RC23062 (W0401-072) January 14, 2004
Computer Science

IBM Research Report

Shingle-Based Query Indexing for Location-Based Mobile E-Commerce

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.

Shingle-Based Query Indexing for Location-Based Mobile E-Commerce

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

We present a shingle-based query index (SQI) for supporting location-based services in mobile e-commerce. SQI is used to efficiently identify moving objects that are currently located inside a geographical region. A set of virtual shingles is predefined, each with a unique ID. One or more shingles are used to cover the geographical region defined by a range query, where the covering shingles may overlap with one another. SQI maintains a direct mapping from individual shingles to the range queries that contain them. The use of covering shingles has two important properties. First, it does not impose any limit on the object moving speed or direction. Second, it allows the reevaluation of continual range queries to capitalize on the incremental changes in object locations. Simulations are conducted to evaluate the effectiveness of SQI and compare it with a cell-based approach.

Keywords: Location-Based Services, E-Commerce Enabling Technologies, Continual Query, Moving Objects, Query Indexing, and Mobile Computing.

1 Introduction

Location-based services have become possible by the advances in mobile computing and location-sensing technologies, such as the global positioning systems (GPS). Knowledge of a potential customer's location can be used to deliver relevant, timely, and engaging content and information. Location-awareness can be added to many objects, such as humans, taxi cabs, ambulances and laptops, opening up new business opportunities for many commercial entities. For example, retail stores in a mall can send timely e-coupons to the PDAs or cell-phones of potential customers who are in the vicinities of their stores. A mobile service provider can provide location tracking service to the retail stores so that they know which cell phones or PDAs are currently close to their stores.

To support location-based services, the service provider must quickly evaluate a set of continual range queries, which locate the moving objects currently contained inside the geographical boundaries defined by the queries. These range queries are termed *continual* because they are repeatedly evaluated to provide up-to-date results as objects move around

continuously. For example, we can place a square or a circle with a radius of 2 miles around the location of a hotel, apartment building, or a subway exit. A taxi cab company can quickly dispatch a taxi to a customer at one of those locations if the company knows which taxi cabs happen to be near the customer location at that moment. The range query that locate the taxi cabs within 2 miles must be evaluated continually.

In this paper, we study the problem of efficient evaluation of numerous continual range queries over moving objects. To do that, an object index or a query index can be used. In this paper, we focus on maintaining an efficient query index because they change less frequently. An object position is used to search the query index in order 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 queries. Periodically, we reevaluate all range queries by using each object location to search the query index.

With query indexing, it is paramount that the time it takes to perform the periodic query reevaluation must be as brief as possible. This can be achieved in two ways. First, each search operation must be efficient. Second, the query index must also allow the query reevaluation to take advantage of incremental changes in object positions. Namely, certain object positions need not be searched. We present a new query index that has such properties.

The concept of building a query index in itself is not new. Various methods have been proposed to efficiently matching events in the contexts of predicate matching [11], pub/sub [2, 6, 23], and continual queries on the Internet [4, 14]. However, these query indexing methods are mostly based on equality predicates, not range predicates. Thus, they are not generally applicable for the evaluation of continual range queries over moving objects.

Query indexing was not used in the moving object environment until recently [12, 17]. In [17], an R-tree-based query indexing method was first proposed for continual range queries over moving objects. In order to avoid excessive location updates, a safe region for each mobile object was defined. The safe region allows an object not to report its location as long as it has not moved outside its safe region. Unfortunately, determining a safe region requires intensive computation. In [12], a cell-based query indexing scheme was proposed. It was a main-memory based approach and we shown to perform better than an R-tree-based query index [12]. Basically, 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. During query reevaluation, these lists are used to find all the queries that cover an object location.

However, using partial lists has a drawback.[1] The object locations must be compared with the range query boundaries in order to identify those queries that truly cover an object. Because of that, it cannot allow query reevaluation to take advantage of the incremental changes in object locations. Even if an object has not moved outside a cell, boundary comparisons against all the queries on the partial list are still needed.

1.1 Our contributions

We propose a novel shingle-based query indexing (SQI) method. It is a main-memory based index. A set of virtual shingles are predefined, each with a unique ID. A shingle is a tile-like object that is conventionally laid in overlapping rows to cover an area, such as the rooftop

[1] Note that the cell-based query index [12] does not need partial lists if the cell size is 1×1. However, as will be shown in Section 4.4, there will be a substantially large storage cost for the index.

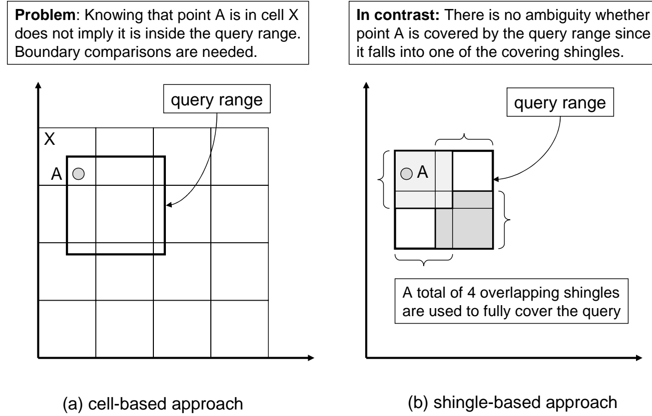

Figure 1: An example showing the problem with cell-based approach.

of a building. It can be a square or a rectangle. One or more shingles are used to strictly cover a range query. Namely, any point inside the query boundaries is covered by at least one covering shingle, and vice versa. These covering shingles may overlap with one another and are contained within the range query. SQI maintains a direct mapping from a shingle to the range queries that contain the shingle.

The use of covering shingles has two important properties. First, no constraint is imposed on the speed or direction of a moving object. An efficient algorithm is provided to identify the shingles that contain an object at any location. The containing shingles are then used to find all the range queries that contain the object. Second, more importantly, it reduces the amount of computation during query reevaluation by taking advantage of incremental changes in object locations. Computation is saved for those objects that have not moved outside a shingle.

Fig. 1 shows an example to illustrate the problem of a cell-based approach and contrasts it with the shingle-based approach presented in this paper. Under the cell-based approach (Fig. 1(a)), the query range is intersecting with 9 fixed cells, where one of them is fully covered by the query but the others are partially covered. The problem arises for objects located in a cell that intersects partially with the query boundaries. In Fig. 1(a), knowing that point A is in cell X does not imply that it is inside the query boundaries. Boundary comparisons are needed. In contrast, under our approach (Fig. 1(b)), the same range query is fully covered by 4 overlapping shingles (shingles and cells are of the same size). There is no ambiguity as to whether or not point A is covered by the query since it falls inside one of the covering shingles. Hence, incremental reevaluation approach can be easily pursued. However, because a point can be covered by multiple shingles, we need an efficient way to determine these covering shingles. We also need to find a set of shingles to strictly cover a range query. Note that both of these issues are trivial under the cell-based approach.

1.2 Related work

Although range queries can be treated as rectangles, traditional spatial indexing methods [18], such as an R-tree or any of its variants [8, 10], are not effective because they are mostly disk-based indexing methods. As shown in [12], R-tree-based query indexing is

not as effective as the cell-based approach even if it is modified for main memory access. Moreover, the performance of an R-tree quickly degenerates when the ranges of queries start to overlap one another [8, 11].

There are research papers focusing on other issues of moving object databases. For example, various indexing techniques on moving objects have been proposed [1, 5, 13, 20, 15, 16]. The trajectories, the past, current, and the anticipated future positions of the moving objects have all 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 [7, 9, 19, 22]. Uncertainty in the positions of the moving objects was dealt with by controlling the location update frequency [21, 22], 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 paper is organized as follows. Section 2 presents the shingle-based query indexing method. Section 3 presents the algorithm for the computation of query reevaluation with SQI. We show how to capitalize on the incremental changes in object locations. Section 4 shows the performance evaluation. Section 5 summarizes our paper.

2 Shingle-based query indexing

2.1 System model

The monitoring region is assumed to be partitioned into an R_xR_y virtual grid region. The grid coordinates are then used to specify range queries and moving objects in terms of positions and boundaries. Range queries are assumed to be rectangles defined along the grid lines. Namely, query boundaries are specified with integer grid coordinates. [2] However, object locations can be anywhere. We assume that continual range queries are stationary, but they can be inserted or deleted dynamically. Objects are moving continuously. Fig. 2(a) shows an example of a 13×13 monitoring region with 2 range queries and 3 moving objects. For example, query $q_1 : (1, 3, 5, 6)$ is a continual range query whose bottom-left corner is at $(1, 3)$ and its width is 5 and height is 6. An object location is specified as its x-grid and y-grid coordinates and they can be non-integers, e.g., $o_1 : (9.3, 4.15)$.

2.2 Virtual shingles

We define a set of B virtual shingles as basic building blocks for each integer grid point (a, b), where $0 \leq a < R_x$ and $0 \leq b < R_y$. These B shingles share the common bottom-left

[2] The R_xR_y virtual grid region can be used to model a physical area with different resolutions. Let g denote the physical distance for the side length of a virtual grid cell. If $g = 1/10$ mile, a 50×50 square-mile monitoring region can be represented by a 500×500 virtual grid region. On the other hand, if $g = 1/2$ mile, the same 500×500 grid region represents a 250×250 square-mile physical region. We assume g is chosen such that all range queries can be approximately and satisfactorily specified with integer grid coordinates. If query boundaries need to be specified with a higher resolution, a smaller g must be used.

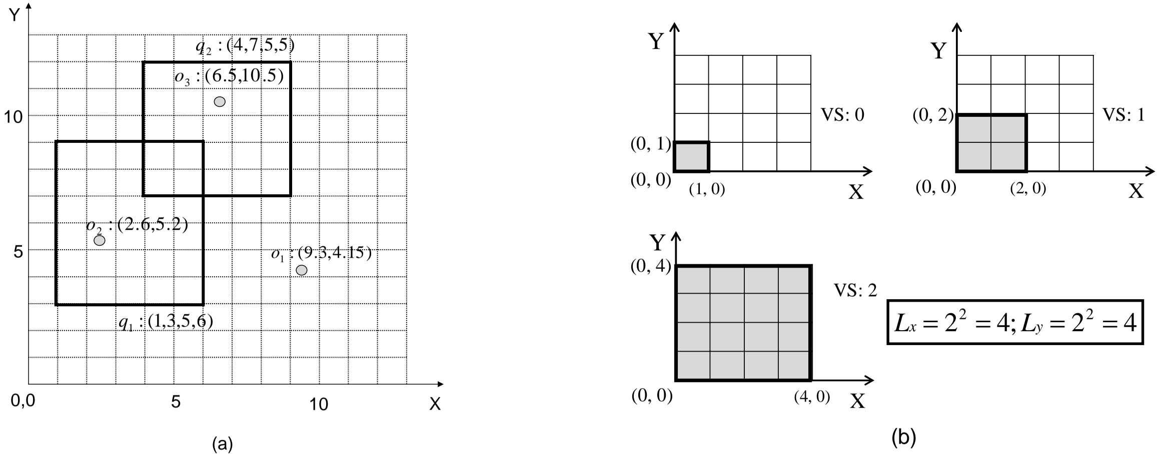

Figure 2: (a) Specifying range queries and object locations with virtual grid coordinates; (b) Assigning IDs to virtual shingles sharing the same bottom-left corner.

corner at (a, b) but have different sizes. We assume shingles are squares.[3] If the maximum side length of a virtual shingle is L, we assume $L = 2^k$ and $B = k + 1$. These B basic building blocks are defined such that a range query of any size can be strictly covered by one or more virtual shingles. Fig. 2(b) shows an example of 3 virtual shingles sharing the same bottom-left corner at $(0, 0)$, with side lengths of 1, 2 and 4, respectively.

Each virtual shingle has a unique ID and it can be easily computed as follows:

$$VS(a, b, 2^i, 2^i) = B(bR_x + a) + i, \quad (1)$$

where (a, b) is the bottom-left corner of a shingle whose side length is 2^i. The first term is derived by horizontally scanning the integer grid points from $(0, 0)$ to $(R_x - 1, 0)$, then from $(0, 1)$ to $(R_x - 1, 1), \cdots$, until $(a - 1, b)$. There are $(a + bR_x)$ such grid points (see Fig. 2(a)). For each grid point, there are B shingles defined as the basic building blocks. The second term is derived by the ID assignment shown in Fig. 2(b). Note that these shingles are virtual. A virtual shingle becomes activated when it is used to cover a continual range query. Even though there are $(k + 1)R_xR_y$ virtual shingles, there are far fewer activated shingles.

2.3 Query insertion and deletion

Let (a, b, w, h) represent a rectangle whose bottom-left corner sits at (a, b), width is w and height is h. To insert a range query q, specified as (a, b, w, h), we first find the covering shingles for q. Then, the query ID q is inserted into each of the ID lists associated with the covering shingles.

Many ways can be used to cover q with a set of virtual shingles. We present a very simple yet systematic approach. We simply find the largest i, $0 \leq i \leq k$, such that $2^i \leq \min(w, h)$, and then we use this $2^i \times 2^i$ shingle to cover q. The same sized shingle is used to strictly cover q. The covering procedure starts from the bottom-left corner, moves towards the right then moves upward. Overlapping is allowed on the rightmost column and topmost row (see Fig. 1(b)). As a result, the number of shingles used to strictly cover q is $\lceil \frac{w}{2^i} \rceil * \lceil \frac{h}{2^i} \rceil$.

[3]Note that a virtual shingle need not be a square. It can be a rectangle.

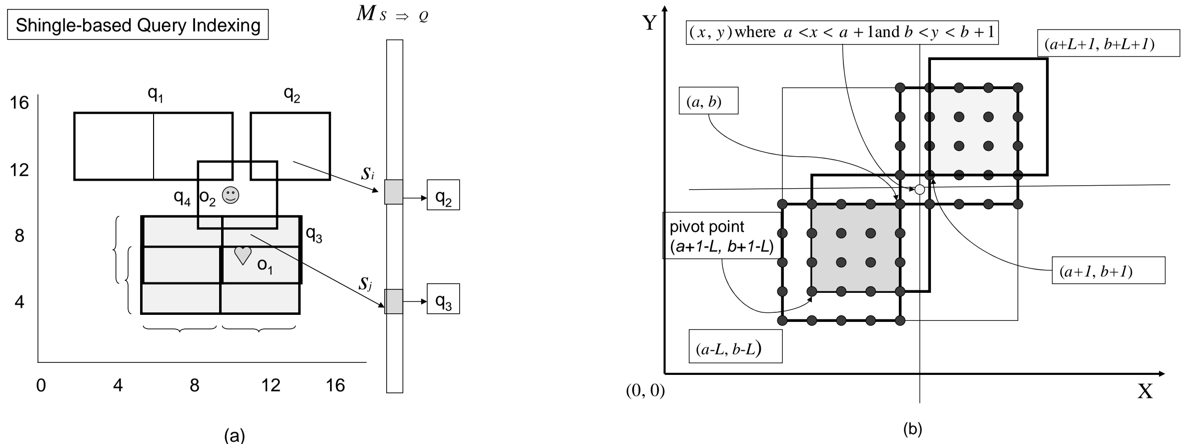

Figure 3: (a) An example of shingle-based query index; (b) Finding the covering shingles that contain an object location.

After all range queries are inserted, the shingle-based query index maintains a mapping, $M_{S \Rightarrow Q}$, from S to Q, where S is the set of activated shingles and Q is the set of all queries. Fig. 3(a) shows an example of SQI. A total of 8 shingles are activated in Fig. 3(a), 2 for q_1, 1 for q_2, 4 for q_3 and 1 for q_4. For each activated shingle $s \in S$, there is a query ID list, denoted as $QL(s)$. Note that s is contained within all the queries in $QL(s)$. The data structure for SQI includes the query lists and an array of pointers to the associated query lists.

To delete a query, we first find the covering shingles for the query, similar to query insertion. Then, the query ID is removed from each $QL(s)$, where s is a covering shingle for the query.

2.4 Search queries that contain a given object

The search algorithm with SQI is based on the following observation. Let $o \odot s$ denote that shingle s contains object o; $s \odot q$ denote that query q contains shingle s. If $o \odot s$ and $s \odot q$, then $o \odot q$. Hence, to find all the range queries containing an object, we find all the shingles containing that object.

Assume that $CQ(o)$ is the set of queries that contain an object o; $CS(o)$ is the set of covering shingles that contain an object o. $CQ(o)$ can be computed from $CS(o)$ and the $QL(\cdot)$'s maintained by SQI as follows:

$$CQ(o) = \{q | q \in QL(s) \wedge s \in CS(o)\}. \tag{2}$$

The key step in finding $CQ(o)$ is finding $CS(o)$ since $QL(s)$'s are readily available. Because the way virtual shingles are defined, $CS(o)$ can be systematically and efficiently enumerated.

$CS(\cdot)$'s for all possible object locations share two common properties: *constant size* and *identical gap pattern*. These two properties make the enumeration of $CS(\cdot)$ efficient. First, the number of virtual shingles containing a point is the same for all the object positions. [4]

[4] Strictly speaking, this is only true for a location that is not in the boundary regions. The boundary regions are defined by $0 \leq x < L$ or $R_x - L \leq x < R_x$ or $0 \leq y < L$ or $R_y - L \leq y < R_y$. However, we

Namely, $|CS(o_i)| = |CS(o_j)|$ even if $o_i \neq o_j$. Second, if we sort, in an increasing order, the IDs of shingles in each $CS(\cdot)$, then $s^{o_i}_{m+1} - s^{o_i}_m = s^{o_j}_{m+1} - s^{o_j}_m$ for $1 \leq m < |CS(\cdot)|$ and any two objects o_i and o_j. Here, $s^{o_i}_m$ is the m-th virtual shingle in the sorted $CS(o_i)$. In other words, the gap between any two virtual covering shingles of matching positions is identical for any two locations.

To verify these two properties, let us look at an example. Fig. 3(b) shows $CS(o)$ for an object o whose location is (x, y), where $a < x < a + 1$, $b < y < b + 1$, and a and b are integers. The bottom-left corners of these covering shingles must reside in the south-west shaded area of (x, y) and the upper-right corners must reside in the north-east shaded area of (x, y). It can be easily verified that if a shingle whose bottom-left and upper-right corners are positioned in the respective shared areas, it will indeed contain (x, y). The two properties can be proved by first grouping all the drawings in Fig. 3(b) as a unit and then moving it around. When the center is moved from (a, b) to another point (c, d), the relative positions of all the covering shingles stay the same.

With these two properties, we can design an efficient algorithm for computing $CS(o)$ at location (x, y). We first define a *pivot point* as PV whose location is $(\lfloor x \rfloor + 1 - L, \lfloor y \rfloor + 1 - L)$ and a *pivot shingle* as PS which is defined as $(\lfloor x \rfloor + 1 - L, \lfloor y \rfloor + 1 - L, 2^0, 2^0)$. Namely, the bottom-left corner of PS is at the pivot point PV and PS is a unit square. Then we use a pre-computed *difference array* D, which stores the differences on the ID's between two neighboring shingles in a sorted $CS(\cdot)$, and the pivot shingle PS to enumerate $CS(o)$. $CS(o)$ can be efficiently computed at runtime by simple additions of the pivot shingle ID to each element stored in D.

Theorem 1 $|CS(o)| = \frac{4L^2-1}{3}$, $\forall o(x, y)$, *where* $L \leq x < (R_x - L)$ *and* $L \leq y < (R_y - L)$.

Proof: From Fig. 3(b), we know that there is 1 shingle with the size of 1×1 that covers (x, y); 4 shingles with the size of 2×2 that cover (x, y); 16 shingles with the size of $2^2 \times 2^2$ that cover (x, y). Hence, $|CS(o)| = \sum_{i=0}^{i=k} (2^i)^2$, which is $\frac{4L^2-1}{3}$ after a few manipulations. □

3 Evaluation of continual range queries using SQI

Here, we present an incremental reevaluation algorithm based on SQI. Query results are maintained in an array of object lists, one for each query. Assume that $OL(q)$ denotes the object list for q. It contains the IDs of all objects that are inside the boundaries of q. We periodically re-compute all $OL(\cdot)$'s taking into account the changes in object locations since the last reevaluation.

If the period between two consecutive reevaluations is short, many objects may not have moved outside the shingle boundaries. As a result, many of the computations can be saved. The use of covering shingles in SQI provides a convenient way to capitalize on the incremental changes in object movements.

The pseudo code for Algorithm SQI_IR is described in Fig. 4. IR stands for Incremental Reevaluation. We assume that the object locations used in the last reevaluation are available. These locations are referred to as the *old* locations,[5] in contrast to the *new* locations for the

can also efficiently compute $CS()$'s for locations in these regions. For simplicity of presentation, we focus on locations that are not inside the boundary regions.

[5]A double buffering approach can be used to maintain both the old and new locations.

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

Figure 4: Pseudo code for Algorithm SQI_IR.

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 shingle sets: $CS_{new}(o_i)$ with the new location data and $CS_{old}(o_i)$ with the old location data.

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

4 Performance evaluation

4.1 A cell-based query indexing approach

Here, for comparison purpose, we describe the implementation of a cell-based query indexing approach, similar to the one described in [12]. The cell size is an integer multiple of the virtual grid size in Fig. 2(a). Each cell is associated with two lists: one full and one partial. The partial list maintains all the IDs of the queries that partially intersect with the cell. In order to check if $o \odot q$, one must perform comparisons using the boundaries of q and the location data of o.

The pseudo code for continual query reevaluation using CQI is shown in Fig. 5. Due to the

```
Algorithm CQI_CR
    for (i = 0; q_i ∈ Q; i++) { cleanup(OL(q_i));
}
    for (i = 0; o_i ∈ O; i++) {
        q = QL_P(C(o_i));
        while (q ≠ NULL) {
            if (o_i ⊙ q) { insert(o_i, OL(q)); }
            q = q → next;
        }
        q = QL_F(C(o_i));
        while (q ≠ NULL) {
            insert(o_i, OL(q)); q = q → next;
        }
    }
```

Figure 5: Pseudo code for CQI_CR.

use of partial lists, it is difficult, if not impossible, to capitalize on the incremental changes in object locations with CQI. Hence, it must perform a complete reevaluation. In CQI_CR, it first cleans up all the $OL(\cdot)$'s. Then, for each object $o_i \in O$, it performs comparisons to test if $o_i \odot q$ for every $q \in QL_P(C(o_i))$, where $C(o_i)$ is the cell ID in which o_i is located and $QL_P(\cdot)$ is the associated partial list. If $o_i \odot q$, then o_i is inserted into $OL(q)$. For every query q stored in the full list, $QL_F(C(o_i))$, o_i is simply inserted into $OL(q)$.

4.2 Simulation studies

Simulations were conducted to evaluate and compare SQI with CQI for periodic reevaluations of continual range queries over moving objects. Since it has been shown in [12] that a cell-based approach outperforms query indexing schemes based on various R-trees, we focus on comparing our schemes with the cell-based approach. For the simulations, the monitoring region was defined by $R_x = R_y = 500$. 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. Its bottom-left corner was chosen uniformly within the monitoring area. The maximum side length of a shingle was L, $L = 2^k$ and k was 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 moving objects were uniformly distributed within the monitoring area. Then, 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

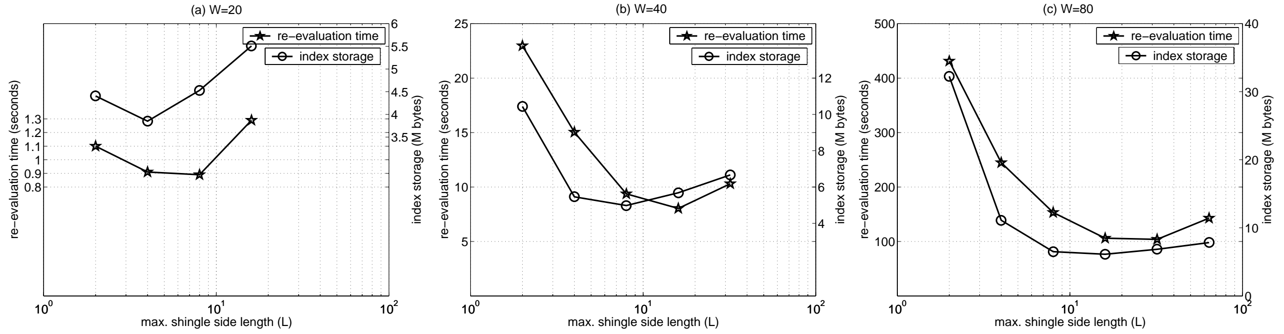

Figure 6: The impacts of maximal shingle size L on reevaluation times and index storage costs for SQI with overlapping shingles when (a) $W = 20$; (b) $W = 40$; (c) $W = 80$.

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 RS6000 44p model 170 machine (CPU 333 MHz; memory size 768 Mbytes) running AIX 4.3.3.

4.3 The impact of shingle size

The maximal shingle size, $L = 2^k$, has important impacts on index storage cost and reevaluation time. In this section, we use both simulations and analyses to study the optimal L for minimizing the index storage cost and query reevaluation time.

The storage cost of SQI can be estimated as follows:

$$C_{storage} \simeq 4BR_xR_y + 8|Q|\overline{Cov(\cdot)}, \tag{3}$$

where $\overline{Cov(\cdot)}$ represents the average number of covering shingles per query. The first term is the storage cost for an array of pointers to query lists and the second term for the query lists. Here, we assume each predicate ID requires 4 bytes and each pointer also requires 4 bytes. Each element in a query list contains a query ID and a pointer to the next element. Hence, we have the constant 8 in the second term. $\overline{Cov(\cdot)}$ depends on W and L. With a smaller L, more shingles are needed to strictly cover a query, increasing $\overline{Cov(\cdot)}$. However, B, which is $\log(L)$, becomes smaller.

Because $B = k + 1 = \log(L) + 1$ and $\overline{Cov(\cdot)} \simeq \lceil \frac{W}{2L} \rceil \times \lceil \frac{W}{2L} \rceil$ (note that the average query size is $\frac{W}{2} \times \frac{W}{2}$), Eq. (3) can be approximately expressed as follows:

$$C_{storage} \simeq 4(\log(L) + 1)R_xR_y + 8|Q|(\frac{W}{2L})^2. \tag{4}$$

We can calculate the optimal L by taking the first derivative of $C_{storage}$ and setting it to zero. As a result, to achieve the minimal index storage cost, L can be chosen as follows:

$$L^{opt}_{storage} \simeq W\sqrt{\frac{Q}{R_xR_y}}. \tag{5}$$

Table 1: The optimal L for minimizing reevaluation time.

$\|Q\|$	$\|O\|$	optimal L ($W = 20$)	optimal L ($W = 40$)	optimal L ($W = 80$)
10,000	10,000	4	8	16
20,000	10,000	8	16	32
40,000	10,000	8	16	32
10,000	20,000	4	8 or 16	16 or 32
20,000	20,000	8	16	32
40,000	20,000	8	16	32
10,000	40,000	8	16	32
20,000	40,000	8	16	32

Query reevaluation time can be approximately expressed as follows:

$$C_{time} \simeq |O| \times |CS(\cdot)| \times \overline{|QL(\cdot)|} \times f \times \frac{\overline{|OL(\cdot)|}}{2}, \quad (6)$$

where f is the fraction of covering shingles that require insertions or deletions of object instances, $\overline{|QL(\cdot)|}$ and $\overline{|OL(\cdot)|}$ represent the average sizes of a $QL(\cdot)$ and $OL(\cdot)$, respectively. We assume that, on average, the insertion and deletion of an object instance need to traverse half of an $OL(\cdot)$. Among these terms, $|CS(\cdot)|$ depends on L; f depends on L, M and W; $|QL(\cdot)|$ depends on $|Q|$, L and W; $|OL(\cdot)|$ depends on $|O|$ and W. When L is small, $|QL(\cdot)|$ becomes large because more shingles are needed to cover a query. When L is large, $|CS(\cdot)|$ becomes large as well.

From Theorem 1, $|CS(\cdot)| = \frac{4L^2-1}{3}$. Furthermore, $\overline{QL(\cdot)} \simeq \frac{|Q|\overline{Cov(\cdot)}}{(k+1)R_xR_y}$ because each query on average is covered by $\overline{Cov(\cdot)}$ shingles and there are a total of $(k+1)R_xR_y$ virtual shingles. $\overline{OL(\cdot)} \simeq \frac{|O|W^2}{4R_xR_y}$ because the average query size is $\frac{W}{2} \times \frac{W}{2}$ and the monitoring area is R_xR_y. However, it is nontrivial to express f in terms of L, M and W. As a result, we use simulations to find the optimal L that minimizes the reevaluation time in Eq. (6).

Figs. 6(a), 6(b) and 6(c) show the reevaluation times (in seconds) and index storage costs (in M bytes) of SQI with overlapped covering shingles when $W = 20$, 40, and 80, respectively. The reevaluation time is on the left y-axis of each figure while the index storage cost on the right y-axis. For these experiments, $|Q| = 10,000$, $|O| = 50,000$ and $M = 1$. As predicated by Eq. (5), the optimal L's for achieving the minimal index storage costs when $W = 20$, 40 and 80 are 4, 8, and 16, respectively. The optimal L's for achieving the minimal reevaluation times when $W = 20$, 40 and 80 are 8, 16 and 32. Note that the optimal L for minimal storage cost is different from that for minimal reevaluation time.

Many experiments with various $|Q|$'s, $|O|$'s and W's were conducted to find the optimal L for achieving minimal reevaluation time. Table 1 shows the optimal L for achieving minimal reevaluation time under a given $|Q|$, $|O|$ and W. In general, the optimal shingle size for minimizing reevaluation time can be approximately expressed as follows:

$$L_{time}^{opt} = 2^i, \text{where } 2^i \leq \frac{W}{2} < 2^{i+1}. \quad (7)$$

For example, the optimal L's are 8, 16 and 32 for $W = 20, 40$ and 80, respectively, for most of the cases in Table 1. However, there are exceptions when both $|Q|$ and $|O|$ are relatively small. For instance, when $|Q| = |O| = 10,000$, the optimal L's are 4, 8 and 16 for $W = 20, 40$ and 80, respectively. In this case, $2^i \leq \frac{W}{4} < 2^{i+1}$.

4.4 Comparisons of SQI and CQI

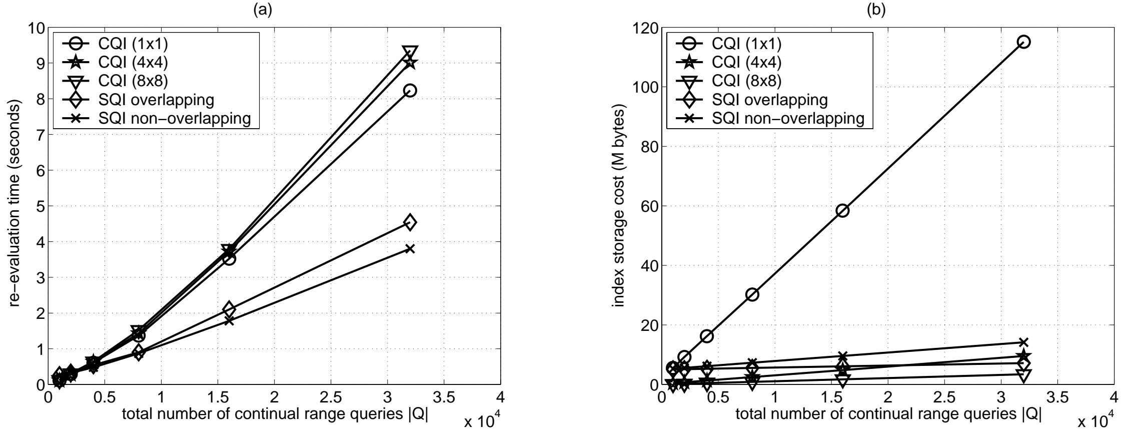

Figure 7: The impacts of $|Q|$ on (a) reevaluation time and (b) index storage cost.

Now we compare SQI with CQI. We varied the total number of range queries $|Q|$. For the CQI scheme, three different cell sizes were used: 1×1, 4×4 and 8×8. Figs. 7(a) and 7(b) show the reevaluation times and index storage costs, respectively, for the 3 CQI schemes and the SQI schemes with and without overlapped covering shingles.[6] For these experiments, $W = 40$, $|O| = 20,000$, and $M = 1$. For the CQI scheme, a smaller cell size can achieve a better reevaluation time, but the storage cost increases (see the 1×1 cases in both Figs. 7(a) and 7(b)).[7] Both the SQI with or without overlapped shingles performs better in reevaluation times than the CQI schemes, especially when $|Q|$ is large. This is because SQI allows the query reevaluation to capitalize on incremental changes in object locations. The performance advantage of SQI in reevaluation time is achieved at a modest increase in storage cost. Note that the CQI scheme with a cell size of 1×1 incurs significantly larger storage cost compared with the SQI schemes, especially for a large $|Q|$ (see Fig. 7(right)). The storage cost of CQI quickly decreases as the cell size increases, but the reevaluation time also increases.

5 Summary

In this paper, we have presented a novel shingle-based query index (SQI) for efficient evaluation of continual range queries against moving objects. It is fundamental to support many

[6]Note that, in the case of SQI without overlapping shingles, we use different sized shingles to fully cover a query range, similar to using various sized tiles to cover a floor.

[7]Note that when the cell size is 1×1, CQI does not have partial lists. CQI is then equivalent to SQI with $L = 1$. However, there will be a substantially large storage cost for both CQI and SQI. As a result, the optimal cell size or shingle size is larger than 1×1.

location-based services in mobile E-commerce environments. SQI is a main-memory based index. The objective is to quickly provide answers to continual range queries that locate the moving objects contained inside the query boundaries.

SQI is based on the concept of covering a range query with one or more overlapping shingles. A shingle is a tile-like object that is conventionally laid in overlapping rows to cover an area, such as the rooftop of a building. A set of virtual shingles is predefined, each with a unique ID. One or more of these virtual shingles are used to strictly cover each query. SQI maintains a direct mapping between a shingle and the range queries that contain the shingle. The use of covering shingles in SQI has two important properties. First, it imposes no restriction on the speed and direction of a moving object. Second, it allows the query reevaluation to capitalize on incremental changes on object positions.

Simulations and analyses have been conducted to find the optimal shingle size for SQI and to evaluate and compare SQI with a cell-based approach. The results show that (1) the optimal shingle size for minimizing index storage cost is different from that for minimizing query reevaluation time; (2) SQI performs better than the cell-based scheme in query reevaluation time.

References

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

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proc. of Symp. on Principles of Distributed Computing*, 1999.

[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 Conference*, 2002.

[4] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. of ACM SIGMOD*, pages 379–390, 2000.

[5] H. D. Chon, D. Agrawal, and A. E. Abbadi. Query processing for moving objects with space-time grid storage model. In *Proc. of 3rd Int. Conf. on Mobile Data Management*, 2002.

[6] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proc. of the ACM SIGMOD*, 2001.

[7] 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.

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

[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 TODS*, 25(1):1–42, Mar. 2000.

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

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

[12] D. V. Kalashnikov, S. Prabhakar, W. G. Aref, and S. E. Hambrusch. Efficient evaluation of continuous range queries on moving objects. In *Proc. of 13th Int. Conf. on Database and Expert Systems Applications*, 2002.

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

[14] L. Liu, C. Pu, and W. Tang. Continual queries for Internet scale event-driven information delivery. *IEEE TKDE*, 11(4):610–628, July/Aug. 1999.

[15] H. K. Park, J. H. Son, and M. H. Kim. An efficient spatiotemporal indexing method for moving objects in mobile communication environments. In *Proc. of Int. Conf. on Mobile Data Management*, 2003.

[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 constrainted 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 ICDE*, 1997.

[20] 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.

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

[22] 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.

[23] K.-L. Wu and P. S. Yu. Efficient query monitoring using adaptive multiple key hashing. In *Proc. of ACM CIKM*, pages 477–484, 2002.