0$, where x and y are integers. For example, for $L = 8$,

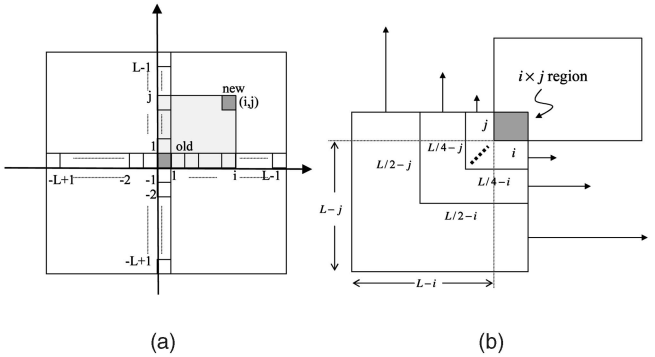


Fig. 21. An example of a movement vector and its covering VCSs.

$$C_{1,2} = (8-1)(8-2) + (4-1)(4-2) = 48.$$

Fortunately, it is possible to derive a closed-form formula for $\sum_{i=-L+1}^{L-1}\sum_{j=-L+1}^{L-1} C_{i,j}$. We outline our approach as follows: Fig. 22 shows a partial result for calculating the total sum on the first quadrant, where each cell (with coordinate $(i,\ j)$) corresponds to a different movement vector and includes the sum for the specific $C_{i,j}$. Based on Fig. 22, we have three cases: 1) $i \neq 0$ and $j \neq 0$: such a $C_{i,j}$ should be counted four times in the end, 2) either $i = 0$ or $j = 0$: such a $C_{i,j}$ should be counted twice in the end, and 3) $i = j = 0$, i.e., no movement: $C_{0,0}$ should be counted only once. Therefore, we have the total sum

$$\sum_{I=-L+1}^{L-1}\sum_{j=-L+1}^{L-1} C_{i,j} = 4\sum_{I=0}^{L-1}\sum_{j=0}^{L-1} C_{i,j} - 2\sum_{j=0}^{L-1} C_{0,j} - 2\sum_{i=0}^{L-1} C_{i,0} + C_{0,0}. \quad (5)$$

By observing from Fig. 22, we can sum the first product term in each cell for column $i = 0$, and obtain $(L-0)[(L-0)+(L-1)+\cdots+1]$; for coulmn $i = 1$, and obtain $(L-1)[(L-0)+(L-1)+\cdots+1]; \ldots$, for column $i = L-1$, and obtain

$$(L-(L-1))[(L-0)+(L-1)+\cdots+1].$$

After summing these L terms up, we have $\left[\sum_{x=1}^{L}(x)\right]^2$. Similarly, we can obtain $\left[\sum_{x=1}^{L/2}(x)\right]^2$ for summing the second product terms of all cells, $\left[\sum_{x=1}^{L/2^2}(x)\right]^2$ for the third product terms of all cells, and so on. Therefore, we have

$$\sum_{i=0}^{L-1}\sum_{j=0}^{L-1} C_{i,j} = \sum_{y=0}^{k}\left[\sum_{x=1}^{L/2^y}(x)\right]^2 = \sum_{y=0}^{k}\left[\frac{(\frac{L}{2^y}+1)(\frac{L}{2^y})}{2}\right]^2 = \frac{4}{15}L^4 + \frac{4}{7}L^3 + \frac{1}{3}L^2 - \frac{6}{35}. \quad (6)$$

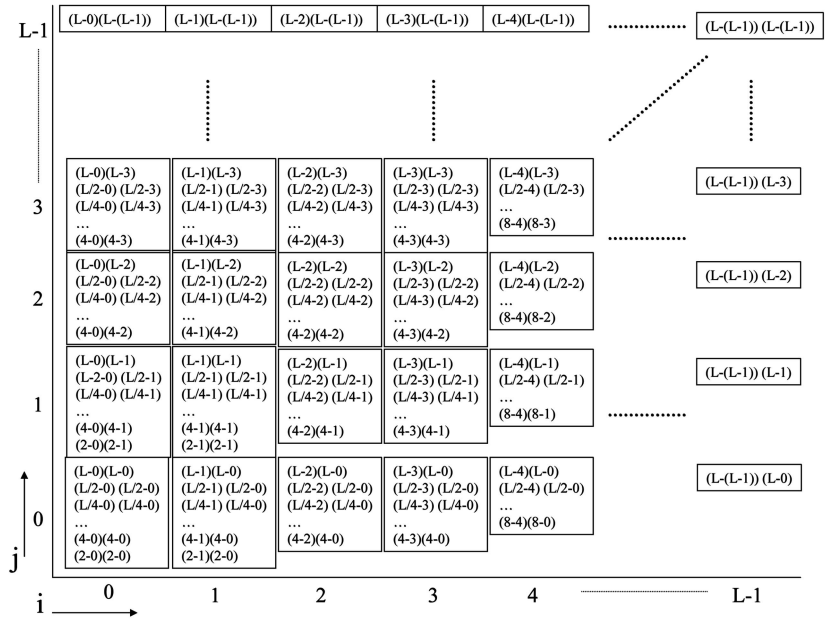


Fig. 22. An example of computing $\sum_{i=-L+1}^{L-1}\sum_{j=-L+1}^{L-1} C_{i,j}$.

It can be easily derived that $C_{0,0} = (4L^2 - 1)/3$. Moreover,

$$\sum_{j=0}^{L-1} C_{0,j}; = \sum_{i=0}^{L-1} C_{i,0} = \sum_{y=0}^{k}\left[\frac{L}{2^y}\left(\sum_{x=1}^{L/2^y} x\right)\right] = \frac{4}{7}L^3 + \frac{2}{3}L^2 - \frac{5}{21}.$$

After derivation, we have a simple form for the total sum as follows:

$$\sum_{i=-L+1}^{L-1}\sum_{j=-L+1}^{L-1} C_{i,j} = \frac{16}{15}(L^4 - 1). \tag{7}$$

Therefore, the average number of query ID lists to be examined equals

$$\begin{aligned}&\frac{\sum_{i=-L+1}^{L-1}\sum_{j=-L+1}^{L-1}(2(4L^2-1)/3 - 2C_{i,j})}{(2L-1)^2}\\ &= \frac{2(4L^2-1)}{3} - \frac{2[16/15}{(L^4-1)]}(2L-1)^2.\end{aligned} \tag{8}$$

The order is $O(L^2)$ with a constant factor of $\frac{32}{15}$.

ACKNOWLEDGMENTS

The authors would like to sincerely express their gratitude toward the anonymous reviewers for their detailed, insightful, and constructive comments, which have helped us significantly improve the quality of the paper. A preliminary version of this paper appeared in the *Proceedings of the Second International Conference on Mobile and Ubiquitous Systems: Networking and Services*.

REFERENCES

[1] P.K. Agarwal, L. Arge, and J. Erickson, "Indexing Moving Objects," *Proc. ACM Symp. Principles of Database Systems,* 2000.

[2] C.C. Aggarwal and D. Agrawal, "On Nearest Neighbor Indexing of Nonlinear Trajectories," *Proc. ACM Symp. Principles of Database Systems,* 2003.

[3] Y. Cai and K.A. Hua, "An Adaptive Query Management Technique for Real-Time Monitoring of Spatial Regions in Mobile Database Systems," *Proc. Int'l Performance, Computing, and Comm. Conf.,* 2002.

[4] Y. Cai, K.A. Hua, and G. Cao, "Processing Range-Monitoring Queries on Heterogeneous Mobile Objects," *Proc. IEEE Int'l Conf. Mobile Data Management,* 2004.

[5] L. Forlizzi, R.H. Guting, E. Nardelli, and M. Scheider, "A Data Model and Data Structures for Moving Objects," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2000.

[6] V. Gaede and O. Günther, "Multidimensional Access Methods," *ACM Computing Surveys,* vol. 30, no. 2, pp. 170-231, June 1998.

[7] B. Gedik and L. Liu, "MobiEyes: Distributed Processing of Continously Moving Queries on Moving Objects in a Mobile System," *Proc. Int'l Conf. Extending Database Technology,* 2004.

[8] B. Gedik, K.-L. Wu, P.S. Yu, and L. Liu, "Processing Moving Queries over Moving Objects Using Motion Adaptive Indexes," *IEEE Trans. Knowledge and Data Eng.,* vol. 18, pp. 651-668, May 2006.

[9] 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. Database Systems,* vol. 25, no. 1, pp. 1-42, Mar. 2000.

[10] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 1984.

[11] E. Hanson and T. Johnson, "Selection Predicate Indexing for Active Databases Using Interval Skip Lists," *Information Systems,* vol. 21, no. 3, pp. 269-298, 1996.

[12] H. Hu, J. Xu, and D.L. Lee, "A Generic Framework for Monitoring Continuous Spatial Queries Over Moving Objects," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2005.

[13] G.S. Iwerks, H. Samet, and K. Smith, "Continuous k-Nearest Neighbor Queries for Continuously Moving Objects," *Proc. Very Large Data Bases Conf.,* 2003.

[14] C.S. Jensen, D. Lin, and B.C. Ooi, "Query and Update Efficient B+-Tree Based Indexing of Moving Objects," *Proc. Very Large Data Bases Conf.,* 2004.

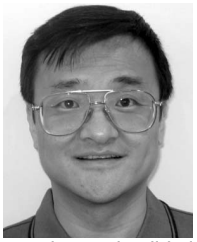
[15] D.V. Kalashnikov, S. Prabhakar, W.G. Aref, and S.E. Hambrusch, "Efficient Evaluation of Continuous Range Queries on Moving Objects," *Proc. Int'l Conf. Database and Expert Systems Applications,* 2002.

[16] G. Kollios, D. Gunopulos, and V.J. Tsotras, "On Indexing Mobile Objects," *Proc. ACM Symp. Principles of Database Systems,* 1999.

[17] M.F. Mokbel, X. Xiong, and W.G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2004.

[18] J.A. Orenstein, "Spatial Query Processing in an Object-Oriented Database System," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 1986.
[19] J.A. Orenstein and T.H. Merrett, "A Class of Data Structures for Associative Searching," *Proc. ACM Symp. Principles of Database Systems,* Apr. 1984.
[20] J.M. Patel, Y. Chen, and V.P. Chakka, "STRIPES: An Efficient Index for Predicted Trajectories," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2004.
[21] D. Pfoser, C.S. Jensen, and Y. Theodoridis, "Novel Approaches to the Indexing of Moving Object Trajectories," *Proc. Very Large Data Bases Conf.,* 2000.
[22] 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. Computers,* vol. 51, no. 10, pp. 1124-1140, Oct. 2002.
[23] H. Samet, *Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.
[24] A.P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao, "Modeling and Querying Moving Objects," *Proc. IEEE Int'l Conf. Data Eng.,* 1997.
[25] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu, "Prediction and Indexing of Moving Objects with Unknown Motion Patterns," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2004.
[26] Y. Tao, D. Papadias, and Q. Shen, "Continuous Nearest Neighbor Search," *Proc. Very Large Data Bases Conf.,* 2002.
[27] Y. Tao, D. Papadias, and Q. Shen, "The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries," *Proc. Very Large Data Bases Conf.,* 2003.
[28] Y.-H. Tsai, K.-L. Chung, and W.-Y. Chen, "A Strip-Splitting-Based Optimal Algorithm for Decomposing a Query Window Into Maximal Quadtree Blocks," *IEEE Trans. Knowledge and Data Eng.,* vol. 16, no. 4, pp. 519-523, Apr. 2004.
[29] S. Šaltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez, "Indexing the Positions of Continuously Moving Objects," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2000.
[30] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez, "Cost and Imprecision in Modeling the Position of Moving Objects," *Proc. IEEE Int'l Conf. Data Eng.,* 1998.
[31] O. Wolfson, A.P. Sistla, S. Chamberlain, and Y. Yesha, "Updating and Querying Databases that Track Mobile Units," *Distributed and Parallel Databases,* vol. 7, no. 3, pp. 257-387, 1999.
[32] K.-L. Wu, S.-K. Chen, and P.S. Yu, "Processing Continual Range Queries over Moving Objects Using VCR-Based Query Indexes," *Proc. IEEE Int'l Conf. Mobile and Ubiquitous Systems: Networking and Services,* Aug. 2004.
[33] K.-L. Wu, S.-K. Chen, and P.S. Yu, "Efficient Processing of Continual Range Queries for Location-Aware Mobile Services," *Information Systems Frontiers,* vol. 7, nos. 4-5, pp. 435-448, Dec. 2005.
[34] K.-L. Wu, S.-K. Chen, and P.S. Yu, "Shingle-Based Query Indexing for Location-Based Mobile E-Commerce," *Proc. IEEE Int'l Conf. E-Commerce,* July 2004.
[35] X. Xiong, M.F. Mokbel, and W.G. Aref, "SEA-CNN: Scalable Processing of Continuous k-Nearest Neighbor Queries in Spatial-Temporal Databases," *Proc. IEEE Int'l Conf. Data Eng.,* 2005.
[36] X. Yu, K.Q. Pu, and N. Koudas, "Monitoring k-Nearest Neighbor Queries over Moving Objects," *Proc. IEEE Int'l Conf. Data Eng.,* 2005.
[37] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D.L. Lee, "Location-Based Spatial Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2003.

Kun-Lung Wu received the BS degree in electrical engineering from the National Taiwan University, Taipei, and the MS and PhD degrees in computer science both from the University of Illinois at Urbana-Champaign. He is currently with the IBM Thomas J. Watson Research Center as a research staff member in the Software Tools and Techniques Group. His recent research interests include data streams, continual queries, mobile computing, Internet technologies and applications, database systems, and distributed computing. He has published extensively and holds many patents in these areas. Dr. Wu is a senior member of the IEEE Computer Society and a member of the ACM. He was an associate editor for the *IEEE Transactions on Knowledge and Data Engineering*, 2000-2004. He was the general chair for the Third International Workshop on E-Commerce and Web-Based Information Systems (WECWIS '01). He is the cochair for the Industrial Track for ACM CIKM 2006. He is also the program cochair for the IEEE Joint Conference of EEE 2007 and CEC 2007. He has served as an organizing and program committee member of various ACM and IEEE conferences. He has received several IBM awards, including IBM Corporate Environmental Affair Excellence Award, Research Division Award, and Invention Achievement Awards. He received a best paper award from the IEEE EEE 2004. He is an IBM Master Inventor.



Shyh-Kwei Chen received the BS degree in computer science and information engineering from the National Taiwan University, Taipei, the MS degree in computer science from the University of Minnesota, Minneapolis, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign. He is with the IBM Thomas J. Watson Research Center, currently as a research staff member. His research interests include XML, model creation and validation, multidimensional indexing, Internet technology, and e-business applications. Dr. Chen received various IBM awards, including a Research Division Award and Invention Achievement Awards. He received a best paper award from IEEE EEE 2004. He is a member of the ACM, the IEEE, and the IEEE Computer Society.

Philip S. Yu received the BS degree in electrical engineering from National Taiwan University, Taipei, the MS and PhD degrees in electrical engineering from Stanford University, and the MBA. degree from New York University. He is with the IBM Thomas J. Watson Research Center and currently manager of the Software Tools and Techniques group. His research interests include data mining, Internet applications and technologies, database systems, multimedia systems, parallel and distributed processing, and performance modeling. Dr. Yu has published more than 450 papers in refereed journals and conferences. He holds or has applied for more than 250 US patents. He is a fellow of the ACM and the IEEE. He is an associate editor of the *ACM Transactions on the Internet Technology* and the *ACM Transactions on Knowledge Discovery in Data*. He is a member of the IEEE Data Engineering steering committee and is also on the steering committee of IEEE Conference on Data Mining. He was the editor-in-chief of the *IEEE Transactions on Knowledge and Data Engineering* (2001-2004), an editor, advisory board member and also a guest coeditor of the special issue on mining of databases. He had also served as an associate editor of *Knowledge and Information Systems*. In addition to serving as a program committee member on various conferences, he will be serving as the general chair and program chair of several conferences in 2006. He was the program chair or cochair of several conferences and workshops. He has received several IBM honors including two IBM Outstanding Innovation Awards, an Outstanding Technical Achievement Award, two Research Division Awards, and the 86th plateau of Invention Achievement Awards. He received a Research Contributions Award from the IEEE International Conference on Data Mining in 2003 and also an IEEE Region 1 Award for "promoting and perpetuating numerous new electrical engineering concepts" in 1999. Dr. Yu is an IBM Master Inventor.