Scalable Spatio-temporal Continuous Query Processing for Location-aware Services

Xiaopeng Xiong Mohamed F. Mokbel Walid G. Aref Susanne E. Hambrusch Sunil Prabhakar
Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398
{xxiong,mokbel,aref,seh,sunil}@cs.purdue.edu

Abstract

The emergence of location-aware services calls for new real-time spatio-temporal query processing algorithms that deal with large numbers of moving objects and large numbers of continuous spatio-temporal queries. In this paper, we use shared execution *as a mechanism to support scalability in location-aware servers. The main idea is to maintain a query table that stores information about continuous spatio-temporal queries. Then, answering spatio-temporal queries is abstracted as a spatial join among the moving objects and queries. Three query join polices are proposed aiming to minimize the cost of the join operation under the shared execution paradigm, namely the* Clock-triggered Join Policy*, the* Incremental Join Policy*, and the* Hot Join Policy*. We introduce the concept of a* No-Action Region *that is used in conjunction with the hot join policy. We propose algorithms that calculate the No-Action region for objects and queries. Experimental performance demonstrates that the No-Action region is more efficient than other approaches when used along with the hot join policy. Experiments also demonstrate that the hot join policy outperforms the clock-triggered join policy and the incremental join policy in terms of both I/O and CPU costs.*

1 Introduction

Combining the functionality of personal locator technologies, global positioning systems (GPSs), wireless and cellular telephone technologies, and information technologies enables new environments where virtually all objects of interest can determine their locations. These technologies are the foundation for pervasive location-aware environments and services. Such services have the potential to improve the quality of life by adding location-awareness to virtually all objects of interest such as humans, cars, laptops, eyeglasses, canes, desktops, pets, wild animals, bicycles, and buildings.

Location-aware environments have the following distinguished characteristics: (1) A large number of objects and a large number of queries interested on these objects, (2) Most of the queries issued to the location-aware server are continuous spatio-temporal queries. Unlike snapshot queries that are evaluated only once, continuous queries require continuous evaluation as the query result becomes invalid with the change of information, and (3) Queries as well as objects have the ability to be stationary or moving. Thus a location-aware server should have the ability to support a wide variety of continuous spatio-temporal queries, e.g., stationary queries on moving objects, moving queries on stationary objects, and moving queries on moving objects.

By employing the *shared execution* paradigm [6, 7, 16] in location-aware servers, the problem of concurrently evaluating a set of continuous spatio-temporal queries is abstracted into a spatial join problem. The main idea is to group the set of continuous spatio-temporal queries in one table that is joined with another table that contains the locations of moving objects. So, instead of having one query evaluation plan per query, we have only one shared execution query evaluation plan for all the queries. The shared execution paradigm has been widely utilized for continuous web queries, and continuous streaming queries. For continuous spatio-temporal queries, the shared execution paradigm is utilized to support only the case of stationary queries over moving objects. The underlying join operation in the shared execution paradigm is mainly dependent on the underlying application.

In this paper, we focus on realizing the join operation of the shared execution paradigm to support all mutability variations of continuous spatio-temporal queries. We propose three join policies that aim to minimize the number of computations. In addition, we propose the notion of a *No-Action Region*, which is

the region that an object or a query can move inside without affecting the latest reported result of any continuous query. *No-Action Regions* are utilized to minimize the required join operations in a shared execution query plan. Although our focus is on continuous spatio-temporal range queries, the ideas and concepts are applicable to other kinds of continuous spatio-temporal queries. The contributions of this paper are summarized as follows:

1. We utilize the shared execution of spatio-temporal queries to achieve scalability in terms of large query and object numbers, and abstract the problem of evaluating multiple continuous spatio-temporal range queries to the spatial join problem.

2. We propose three join policies; the *Clock-triggered Join Policy*, the *Incremental Join Policy*, and the *Hot Join Policy* that join a set of spatio-temporal objects with a set of spatio-temporal queries. The three join policies fit with the *shared* execution paradigm.

3. We propose the concept of *No-Action Region* that generalizes the concepts of *validity region* [21] and *safe region* [18]. The *No-Action region* concept is used in conjunction with the *Hot Join Policy* to minimize the number of join operations.

4. We provide a comprehensive set of experiments that show the performance of the proposed join policies. In addition, we show that utilizing *No-Action regions* is more efficient than utilizing validity regions or safe regions.

The rest of the paper is organized as follows: Section 2 gives some preliminaries that are used throughout the paper. In Section 3, we introduce the shared execution paradigm that is used as a means for achieving scalability in location-aware servers. Section 4 proposes three join policies that fit with the shared execution paradigm. Section 5 provides an extensive set of experiments to study the performance of the proposed join policies and *No-Action* regions. In Section 6, we highlight related work for continuous spatio-temporal query processing. Finally, Section 7 concludes the paper.

2 Preliminaries

2.1 Location-aware Environment Model

In this section, we briefly introduce the *Pervasive Location Aware Computing Environments* Project (PLACE); a project being developed at Purdue University [1, 16]. Location-detection devices (e.g., GPS devices) provide the objects with their geographic locations. Objects connect directly to regional servers that handle the incoming data and process time-critical spatio-temporal queries. Regional servers communicate with each other, as well as with higher level repository servers.

The PLACE server keeps track of stationary objects (e.g., gas stations) as well as moving objects (e.g., cars). Moving objects update their location in the PLACE server every T seconds. An object is considered stationary at the time interval $[T_i, T_j]$ if the server does not receive any update from o during this interval. Moving objects have the ability to issue stationary or moving spatio-temporal queries. To cope with the continuous nature of spatio-temporal queries, the PLACE server updates the answer of a continuous spatio-temporal query every T seconds.

2.2 Spatio-temporal Query Types

Unlike traditional and spatial queries, in spatio-temporal queries, both objects and query regions may change their locations over time. In this section, we classify the spatio-temporal queries based on the mutability of both objects and queries.

- **Moving Queries on Stationary Objects.** In this category, query regions are moving, while objects are stationary. An example of this category is *"As I am moving in a certain trajectory, show me all gas stations within 3 miles of my location"*.

- **Stationary Queries on Moving Objects.** In this category, the query regions are stationary, while objects are moving. Examples of these queries include *"How many trucks are within the city boundary?"* and *"Find the nearest 100 taxis to a certain hotel"*. In these queries, the query regions (city boundary and hotel neighborhood) are fixed, while the objects of interest (trucks and cars) are moving.

- **Moving Queries on Moving Objects.** In this category, both query regions and objects are moving. An example of such queries is *"As I (the sheriff) am moving, make sure that the number of police cars within 3 miles of my location is more than a certain threshold"*. In this case, the query region is moving. Also, the objects of interest (police cars) are moving.

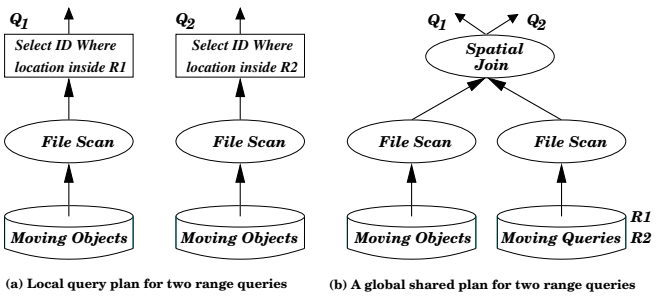


Figure 1. Shared execution of continuous queries

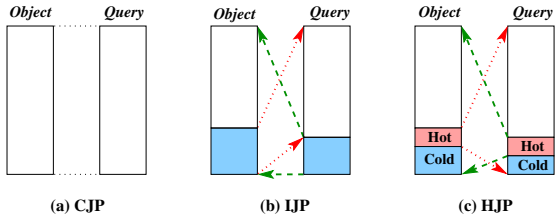


Figure 2. Illustration of join policies

3 Shared Execution

The main idea behind *shared execution* is to group all continuous spatio-temporal queries in a query table QT. In addition, we keep track of all the moving objects in an object table OT that contains the recent locations of moving objects. Then, the evaluation of a set of continuous spatio-temporal queries is abstracted as a spatial join between the moving objects table OT (table of points) and the moving queries table QT (table of query rectangles) where the join predicate is the *containment* (i.e., find all pairs of (p, r) of points and rectangles where the point p is contained in the rectangle r).

Figure 1a gives the execution plans of two simple continuous spatio-temporal queries, Q_1: *"Find the objects inside region R_1"*, and Q_2: *"Find the objects inside region R_2"*. Each query performs a file scan on the moving object table followed by a selection filter. With shared execution, we have the execution plan of Figure 1b. The table for moving queries contains the regions of the range queries. Then, a spatial join is performed between the table of objects (points) and the table of queries (regions). The output of the spatial join is split and is sent to the queries.

Shared execution has been exploited for continuous web queries in the NiagaraCQ project [6, 7], for continuous streaming queries in PSoup [5] and [10, 15], and for continuous spatio-temporal queries [16, 18]. However, the realization of the join operation depends on the underlying application. For example, in the case of web queries, the join operation can use traditional index structures. For streaming queries, indexes are not available. In this paper, we are concerned with spatio-temporal queries. Thus the join operation between the objects and queries is a spatial join.

For stationary objects (e.g., gas stations), the spatial join can be performed using a simple R-tree index [9] on the object table. Then, queries are used to probe the R-tree as range queries. In contrast, if queries are stationary and the objects are moving, an index on the queries, e.g., the Q-index [18], can be used to index the queries rather than the objects. Then, the objects are used to probe the query index to determine the queries that are satisfied by each object.

If both objects and queries are moving, then we can use a variation of the traditional R-tree that supports frequent updates (e.g., the Lazy Update R-tree (LUR-tree) [12], and the Frequently Updated R-tree (FUR-tree) [14]). In this case, the spatial join operation can be performed with two R-trees as in [3]. However, if objects and queries are highly dynamic, then it is more efficient not to use an index but use a non-index spatial join (e.g., [17, 2]).

4 Shared Execution in Location-aware Servers

In this section, we focus on the spatial join operation between the object and query tables. Ideally, the spatial join should be reevaluated as soon as an object or a query reports a change in location. However, with a large number of updates from objects and/or queries, it becomes impractical to continuously reevaluate the spatial join. Thus, we propose three join policies that aim to provide a practical realization of the shared execution paradigm in location-aware servers.

4.1 POLICY I: Clock-triggered Join Policy

We include this basic policy for comparison purposes. The main purpose of the *Clock-triggered Join Policy* (CJP, for short) is to avoid the continuous reevaluation of spatio-temporal queries. In the CJP policy, we reevaluate the spatial join every T seconds. Thus, for any evaluation time T_i, any change in the objects and/or the queries location information will not take effect until the next evaluation time $T_{i+1} = T_i + T$. The spatial join is reevaluated every T seconds by joining all the records from the object table with all the records from the query table. A larger value of T would result in having long periods of outdated results. However, smaller values of T may result in an excessive

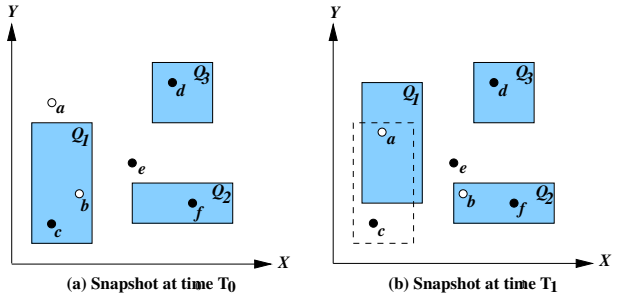


Figure 3. Example of IJP

number of computations. Typically, the interval T is around one minute [11] or 50 seconds [18]. A major drawback in the CJP policy is that at time T_{i+1}, CJP joins all objects with all queries even if most of the objects and queries did not change their location information from T_i.

4.2 POLICY II: Incremental Join Policy

The *Incremental Join Policy*, IJP for short, aims to avoid the drawbacks of the CJP policy by avoiding the recomputation of the spatial join for objects and queries that do not report any change of information from the previous evaluation time. Figure 2b sketches the IJP policy. The white parts in the object and query tables indicate the set of objects and queries that do not change their locations from the last evaluation time. The shaded parts in the tables indicate the set of objects and queries that change their locations from the last evaluation time. Then, the spatial join is performed in two steps: (1) The set of moving objects is joined with the set of stationary queries, (2) The set of moving queries is joined with all objects (stationary and moving). Notice that in the first operation, moving objects are joined with only stationary queries. This is mainly to avoid duplications in join results that would result from the second operation where moving queries is joined with moving objects.

Figure 3 gives an example of a set of objects and queries. Query Q_1 and objects a and b are the only moving query and moving objects, respectively, during the time interval $[T_0, T_1]$. According to IJP, in Step (1), a and b (moving objects) are joined with Q_2, Q_3 (stationary queries). In this step, b is added into Q_2's query result. Then in Step (2), Q_1 (moving query) is joined with a, b and c (all objects). b, c are deleted from Q's query result and a is added into Q_1's result in this step.

4.3 POLICY III: Hot Join Policy

The *Hot Join Policy* (HJP, for short) is enhanced from the incremental join policy. At every evaluation time T_i, moving entities [1] are classified into two categories, namely, *cold* and *hot entities*. An entity is identified as *cold* if its movement has no effect on the status of any query answer. On the contrary, if the entity's movement may cause changes to some query answers, the entity is identified as *hot*. In order to identify an entity as being hot or cold, we introduce the notion of a *No-Action Region* for each entity. A No-Action region is a region that the object or the query (actually, the centroid of the query) can move inside without affecting the latest result of any continuous query. At every evaluation time, we identify a moving object/query as hot if the object/centroid of the query moves out of the former calculated No-Action region. Otherwise, the object/query is identified as cold.

Figure 2c gives a sketch of HJP. The white parts in the object and query tables represent the set of objects and queries that did not report location updates from the last evaluation time. The shaded parts marked as "Hot" in the tables represent moved entities that became hot, while the shaded parts marked as "Cold" represent moved entities that remain cold. At every evaluation time, HJP executes the following steps: (1) Categorize every moved entity as hot or cold by comparing the entity's current position with the former computed No-Action region; (2) The set of hot objects is joined with all queries. During this step, the No-Action region of each hot object is recomputed; (3) The set of hot queries is joined with all objects. During this step, the No-Action region of each hot query is recomputed.

We focus on the problem of computing the No-Action regions for objects and queries in HJP. In the case of moving queries on stationary objects, *cold* and *hot* entities apply only to queries. So HJP calculates No-Action regions only for queries. In Section 4.3.1, we propose an efficient algorithm for computing a rectangular No-Action region for every query. In the context of stationary queries on moving objects, the concept of *cold* and *hot* entities apply only to objects and it is natural to compute a No-Action region for each object. We discuss the algorithm that computes a rectangular No-Action region for each object in Section 4.3.2. In Section 4.3.3, we further introduce the notion of *Adaptive No-Action regions* to deal with moving queries on moving objects, and propose an algorithm to calculate adaptive No-Action regions. Section 5 demonstrates that the proposed No-Action region algorithms out-

[1] An entity is either an object or a continuous query.

Procedure NoActionRegionCalc(q, $Oset$) **Begin**
//Input: the query q and the object set $Oset$;
//Output: the No-Action region of q.

1. Divide the data space into nine non-overlapped regions with Q's boundaries and boundary extensions.
2. $D_{e,s,w,n} = INFINITY$. //$D_{e,s,w,n}$ are the maximum distances q can move in the east, south, west or north directions, respectively.
3. For each object o in $Oset$:
   (a) Based on which region o lies at, compute $d_e(o,q)$, $d_s(o,q)$, $d_w(o,q)$ and $d_n(o,q)$; //$d_e(o,q)$, $d_s(o,q)$, $d_w(o,q)$ and $d_n(o,q)$ are the maximum distances that q can move in the east, south, west or north directions, respecively, before o may change q's answer.
   (b) $D_e = \min(D_e, d_e(o,q))$;
   $D_s = \min(D_s, d_s(o,q))$;
   $D_w = \min(D_w, d_w(o,q))$;
   $D_n = \min(D_n, d_n(o,q))$;
4. q's No-Action region is formed by the rectangle (q_c.x - D_w, q_c.y - D_s, q_c.x + D_e, q_c.x + D_n), where q_c is the centroid of q. //Assume q is represented by ($q.xlow$, $q.ylow$, $q.xhigh$, $q.yhigh$).

End.

Figure 4. Pseudo code for computing the No-Action region of a query

perform other algorithms, e.g., the *Validity Region* algorithm [21] and the *Safe Region* algorithm [18] when used in conjunction with HJP.

4.3.1 Calculating No-Action Regions for Queries

A No-Action region is to be used inside HJP for identifying hot or cold entities. In this section, we propose an algorithm for computing the No-Action region of each query in the context of moving queries on stationary objects.

Assume that entities can move only along axis directions. Movement in any other direction can be projected to movements in both x- and y-axis directions. Our algorithm determines the maximum distances a query q can move in every axis direction with the guarantee that no object will enter or leave q's query region. Let D_e, D_s, D_w and D_n denote the maximum distances in the east, south, west or north directions, respectively. For simplicity, let D_i be any one of the above four distances. Then q's No-Action region is constructed as a rectangle surrounding the centroid of q with extents being D_i distances in the corresponding axis directions. It remains to show how to compute the

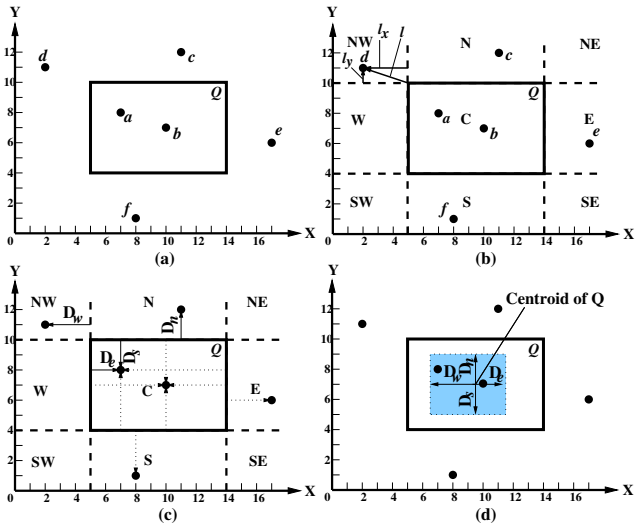


Figure 5. Example for computing the No-Action region of a query

four D_i values.

Figure 4 gives the pseudo code for computing the No-Action region for a query q. The whole space is divided into nine non-overlapped regions by q's boundaries and boundary extensions (Step 1 in Figure 4). These regions are marked as E, S, W, N, C, SE, SW, NW and NE.[2] Figure 5(a) gives an example with one moving query Q and six stationary objects $a - -f$. Figure 5(b) illustrates how Q's boundaries and their extensions divide the data space. Let $d_e(o,q)$, $d_s(o,q)$, $d_w(o,q)$, $d_n(o,q)$ be the maximum distance that query q could move towards the east, south, west, north directions, respectively, before q encounters the object o that may change q's answer. We use $d_i(o,q)$ to refer the above four values. Then D_e (D_s, D_w, D_n) of q is the minimal value of $d_e(o,q)$ ($d_s(o,q)$, $d_w(o,q)$, $d_n(o,q)$) among all objects, respectively.

Depending on the region, $d_e(o,q)$, $d_s(o,q)$, $d_w(o,q)$ and $d_n(o,q)$ values are computed in different ways. For an object o, if o lies in region C, $d_e(o,q)$($d_s(o,q)$, $d_w(o,q)$, $d_n(o,q)$) is the shortest distance between o and q's east (south, west, north) boundary. If o lies in region W, $d_w(o,q)$ is the shortest distance between o and q's west boundary, and $d_e(o,q)$, $d_s(o,q)$ and $d_n(o,q)$ are assigned to infinity since o could not be hit by boundaries of q if q moves towards the east, south or north directions. Similarly, if o lies in the regions E, S or N, then $d_e(o,q)$, $d_s(o,q)$ or $d_n(o,q)$, respectively, is the shortest distance between o and q's east, south or

[2]The letters E, S, W, N and C indicate the regions in the *east*, *south*, *west*, *north* or *Central* directions, respectively.

north boundary, and all other three values are assigned to infinity. In the case o is in any "corner" region, i.e., in SE, SW, NW or NE, our algorithm computes the four $d(o,q)$ distances as follows. Assume that there is a vector l pointing from the vertex of q that is nearest to o. Then l is the shortest path from q to o. Projecting l to the x- and y-axes, we get the two projected vectors l_x and l_y. If $|l_x|>|l_y|$, depending on the direction of l_x, $d_e(o,q)$ or $d_w(o,q)$ value is set to $|l_x|$. If $|l_x|=|l_y|$, depending on the direction of l_x and l_y, two of the four $d_i(o,q)$ values are set to $|l_x|$. All the other unassigned $d_i(o,q)$ values are set to infinity. For example, consider the object d in Figure 5(b). In Figure 5(b), l is the shortest path from Q to o. Since $|l_x|>|l_y|$ (3>1) and the direction for l_x is to the west, $d_w(d,Q)$ is set to 3 and $d_e(d,Q)$, $d_s(d,Q)$, $d_n(d,Q)$ are set to infinity. Observe that before q's boundary hits o, q needs to move at least $|l_x|$ along l_x's direction or $|l_y|$ along l_y's direction. By considering only the projected distances only, the above technique avoids complex computations.

Initially, all the four D values are assigned to infinity. (Step 2 in Figure 4). These values are then updated by the four $d(o,q)$ values of every object(Step 3 in Figure 4). Figure 5(c) illustrates the values of $d(o,q)$ (in dotted lines) and the final D values (in solid lines). Then, the No-Action region of q is constructed as a rectangle surrounding the centroid of q with D distances as extents in the corresponding axis directions (Step 4 in Figure 4). In Figure 5(d), the shaded area represents the No-Action region of Q.

Practically, objects that are far from q have little effect on q's No-Action region. Only the objects near query q need to be processed. An expanded query q' from q serves as a pre-filter before computing the No-Action region. Hence, only the objects inside q' are considered. In that case the initial D values are set as the distances between q' and q in the corresponding axis directions.

4.3.2 Calculating No-Action Region for Objects

In the case of stationary queries on moving objects, a No-Action region is computed for each object. Here we adapt the algorithm in Section 4.3.1 to compute No-Action region for object in contrast to queries. We redefine $d_e(o,q)$, $d_s(o,q)$, $d_w(o,q)$ and $d_n(o,q)$ to be the maximum distance that object o can move towards the east, south, west, and north directions, respectively, before o needs a check for answer changes with any continuous query.

Each stationary query q divides the data space into nine regions. Then, the four $d_i(o,q)$ values are determined similarly as in Section 4.3.1. The only difference is that the present $d_w(o,q)$ is equivalent to the former $d_e(o,q)$ in computing a querys No-Action region because now o is moving to q, and not the reverse. Similarly, the present $d_e(o,q)$, $d_n(o,q)$, $d_s(o,q)$ are equivalent to the former $d_w(o,q)$, $d_s(o,q)$, $d_n(o,q)$ values, respectively. Let D_e, D_s, D_w, D_n be the minimal $d_e(o,q)$, $d_s(o,q)$, $d_w(o,q)$, $d_n(o,q)$ values among the queries. Then D_i values are determined by checking o against nearby stationary queries. After determining the D values, the object No-Action region is formed as a rectangle whose lower-left and upper-right vertexes are $(o.x-D_w,\ o.y-D_s)$ and $(o.x+D_e,\ o.y+D_n)$. Figure 6 gives an example that computes the No-Action region for an object o with four stationary queries Q_1 to Q_4 surrounding o. The computed four D values are showed in Figure 6(a) and the No-Action region is formed as the shaded region in Figure 6(b).

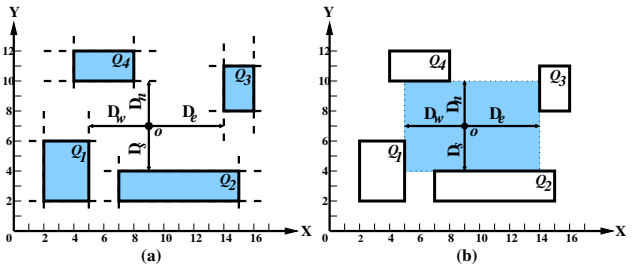


Figure 6. Computing the No-Action region of an object

4.3.3 Calculating Adaptive No-Action Regions

A No-Action region is static and is not applicable in the case of moving queries on moving objects. In this section, we extend the notion of No-Action region to *Adaptive No-Action Region* that can be used in HJP to identify hot or cold entities where both objects and queries are moving. Adaptive No-Action regions are calculated for both objects and queries. For space limitations, we only discuss adaptive query No-Action regions.

An adaptive query No-Action region is the same as a regular query No-Action region except that the adaptive one may change its shape over time. We make an assumption that the maximum velocity V_{max} among all objects is known. Assume that at time t_0, the No-Action region for a query q is computed using the procedure $NoActionRegionCalc(q, Oset)$ in Figure 4. After some time $t_0+\Delta t$, the No-Action region is invalid because objects may have moved towards q even when

Procedure AdaptiveNARegionCalc(q, $Oset$, $t_{current}$)
Begin
//Input: the query q, the object set $Oset$ and current time $t_{current}$;
//Output: the updated adaptive No-Action region of q.

//$D_{e,s,w,n}$ are defined as in Procedure NoActionRegionCalc(q, Oset); t_{recent} is the timestamp when the No-Action region of q was recently computed or updated. t_{recent} equal to 0 means q's No-Action region was never computed.

1. If (t_{recent} is 0) then {compute q's No-Action region as in Procedure NoActionRegionCalc(q, Oset); store D_e, D_s, D_w, D_n; store coordinates of q's current centroid C; store $t_{recent} = t_{current}$; return;}
2. $\Delta D_{max} = V_{max} \times (t_{current}$ - $t_{recent})$.
3. $D_e = D_e$ - ΔD_{max}; $D_s = D_s$ - ΔD_{max}; $D_w = D_w$ - ΔD_{max}; $D_n = D_n$ - ΔD_{max};
4. if any of D_e, D_s, D_w, $D_n \leq 0$, then { q's No-Action region is set to $null$; $t_{recent} = 0$; return;}
5. q's No-Action region is updated as the rectangle (C.x - D_w, C.y - D_s, C.x + D_e, C.y + D_n); store D_e, D_s, D_w, D_n; store $t_{recent} = t_{current}$; return;

End.

Figure 7. Pseudo code for computing adaptive query No-Action region

q did not move.

By changing the size of a No-Action region, the query No-Action region can be revalidated at time $t+\Delta t$. During the time interval Δt, the distance an object moves does not exceed the distance ΔD_{max}=$V_{max} \times \Delta t$. This suggests that if the former No-Action region shrinks at all axis directions by ΔD_{max}, we get a conservative No-Action region. Then, if the centroid of q remains in the updated No-Action region, q is considered a cold query with respect to the time $t+\Delta t$. q is considered a hot query if q's centroid lies out of the updated No-Action region.

The algorithm for computing adaptive query No-Action regions is listed in Figure 7. Figure 8 gives an example for computing adaptive No-Action region for query Q. In Figure 8(a), the query rectangle of Q is drawn as a solid line and the initial No-Action region of Q with respect to four moving objects a to d at time step 0 is drawn as a shaded area. Assume that V_{max} is one axis unit per one time step. Figure 8(b) and Figure 8(c) show the updated No-Action region at time steps 1 and 2, respectively. The centroid of Q remains in the updated No-Action region at time step 1 and 2, so Q is cold and avoids checking for answer changes. Figure 8(d) shows that at time step 3, the centroid of Q has moved out of the updated No-Action region, so

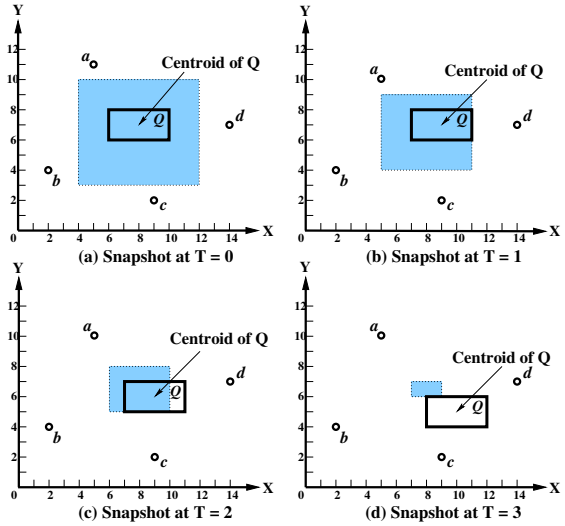


Figure 8. Adaptive query No-Action region example

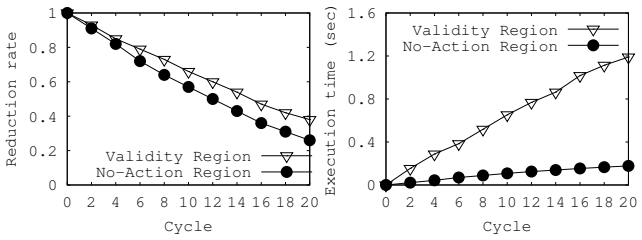
Q becomes hot. In this case, Q must check for answer updates and recomputes a new No-Action region based on Qs current position.

5 Performance Evaluation

In this section, we evaluate the performance of policies and algorithms dealing with continuous queries. Section 5.1 describes the experimental settings. In Section 5.2, we study and compare the performance of HJP when No-Action region algorithms, validity region [21] and safe region [18] are utilized. In Section 5.3, we investigate the performance when the server employs different join policies.

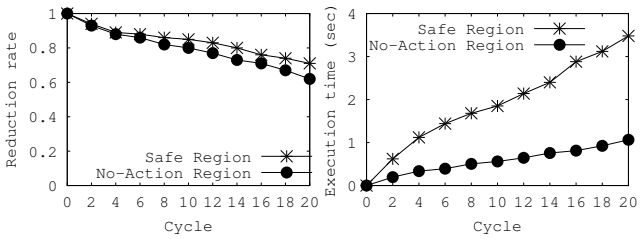
5.1 Experimental Settings

All experiments are performed on Intel Pentium IV CPU 1.4GHz with 256MB RAM running Linux 2.4.4. If not stated otherwise, the data set consists of 100,000 objects and 10,000 continuous range queries uniformly distributed in the unit square. The maximum velocity of entities is set to 0.00007 as in [11] and [18]. Query sizes are assumed to be square regions of side 0.01. One time step (or cycle) is taken to be 50 seconds as in [18]. At each time step, a pre-set percentage of objects and/or queries is randomly chosen to move with pre-assigned velocities. R-trees are implemented and built on both objects and queries. For all our experiments, the page size is 2KB and the first two levels of R-trees are assumed to reside in main memory. We focus only on the joining performance and ignore the



(a) Reduction Rate (b) Execution Time

Figure 9. No-Action region vs Validity region



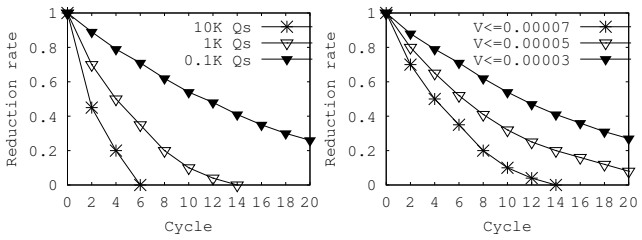
(a) Reduction Rate (b) Execution Time

Figure 10. No-Action region vs Safe region

updating costs of R-trees.

5.2 Performance of HJP

First we study the performance when the server is running HJP policy. Figure 9 compares the performance when the query No-Action region algorithm and the validity region algorithm [21] are respectively used to identify hot/cold entities. All objects are stationary while all queries are moving at every cycle. Figure 9(a) plots the reduction rate comparison up to 20 cycles. The reduction rate is the fraction of moved entities that are within their validity region/safe region/No-Action region. Once a query moves out of No-Action region/validity region/safe region, it remains outside of that region from then on. In Figure 9(a), there are more queries staying in their validity region than in their No-Action region at every cycle, which is because the validity region is more "precise" than No-Action region. However the difference in reduction rate between the validity region and No-Action region is lower than 15% within 20 cycles. Figure 9(b) shows the cumulative execution time when using validity region or No-Action region for HJP. The execution time consists of the time of joining hot queries with objects and



(a) Reduction Rate (b) Reduction Rate

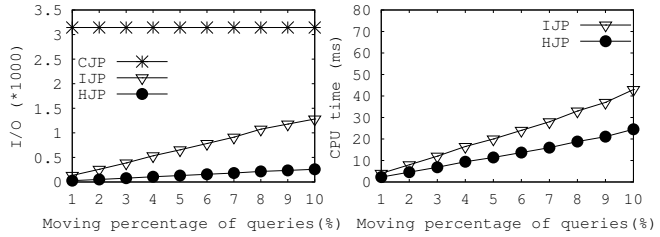
Figure 11. Adaptive No-Action region for objects

the time of recomputing validity regions/No-Action regions. The cumulative time to compute the validity region is about six times higher than the time to compute the No-Action region. This is mainly because at each cycle, queries that have moved out of their validity region/No-Action region need to recompute a new validity region/No-Action region, and the computation cost of a validity region is much higher than that of a No-Action region.

Figure 10 compares the performance when the server runs HJP when the object No-Action region algorithm and the safe region algorithm [18] are used, respectively. Only rectangular safe regions are compared. All queries are stationary and all objects are moving at every cycle. Figure 10(a) gives a comparison of the reduction rate while the Figure 9(b) gives the cumulative execution times of HJP. Similar to the comparison with validity regions, the No-Action region exhibits a reduction rate near that of safe region and outperforms the safe region in terms of the total execution time when used in HJP.

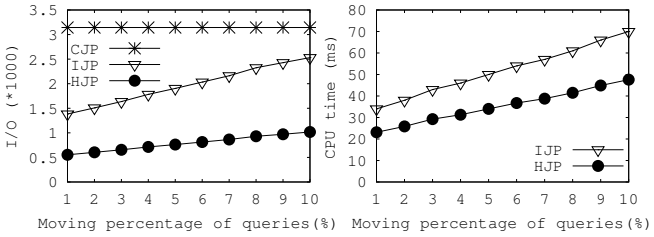
Figure 9 and Figure 10 demonstrate that the No-Action region is more efficient than validity region and safe region when used in HJP due to the much smaller computation cost. So we use No-Action regions with HJP in later experiments.

Next, we exploit some properties of the adaptive No-Action region. Figure 11(a) gives the reduction rate of adaptive No-Action regions for objects when the number of queries changes from 0.1K to 10K. We observe that when the number of queries becomes smaller, more objects are moving inside their adaptive No-Action regions. This is because when less queries exist in the space, the density of queries becomes lower. Thus, the initial object No-Action region as well as the updated No-Action regions are generally larger. An object is more likely to remain in the No-Action region. Figure 11(b) studies the relationship between the reduction rate of adaptive No-Action regions for objects and



(a) I/O (b) CPU Time

Figure 12. Moving queries on stationary objects



(a) I/O (b) CPU Time

Figure 13. Moving queries on moving objects

the maximum velocity V_{max} of queries. The number of queries is set to 1K and V_{max} varies between 0.00003 to 0.00007. In Figure 11(b), the adaptive No-Action region shows better performance when V_{max} is smaller. This is because at each cycle, the No-Action region shrinks in all directions by ΔD_{max}=V_{max}×Δt. Given that Δt is fixed, a smaller V_{max} results in a smaller ΔD_{max} and thus a larger No-Action region, so the object is more likely to stay in the adaptive No-Action region longer. The performance of Adaptive No-Action regions for queries exhibits similar properties and is omitted here for brevity.

5.3 Performance of Join Policies

We compare the performance when the server runs the three join policies discussed in this paper. Figure 12 presents the query evaluation costs in one evaluation cycle when the server runs CJP, IJP, or HJP. The percentage of moving queries at every cycle varies from 1% to 10% while objects are always static. Both I/O cost and CPU costs are compared. The I/O cost is measured as the number of disk I/Os without assuming any buffering effects, and the CPU time is measured as the time consumed in the memory. For CJP, an R-tree based spatial join algorithm is implemented [3]. For IJP and HJP, each moving (or moving hot) query exploits the object R-tree once. Figure 12(a) shows the I/O performance of the three policies. The naive CJP policy has constant number of I/O regardless the percentage of moving queries, where every time the whole query table is joined with the whole object table. As the percentage of moving queries increases, the number of I/Os increases for both IJP and HJP policies. However, the relative enhancement in performance of HJP over IJP increases. This is because when more queries are moving, there are more queries remaining in the No-Action region and do not need to search the object R-tree. Figure 12(b) shows the CPU costs for IJP and HJP (CJP is omitted because its CPU cost is about two orders of magnitude higher than the other two). In Figure 12(b), even when considering the CPU cost of recomputing the No-Action region, HJP still has much lower CPU cost than IJP policy by avoiding many join operations. The case for stationary queries on moving objects exhibits the same performance trends and is omitted for brevity.

Figure 13 presents the performance comparison of the three policies in the case of moving queries on moving objects. In Figure 13, the percentage of moving objects is fixed to 1% and the percentage of moving queries varies from 1% to 10% at every cycle. For IJP and HJP, moving (or moving hot) objects (queries) are used to probe the query (object) R-tree. Figure 13(a) and Figure 13(b) present the I/O and CPU performance, respectively. Again, CPU cost for CJP is omitted. In both Figure 13(a) and (b), HJP has the lowest I/O and CPU costs because most of moving entities still remained in their No-Action region and avoid joining costs. The cost saving between IJP and HJP results from the effect of adaptive No-Action region. Experiments using various combinations of percentages of moving objects and queries are conducted, and their results are similar to Figure 13. We conclude from the experiments that IJP outperforms CJP for all combinations of moving and stationary queries, and HJP outperforms both IJP and CJP.

6 Related Work

Most of the research on continuous query processing in spatio-temporal databases focus on efficiently evaluating one outstanding continuous query (e.g., see [13, 19, 20, 22]). Concurrent execution of a set of outstanding spatio-temporal queries is recently addressed for centralized [18] and distributed environments [4, 8]. However, for the centralized environ-

ments [18], the focus is only on processing monitoring queries, e.g., stationary queries on moving objects. Distributed environments [4, 8] assume that the moving clients have the capability to share the processing with location-aware servers.

The concept of *No-Action region* generalizes the concepts of *Validity region* [21] and *Safe region* [18]. The *Validity* region concept is introduced in [21] as the region that a moving query can move freely inside without affecting its results. The validity region concept works only for the case of stationary objects. The *Safe* region concept is introduced in [18] as the region that a moving object can move freely inside without affecting the result of any outstanding continuous query. The *Safe* region concept works only with stationary queries. A similar concept is recently introduced as the *Safe period* [8]. The *Safe* period is computed for each moving object with respect to every stationary or moving query. Our proposed *No-Action region* concept generalizes the other concepts where it works in the case of both objects and queries are moving and is scalable to large numbers of moving objects and queries.

7 Conclusion

Location-aware servers are characterized by large numbers of moving and stationary objects, and by large numbers of spatio-temporal queries. In this paper, we use *shared execution* to process similar spatio-temporal continuous queries. By utilizing shared execution, the problem of processing concurrent continuous spatio-temporal queries is abstracted as a spatial join problem. Three query join policies (the *Clock-triggered Join Policy*, the *Incremental Join Policy* and the *Hot Join Policy*) are proposed under the shared execution paradigm. CJP joins all objects with all queries at every evaluation time. IJP only checks answer changes for moved entities and avoid joining between unmoved objects and unmoved queries. HJP enhances over IJP by ignoring moved entities that do not affect query answers. The *No-Action region* is introduced to identify *hot* and *cold* entities in HJP. Efficient algorithms are proposed to compute No-Action region for both objects and queries. Experiment evaluation demonstrates No-Action region is more efficient than validity region [21] or safe region [18] when used in hot join policy. Experiments comparing different policies show that HJP outperforms both the CJP and IJP join policies in both I/O and CPU costs.

References

[1] W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Pervasive Location Aware Computing Environments (PLACE). http://www.cs.purdue.edu/place/.

[2] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable Sweeping-Based Spatial Join. In *VLDB*, 1998.

[3] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. In P. Buneman and S. Jajodia, editors, *SIGMOD*, 1993.

[4] Y. Cai, K. A. Hua, and G. Cao. Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. In *Mobile Data Management, MDM*, 2004.

[5] S. Chandrasekaran and M. J. Franklin. Streaming Queries over Streaming Data. In *VLDB*, 2002.

[6] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *ICDE*, 2002.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.

[8] B. Gedik and L. Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *EDBT*, 2004.

[9] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

[10] M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, 2003.

[11] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Mobile Objects. In *PODS*, 1999.

[12] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *Mobile Data Management, MDM*, 2002.

[13] I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic Queries over Mobile Objects. In *EDBT*, 2002.

[14] M.-L. Lee, W. Hsu, C. S. Jensen, and K. L. Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB*, 2003.

[15] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[16] M. F. Mokbel, W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Towards Scalable Location-aware Services: Requirements and Research Issues. *GIS*, November 2003.

[17] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD*, 1996.

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

[19] Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *SSTD*, 2001.

[20] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *VLDB*, 2002.

[21] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based Spatial Queries. In *SIGMOD*, 2003.

[22] B. Zheng and D. L. Lee. Semantic Caching in Location-Dependent Query Processing. In *SSTD*, 2001.