PLACE*: A Distributed Spatio-temporal Data Stream Management System for Moving Objects

Xiaopeng Xiong Hicham G. Elmongui Xiaoyong Chai Walid G. Aref

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398
{xxiong, elmongui, chai, aref}@cs.purdue.edu

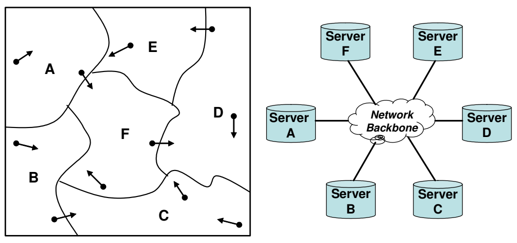
Abstract

Moving objects equipped with locating devices can report their locations periodically to data stream servers. With the pervasiveness of moving objects, one single server cannot support all objects and queries in a wide area. As a result, multiple spatio-temporal data stream management systems must be deployed and thus result in a server network. It is vital for servers in the network to collaborate in query evaluation. In this paper, we introduce PLACE, a distributed spatio-temporal data stream management system for moving objects. PLACE* supports continuous moving queries that hop among multiple regional servers. To minimize the execution cost, a new* Query-Track-Participate *(QTP) query processing model is proposed inside PLACE*. In the QTP model, a query is continuously answered by a querying server, a tracking server, and a set of participating servers. In this paper, we focus on distributed query plan generation, query execution and update algorithms for answering continuous range queries and continuous k-Nearest-Neighbor queries in PLACE* using QTP. An extensive experimental study is presented to demonstrate the effectiveness of the proposed algorithms on the scalability of PLACE*.*

1. Introduction

With the advances of locating technologies and mobile devices, moving objects are able to report their locations periodically to data stream servers while they move in space. Based on collected location information, spatio-temporal data stream management systems are deployed to answer continuous queries over moving objects.

Due to the pervasiveness of moving objects, a single data stream server can not sustain excessive numbers of moving objects and continuous queries for a wide area. As a result, a wide area is usually divided into smaller geographical regions each of which is covered by a regional data stream server. Each regional server communicates with only local objects and processes only local queries within the server's coverage region. Consequently, the regional data stream servers form a server network. Figure 1(a) gives an example where the entire space is divided to six regions A to F. Figure 1(b) shows a network of six regional data stream servers each of which covers a corresponding region given in Figure 1(a). An object reports its location periodically to the server covering its current location. Note that an object may switch the server to which it reports based on the object's location as it moves.

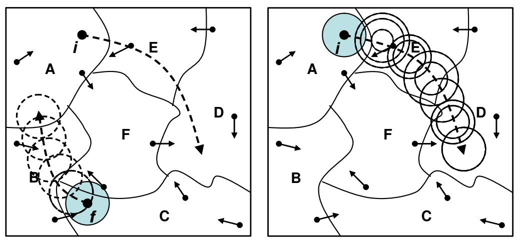


(a) Spatial Partitions (b) Regional Servers

Fig. 1. Distributed Regional Servers

Spatio-temporal server networks bring new challenges for continuous query processing. To illustrate the challenges and thus motivate our work, we consider the following illustrative query example. The first query is a continuous range query over a data stream server network while the second query is a continuous k-Nearest-Neighbor (kNN) query over the network.

Query I. Refer to Figure 2(a). Assume that in a battlefield, a commander may issue the following query q_1: *"Continuously, inform Commander i with all friendly units that are within ten miles from Soldier f"*. In Figure 2(a), the circles represent the query region at different times as f moves. q_1 has the following characteristics: (1) q_1 must be



(a) q_1: Range Query (b) q_2: kNN Query

Fig. 2. Example: Continuous Queries

answered collectively and continuously by regional servers whose coverage regions overlap with q_1's query region. (2) During f's move, the overlapping regions between q_1 and regional servers continuously change. Further, the set of regional servers that q_1 hops among dynamically changes as some servers become overlapped and some other servers no long overlap with q_1. (3) Possibly, the focal object f resides in a regional server that is different from the server of the query issuer i. To enable query updating, effective mechanisms must be established between the server of f and the server of i. (4) Moving objects including i and f may change their regional servers as they move. Proper handoff procedures must be designed to ensure the continuity and correctness of query processing as objects and/or query issuers move from one regional server to another.

Query II. Refer to Figure 2(b). Assume that Sheriff i wants to track the three nearest police cars during her travel in region A, E and D. She submits the following query: *"Continuously, send Sheriff* i *the position of the three nearest police cars"*. All the characteristics of q_1 apply to q_2. However, for q_2, the query size depends on the answer region (the minimal circular region containing the k nearest objects of the focal object) that changes dynamically during query execution. In Figure 2(b), the circles represent the changing answer region at different time points. The answer region changes whenever the focal object i moves. Moreover, even when i remains stationary, the answer region keeps changing due to movements of data objects. Consequently, the set of servers that collaborate in answering the query dynamically changes.

Motivated by the above challenges, we develop the PLACE* system, a distributed spatio-temporal data stream management system over moving objects. PLACE* supports distributed continuous spatio-temporal queries over a network of regional spatio-temporal data stream servers (PLACE servers). Query processing in PLACE* is based on a unique *Query-Track-Participate* (QTP) model. In QTP, a regional server collaborates in answering a query q based on the server's role(s) with respect to q, i.e., a *querying server*, a *tracking server*, or a *participating server* to q. The QTP model is scalable and is designed to minimize communication cost while avoid computation bottlenecks. Based on the QTP model, efficient distributed query processing and query updating algorithms are proposed. PLACE* supports objects moving among regional servers while keeping continuity and correctness of query processing. This is achieved by providing *query handoff* procedures to ship partial query plans from old regional servers to new ones. To the best of the authors' knowledge, PLACE* is the first system that supports continuous spatio-temporal queries over moving objects in distributed data stream management systems.

In this paper, we focus on the distributed continuous query processing algorithms in PLACE*. The contributions of this paper can be summarized as follows.

- We introduce the *Query-Track-Participate* (QTP) model for distributed continuous query processing inside the PLACE* system.
- We propose efficient algorithms for continuous range query and continuous kNN query processing in PLACE*. Specifically, the algorithms cover initializing, executing and updating distributed query plans. Efficient handoff algorithms are also proposed to support queries and/or objects that switch servers.
- We present a comprehensive set of experiments that demonstrate the scalability and effectiveness of the PLACE* system.

The remainder of this paper is organized as follows. Section 2 highlights the related work. Section 3 gives an overview to the PLACE* system. In Section 4, we present the algorithms for processing distributed continuous range queries in PLACE*. In Section 5, we present the algorithms to process distributed continuous kNN queries. Experimental evaluations of the PLACE* system are given in Section 6. Finally, Section 7 concludes the paper.

2. Related Work

There are several research prototypes of data stream management systems. Examples of the prototypes include TelegraphCQ [8], NiagaraCQ [10], PSoup [9], STREAM [4, 21], Aurora [1], NILE [13], PIPES [7], and CAPE [24]. One common characteristic of the above systems is that moving objects and continuous queries are processed in a centralized fashion.

Distributed continuous query processing over data streams has been addressed in the literature. Distributed Eddies [30] has policies for routing tuples between operators of an adaptive distributed stream query plan. Aurora* [11] is a distributed version of Aurora [1]. It focuses on scalability in the communication infrastructure, adaptive load management, and high system availability. Flux [26] addresses the challenges of detrimental imbalances as workload conditions change during execution of continuous queries. Borealis [2] addresses the issues of dynamically revising query results and query specifications during query execution. D-CAPE [16] extends CAPE [24] to work over a cluster of query processors using a centralized controller. D-CAPE is designed to distribute query plans and monitor the performance of each query processor with minimal communication between the controller and query processors. However, none of the previous work has addressed the challenges of processing continuous spatio-temporal queries over objects that move among distributed servers.

Recent research efforts focus on continuous query processing in spatio-temporal database management systems, e.g., answering stationary range queries [6, 23], continuous range queries [12, 32], continuous k-nearest-neighbor queries [15, 22, 27, 28, 29, 31], and generic query processing [14, 18]). In contrast to PLACE*, these works assume that all object data and queries are processed by a centralized server.

PLACE* is part of the PLACE (*Pervasive Location-Aware Computing Environment*) project [3] at Purdue University. PLACE* is a distributed data stream management system built on top of a set of regional PLACE servers [17, 18, 20, 19]. PLACE* distinguishes itself by supporting continuous spatio-temporal queries over a set of distributed regional (PLACE) servers where both queries and objects constantly move.

3. Overview of The PLACE* System

3.1. PLACE* Distributed Environment

The PLACE* distributed environment consists of a set of n regional servers. Each regional server covers some geographical region. Regions covered by two regional servers are allowed to overlap.

Home Server Each mobile object o permanently registers with one regional server. Upon registration, o gets a lifetime globally-unique identifier with its server identifier as a prefix. This is similar to what happens in a cellular phone network; a subscriber in the Greater Lafayette Area (in Indiana, USA) is assigned a phone number starting with an area code of 765. The subscriber keeps the same number even if she roams somewhere else. The permanently registered server of an object o is referred to as the *home server* of o ($HS(o)$). $HS(o)$ can be identified by simply checking the prefix of o's global identifier.

Visited Server A moving object o moves freely in space and reports its location periodically to the server covering o's current locations. The server that o currently reports to is referred to as the *visited server* of o ($VS(o)$). If o lies in a common region covered by multiple servers, o selects its visited server based on pre-defined criteria such as signal strength. When o switches its visited server (as o roams), the home server of o ($HS(o)$) is notified about this switch so that $HS(o)$ is always aware of the current $VS(o)$.

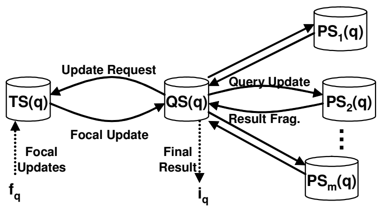


Fig. 3. The QTP Model

3.2. Regional PLACE Servers

PLACE servers [19, 20] are employed in PLACE* as regional data stream servers. Regional PLACE servers are connected with each other through high-speed reliable wired networks. Regional servers are time-synchronized. The spatial region covered by any regional server is global information. Each server periodically advertises its presence by flooding over the wired network. Inside every regional PLACE server, a *Regional Server Table* (RST) is maintained to keep information including the server identifiers, the coverage regions and the network addresses of all the servers.

A regional PLACE server processes spatio-temporal queries based on the data stream of its local region. In a PLACE server, a query is processed in an incremental manner. Based on the updates of moving objects, a PLACE server continuously outputs *positive* or *negative* answer tuples. A positive tuple implies that the tuple is to be added into the previous query answer set. A negative tuple implies that the tuple is no longer valid and is to be removed from the previous answer set. The incremental query processing algorithms inside a single PLACE server have been extensively studied in [19, 20, 18, 17]. To simplify our discussion, in the paper we view regional (participating) PLACE servers as *black boxes* that accept query registrations and output *positive/negative* answer tuples according to local streams.

3.3. The QTP Model

PLACE* processes distributed continuous spatio-temporal queries through its unique *Query-Track-Participate* (QTP) model. In the QTP model, a query q is answered collaboratively by a *querying server*, a *tracking server*, and a set of *participating servers*.

Notation	Definition
$HS(o)$	Home server of object o
$VS(o)$	Visited server of object o
$QS(q)$	Querying server of query q
$TS(q)$	Tracking server of query q
$PS(q)$	Participating server of query q
RST	Regional server table

Table 1. Table of Notations

Definition 1 *For a query q, the querying server $QS(q)$ is the regional server that q's issuer object i_q currently belongs to, i.e., $QS(q) = VS(i_q)$.*

Definition 2 *For a query q, the tracking server $TS(q)$ is the regional server that q's focal object f_q currently belongs to, i.e., $TS(q) = VS(f_q)$.*

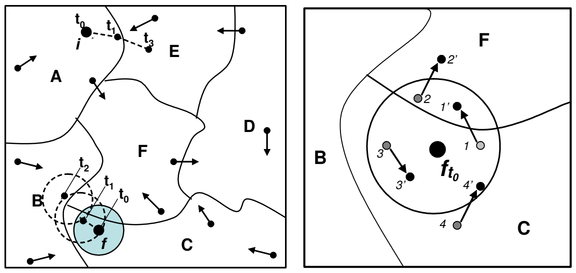
Definition 3 *For a query q, a participating server $PS(q)$ is a regional server that currently participates in answering q.*

The QTP model is depicted in Figure 3. In this figure, $PS_1(q)$, $PS_2(q)$, $\ldots PS_m(q)$ stand for m different participating servers for query q. $PS(q)$s are responsible for processing q locally within $PS(q)$s' local coverage regions. $PS(q)$s provide local query result fragments to $QS(q)$. $QS(q)$ is responsible for assembling result fragments from $PS(q)$s and transmits final assembled query result to i_q. $QS(q)$ is also responsible for updating the set of $PS(q)$s and coordinating query updates with $PS(q)$s. $TS(q)$ is responsible for tracking updates of f_q and forwarding the updates to $QS(q)$. It is worthy to mention that for a query q, a regional server may act as a combination of the above roles.

Example. Consider q_1 in Figure 2(a). When q_1 starts (refer to the shaded circle), $QS(q_1)$ is server A since q_1's issuer object i belongs to A at that time. $TS(q_1)$ is server C as q_1's focal object f belongs to C. $PS(q_1)$s include servers C and F as q_1 overlaps the coverage space of these two servers. At the last timestamp (refer to the last dashed circle), $QS(q_1)$ changes to server D as i belongs to D. Then, $TS(q_1)$ changes to server A as f belongs to A. The $PS(q_1)$s consist of server A and B as q_1 overlaps the coverage space of these two servers.

The QTP model has the following desirable properties: (1) It classifies the responsibilities of regional servers clearly. (2) It supports flexible query types by allowing the query issuer object to be different from the query focal object. (3) It avoids bottlenecks by pushing local processing down to participating regional servers. (4) It minimizes the communication cost. Users issue queries to and obtain query answers from the currently visited server without message forwarding through other servers.

The notations used throughout the paper are summarized in Table 1.



(a) Snapshot Series (b) Between t_0 and t_1

Fig. 4. Snapshots of Range Query Example

4. Distributed Continuous Range Query

In this section, we focus on distributed continuous range query processing inside PLACE*. To make the proposed algorithms generic, we assume that the query issuer of a query q is different from the query focal object of q. The proposed algorithms apply directly to the case that the query issuer object and the query focal object are identical. Further, it is straightforward to apply the algorithms to static range queries that do not move during query execution.

Throughout this section, q_1 in Figure 2(a) is used as an illustrative example. Figure 4(a) re-plots q_1 using discrete time points (t_0, t_1, etc.). Every time point represents a time when focal object f reports a new location and thus causes query updating. Without loss of generality, we assume that q_1 is issued at time t_0.

4.1. Initial Plan Generation

In PLACE*, an issuer i submits a query q to i's visited server $VS(i)$. $VS(i)$ assigns q a global query identifier with $VS(i)$'s server identifier as a prefix. Then, $VS(i)$ starts the process of generating an initial execution plan for q. This process consists of three phases, namely, *focal localization*, *assembler operator generation*, and *local plan generation*.

Phase I: Focal Localization. q's query range can be determined only after the location of the focal object f is obtained. Focal localization obtains the current location of f from f's visited server $VS(f)$. $VS(i)$ requests $VS(f)$ to send updates of f to $VS(i)$. Focal localization takes place in two round-trip steps.

1. $VS(i) \Longleftrightarrow HS(f)$: $VS(i)$ requests the server identifier of $VS(f)$ from f's home server $HS(f)$. Notice that $VS(i)$ is aware of $HS(f)$ by checking the prefix of f's life-time identifier. $HS(f)$ acknowledges by returning the server identifier of $VS(f)$ to $VS(i)$.

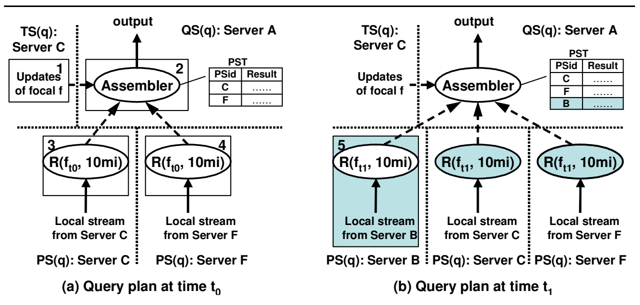


Fig. 5. Example: Query Plan Initialization/Updating

2. $VS(i) {\Longleftrightarrow} VS(f)$: $VS(i)$ subscribes f's current location from $VS(f)$. In the subscription, $VS(i)$ sends f's object identifer (oid_f) along with q's query identifier (qid_q) to $VS(f)$. Upon receiving the subscription, $VS(f)$ stores the pair of (oid_f, qid_q) in a *forwarding request table* so that future updates of f can be forwarded to $VS(i)$. $VS(f)$ acknowledges by sending f's current location back to $VS(i)$.

After focal localization, $VS(i)$ is referred to as $QS(q)$ and $VS(f)$ is referred to as $TS(q)$.

Phase II: Assembler Operator Generation. After q's query range is determined, $QS(q)$ continues to determine $PSet(q)$, i.e., the set of $PS(q)$s. $QS(q)$ searches the *Regional Server Table* (RST) using q's range. Recall that the RST stores all servers' information including coverage regions. For range queries, all regional servers whose coverage regions overlap q are included in $PSet(q)$.

$QS(q)$ generates an *assembler operator* based on $PSet(q)$. An assembler operator stores query result fragments from all participating servers $PS(q)$s and generates the final query result. The assembler operator maintains a *participating server table* (PST). For every $PS(q)$, there is one PST entry containing the local result from the corresponding $PS(q)$. A PST entry is of the form $(PSid, Result)$, where $PSid$ is the identifier of a $PS(q)$ and $Result$ is the local result set sent from a $PS(q)$.

Phase III: Local Plan Generation. After generating the assembler operator, $QS(q)$ sends q along with f's location to all the servers in $PSet(q)$. Upon receiving the request, a $PS(q)$ generates a local query plan based on the query processing engine of the regional PLACE server.

Example. Figure 5(a) gives the initialized distributed query plan for the query shown in Figure 4(a) with respect to time t_0. The plan consists of four parts at different servers. Part 1 lies in server C (serving as $TS(q)$) which forwards f's update to $QS(q)$. Part 2 lies in server A (serving as $QS(q)$) which contains the assembler operator. Notice that the PST includes one entry per $PS(q)$. Part 3 and 4 contain q's local plans in server C and F (serving as $PS(q)$s), respectively. $R(f_{t_0}, 10mi)$ represents the query region centered at f's location (at time t_0) with a radius of 10 miles.

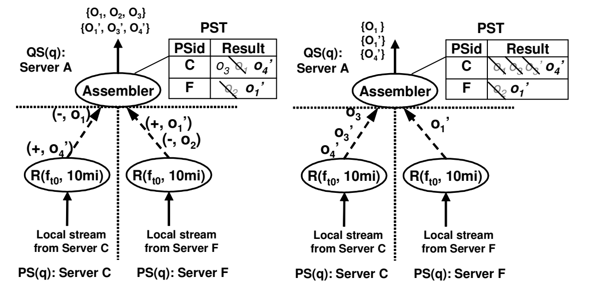
4.2. Distributed Query Execution

In this section, we present the execution algorithms for distributed range queries after the query plan has been established. We assume that the query range does not move during execution. Handling query movement is addressed in Sections 4.3 and 4.4.

After the plan for a continuous range query q is generated, $PS(q)$s treat q as a local query and process q independently based on local object streams. Then, the $PS(q)$s send incremental local results to $QS(q)$. PLACE* distinguishes two different types of range queries, namely, *non-aggregate queries* and *aggregate queries*.

Non-aggregate Range Query. This type of range query asks for moving objects within the query range without aggregations. q_1 in Figure 2(a) is an example of a non-aggregate range query. For non-aggregate range queries, the $PS(q)$s send positive and negative object tuples to $QS(q)$ directly without performing aggregations. Upon receiving an object tuple t from a $PS(q)$, the assembler operator of $QS(q)$ inserts t (positive tuple t) into or removes t (negative tuple t) from the previous result set of $PS(q)$'s PST entry, according to t's *positive* or *negative* property.

Aggregate Range Query. The second type of range query asks for aggregated result within the query range. Currently, the aggregate queries supported in PLACE* are COUNT(), MIN() and MAX(). COUNT() reports the total number of objects within the query range. MIN()/MAX() reports the object whose coordinate is the smallest/largest along the x- or y-axis within the query range. An example of a MIN()/MAX() query is to return the object whose location is west-most/east-most among all the objects within the query range. For aggregate range queries, the $PS(q)$s perform the aggregations (COUNT(), MIN(), MAX()) over local results before sending the aggregated result to $QS(q)$. Pushing aggregations down to the $PS(q)$s minimizes the communication costs between the $PS(q)$s and $QS(q)$. When the assembler operator of $QS(q)$ receives a new aggregated answer tuple t from a $PS(q)$, the assembler operator stores t in PST as the latest result from the $PS(q)$. $QS(q)$ calculates the final query result by aggregating among the local results from $PS(q)$s. For COUNT(), the final result is the sum of the local COUNT() numbers from the $PS(q)$s. For MIN()/MAX(), the final result is the object with the smallest/largest coordinate along the given axis among all the local MIN()/MAX() objects from the $PS(q)$s.



(a) Non-aggregate (b) Aggregate MAX()

Fig. 6. Example: Range Query Execution

Example. Figure 4(b) gives a snapshot for the query shown in Figure 4(a). The grey points represent the locations of four objects o_1 to o_4 at time t_0, while the black points represent the locations of the four objects at some time between t_0 and t_1.

Assume that the query given in Figure 4(b) is a non-aggregate query asking for the identifers of all objects inside the query range. Figure 6(a) illustrates the query execution process. At time t_0, the query result consists of o_1, o_3 from server C and o_2 from server F. After some time, the objects move as illustrated in Figure 4(b). When o_1 leaves server C and enters server F, server C reports a negative tuple for o_1 while server F reports a positive tuple for o_1'. Similarly, server F reports a negative tuple for o_2 when o_2 moves out of the query range. Notice that server C does not report o_3 when o_3 remains in the query range after o_3 moves. o_4' is reported by server C as a positive tuple when o_4 moves to inside the query range. Server A receives these result tuples and update its PST incrementally. At the end of the execution, the final result consists of o_3', o_4' from server C and o_1' from server F.

Now assume that the query given in Figure 4(b) is a MAX() aggregate query asking for the east-most object within the query range. Figure 6(b) illustrates the query execution process. At time t_0, the query result is o_1 by comparing o_1 (the east-most object from server C) with o_2 (the east-most object from server F). When o_1 leaves server C and enters server F, o_3 becomes the new east-most object in server C and o_1' becomes the new east-most object in server F. Therefore, o_3 and o_1' are sent to server A by server C and F, respectively. Server A then calculates o_1' as the new result. Later, the movement of o_2 does not affect the local result of server F, so no update is sent to server A. When o_3 moves, server C sends o_3' to server A to update the local result. However, this update does not cause a change in the final result. Finally, o_4' is reported by server C when o_4 moves into server C and becomes the local east-most object. By comparing o_1' with o_4, server A updates the final result to o_4'.

4.3. Query Plan Updating

When an object moves, queries focusing on this object change their query ranges. In this case, the former query plans must be updated timely based on the new query range. In this section, we concentrate on updating a query plan when the query's focal object moves within the same visited server.

Updating an existing query plan in PLACE* follows three phases, namely, *focal update forwarding*, *assembler operator updating* and *local plan updating*.

Phase I: Focal Update Forwarding. A moving object o in PLACE* periodically reports location updates to o's visited server $VS(o)$. Upon receiving o's update, $VS(o)$ looks up the *forwarding request table* and forwards the new update to all regional PLACE servers that have subscribed to o's updates. Note that one server may have subscribed to o's update for multiple times, each time for a different query. To avoid redundant forwarding, $VS(o)$ forwards every o's update to a server only once even if multiple subscriptions have been sent from this server.

Phase II: Assembler Operator Updating. $QS(q)$ updates q's assembler operator after the forwarded update of q's focal object f is received. The algorithm for updating an assembler operator at $QS(q)$ is given in Table 2. The algorithm starts by obtaining the old set of $PS(q)s$ from the PST inside the assembler operator (Step 1). Based on q's new query range, the new set of $PS(q)$s is calculated by searching the *regional server table* (RST) for the regional servers overlapping q's new range (Step 2). Comparing the old set of $PS(q)s$ against the new set of $PS(q)s$, three sets of regional servers are calculated, namely, regional servers added as new $PS(q)$s, regional servers removed as expired $PS(q)$s, and regional servers remaining as $PS(q)$s (Step 3). For regional servers newly added as $PS(q)$s, the algorithm sends to them a query registration command. The query registration command contains the query q as well as f's location. For regional servers that no longer serve as $PS(q)$s, a command is sent to terminate q's execution in these servers. For regional servers that remain in $PSet$, a query update command along with f's new location is sent to them (Step 4).

Phase III: Local Plan Updating. In this phase, commands sent by $QS(q)$ in Phase II are received by regional servers. If the query registration command is received by a regional server, the server generates a query plan locally. This process is the same as the phase of *local plan generation* in Section 4.1. If the query dropping command is received, the server terminates the query's local plan.

```
Algorithm RangeAssemUpd($Range_f$, $PST$, $RST$)
INPUT: $Range_f$: query range based on the update of $f$
       $PST$: the Participating Server Table
       $RST$: the Regional Server Table
1     $PSet_{old}$ = the set of participating servers in PST;
2     $PSet_{new}$ = the set of regional servers overlapping
                  with $Range_f$ (through searching RST);
3     Compare $PSet_{new}$ against $PSet_{old}$;
3.1       $S_{new} = PSet_{new}$ - $PSet_{old}$;
3.2       $S_{old} = PSet_{old}$ - $PSet_{new}$;
3.3       $S_{cur} = PSet_{old} \cap PSet_{new}$;
4     For every server $S$ in:
4.1       $S_{new}$: send register request for q;
               insert a new entry for $S$ in $PST$;
4.2       $S_{old}$: send drop request for q;
               drop the entry for $S$ from $PST$;
4.3       $S_{cur}$: send update request for q;
```

Table 2. Assembler Operator Updating

If the query update command is received by a regional server, the server updates q's local plan by re-calculating q's query range based on f's new location. If the query range update causes a change in the local query result, the updates to the query result are sent to $QS(q)$.

Example. Figure 5(b) gives the updated distributed query plan for the query q shown in Figure 4(a). The plan is updated at time t_1 based on f's new location. In Figure 5(b), the updated parts are plotted in shaded colors. Following the plan update process, server B is added as a new participating server as its coverage space overlaps q's new query range. Then a new entry for server B is inserted to the PST of the assembler operator. Server B generates a local plan (Part 5) for q once server B receives the query registration command from server A. Server C and F remain as q's participating servers. q's query ranges in server C and F are updated based on f's new location when the query update commands from server A are received by these two servers.

4.4. Query Plan Shipping

When an object o moves in space, o may switch its visited server when o leaves the coverage space of the old visited server ($VS_{old}(o)$) and enters the coverage space of the new visited server ($VS_{new}(o)$). Similar to cellular phone networks [5, 25], handoff procedures are carried out in PLACE* to transfer information of o between $VS_{old}(o)$ and $VS_{new}(o)$. However in PLACE*, handoff procedures need to guarantee the continuity and correctness of query processing. In PLACE*, an object o may move as a focal object of some queries and/or move as an issuer object of some other queries. Accordingly, the handoff procedure in PLACE* consists of two phases, namely, *forwarding request shipping* and *assembler operator shipping*.

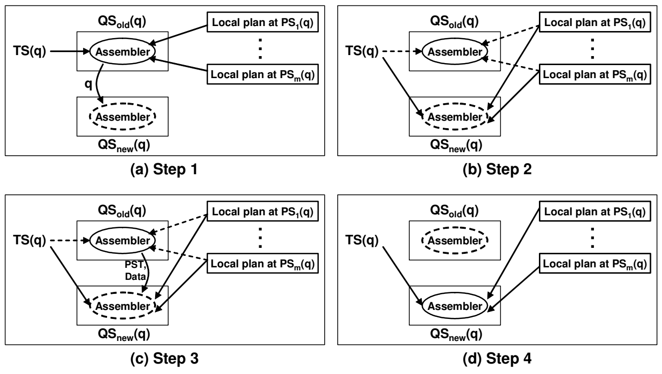


Fig. 7. Assembler Operator Shipping

Phase I: Forwarding Request Shipping. If o is the focal object of some queries, o's updates are forwarded to the corresponding querying servers by $VS(o)$. When o moves from $VS_{old}(o)$ to $VS_{new}(o)$, $VS_{new}(o)$ instead of $VS_{old}(o)$ is responsible to forward o's updates. Consider the example given in Figure 4(a), server B is responsible for forwarding f's updates to server A after time t_2 when f moves to server B. The three-step *forwarding request shipping* phase transfers the update forwarding requests regarding object o from $VS_{old}(o)$ to $VS_{new}(o)$.

S1: $VS_{old}(o)$ searches the *forwarding request table* (FRT, for short) for the corresponding entries of o. $VS_{old}(o)$ sends the found entries to $VS_{new}(o)$.

S2: $VS_{new}(o)$ inserts received entries to local FRT and acknowledges.

S3: $VS_{old}(o)$ removes the forwarding entries of o from local FRT.

Phase II: Assembler Operator Shipping. If o issues a query q, an assembler operator for q is generated in $VS(o)$. When o moves from $VS_{old}(o)$ to $VS_{new}(o)$, the assembler operator of q should be transferred from $VS_{old}(o)$ to $VS_{new}(o)$. Consider the example given in Figure 4(a), the assembler operator is transferred from server A to server E at time t_3 when i moves from server A to server E.

The *assembler operator shipping* process is performed for each query q issued by o. This process aims to minimize the suspension time of query execution. More importantly, the process guarantees that object tuples are neither duplicated nor lost during the transfer while the execution order of object tuples remains unchanged. Figure 7 illustrates the four-step *assembler operator shipping* process.

S1: $VS_{old}(o)$ sends q to $VS_{new}(o)$. $VS_{new}(o)$ generates an assembler operator that is the same as the assembler operator in $VS_{old}(o)$.

S2: $VS_{old}(o)$ notifies $PS(q)$s to send future result fragments of q to $VS_{new}(o)$. Also, $VS_{old}(o)$ notifies $TS(q)$ to send future updates of f to $VS_{new}(o)$. Later on, result fragments and focal updates sent to $VS_{new}(o)$ will be buffered in $VS_{new}(o)$ temporarily. Then, $VS_{old}(o)$ waits for the acknowledgements from the $PS(q)$s and $TS(q)$ while the assembler operator in $VS_{old}(o)$ continues to execute until the acknowledgements from $PS(q)$s and $TS(q)$ are received. As the underlying network provides reliable in-order delivery, it is guaranteed that no more local messages will be sent to $VS_{old}(o)$ after all acknowledgements are received.

S3: $VS_{old}(o)$ sends to $VS_{new}(o)$ the whole *Participating Server Table* (PST) followed by unprocessed result fragments and unprocessed focal updates.

S4: The assembler operator in $VS_{new}(o)$ starts to execute. The unprocessed data forwarded from $VS_{old}(o)$ are processed before the buffered data sent from $PS(q)$s and $TS(q)$. This guarantees the in-order execution of data tuples. At this time, the assembler operator in $VS_{old}(o)$ can be safely removed.

5. Distributed Continuous kNN Query

A continuous k-Nearest-Neighbor (kNN) query tracks the k nearest objects to a given focal object continuously. In PLACE*, the kNN query answer may consist of moving objects from multiple regional servers. In this section, we extend the range query processing algorithms to process distributed continuous kNN queries in PLACE*.

The main idea to process a kNN query q is to associate a *search region* with q. A search region for a kNN query is a circular region surrounding the query focal object. Then, only the moving objects inside the search region are considered in the final answer. By having a search region, the range query processing algorithms can be utilized for kNN query processing.

It is essential to maintain a proper search region (R_{search}) during the execution of the kNN query. If R_{search} is too small, there may not be a sufficient Number of objects inside R_{search}. On the other hand, if R_{search} is too large, a wide range of objects will be unnecessarily processed. R_{search} has to be adjusted adaptively; a fixed search region cannot always be optimal as objects move in and out of the search region dynamically. Further, R_{search} has to be updated every time when the focal object f moves.

In section 5.1, we first present the process of kNN query processing extended from the range query processing algorithms. In Section 5.2, we focus on the algorithm for obtaining a proper search region for a kNN query.

5.1. Overview of kNN Query Processing

Similar to range query processing, KNN query processing in PLACE* consists of *initial plan generation*, *distributed query execution*, *query plan updating* and *query plan shipping*.

5.1.1. Initial Plan Generation Similar to range queries, *focal localization* is first carried out to obtain the location of the focal object f when a kNN query q arrives the querying server $QS(q)$. Next, $QS(q)$ calculates an initial search region R_{search} for q. The algorithm of calculating R_{search} will be given in Section 5.2. By taking R_{search} as query range, the rest process of initial plan generation is similar to the range query process in Section 4.1. In this phase, the assembler operator is generated in $QS(q)$ and local plans are established in $PS(q)$s that are regional servers overlapping with R_{search}.

5.1.2. Distributed Query Execution A $PS(q)$ computes up to k local objects that are within R_{search} and are nearest to f as local answer. The local answer is updated continuously and is sent to $QS(q)$ incrementally. This is the same as range query execution discussed in Section 4.2. $QS(q)$ then stores the local results in the PST of q's assembler operator. Additionally, an object list $objList$ is maintained in the assembler operator. $objList$ sorts all local answer objects based on their distance to the focal object. The first k objects in $objList$ are returned as query answer.

5.1.3. Query Plan Updating kNN query updating may happen in three cases: (1) a new location of f forces a movement of R_{search}. In this case, $objList$ is re-sorted based on the new location of focal object and the query answer is updated to the first k objects in $objList$. Meanwhile, R_{search} needs to move with the focal object. (2) The total number of objects within R_{search} is less than k. In this case, R_{search} has to expand until it contains at least k objects. (3) The total number of objects within R_{search} is significantly greater than k. In this case, R_{search} can shrink to reduce processing cost. In Section 5.2, the algorithm for updating R_{search} is applicable to all the three cases. After obtain a new R_{search}, range query updating algorithms in Section 4.3 can be applied to kNN queries directly by taking the new R_{search} as the updated query range.

5.1.4. Query Plan Shipping The handoff procedure and query plan shipping algorithms presented in Section 4.4 work for kNN queries as well.

5.2. Calculating Search Region

Table 3 gives $CalcSearchRegion()$, the function calculating a search region for a kNN query. This function can be called to obtain an initial search region or to update an existing search region when focal object moves. Besides, this function is called every T seconds to adjust the search region periodically.

This function starts by getting m, the number of objects inside *objList*, and *oldRadius*, the radius of previous search region (Steps 1 and 2). Then three cases can be distinguished (Steps 3, 4 and 5). (1) $m = 0$. In this case, the function asks $VS(f)$ to process the kNN query locally and then obtains the local k_{th} NN from $VS(f)$. The radius of new search region is set as the distance between the focal object and the local k_{th} NN from $VS(f)$; This radius will be adjusted to be more "tight" in later calls of $CalcSearchRegion()$ when m changes. (2) $0 < m < k$. In this case, the search region needs to expand. By assuming that all objects are uniformly distributed in space, it is expected that k objects can be found in the new search region if the radius of previous search region is expanded by a factor of $\sqrt{\frac{k}{m}}$. Note that the new radius is further expanded by an expanding factor α. The purpose of having the expanding factor is to be more confident that k objects can be found in the new search region. Therefore, the search region is finally adjusted to $(1+\alpha) \cdot oldRadius \cdot \sqrt{\frac{k}{m}}$. (3) $m >> k$. In this case, the search region may shrink to save processing cost. The radius of new search scope is set as the radius of current answer set, that is, the distance between the focal object and the k_{th} NN. Similar to case 2, the new radius is further expanded by a factor of α. Finally, the new search region is obtained based on the calculated radius (Step 6). If the new search region is different from the old search region, a plan update procedure same as for range queries is invoked to notifies $PS(q)$s about the new search region (Step 7).

```
Function CalcSearchRegion($u_f$,objList,k,$R_{search\_old}$,$\alpha$)
INPUT: $u_f$: current location of focal object f
       objList: object list sorted on the distance to $u_f$
       k: the number of required NNs
       $R_{search\_old}$: previous search region
       $\alpha$: expanding factor to search region
1     $m$ = the number of objects in $objList$;
2     $oldRadius$ = $newRadius$ = radius of $R_{search\_old}$;
3     If ($m$ == 0)
3.1       Request $o$, the local $k_{th}$ NN of $f$ from VS(f);
3.2       $newRadius$ = $\|f - o\|$;
4     Else if ($m < k$)
4.1       $newRadius$ = $(1+\alpha) \cdot oldRadius \cdot \sqrt{\frac{k}{m}}$;
5     Else if ($m >> k$)
5.1       AnswerRadius = $\| f - k_{th}$ object in objList $\|$;
5.2       $newRadius$ = $(1+\alpha) \cdot$ AnswerRadius;
6     $R_{search\_new}$ = a circle focused at $u_f$ with $newRadius$;
7     If $R_{search\_new}$ is different than $R_{search\_old}$
7.1       Call RangeAssemUpd($R_{search\_new}$, PST, RST);
```

Table 3. Calculate Search Region

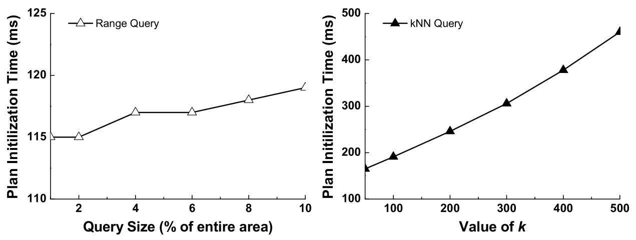
6. Performance Evaluation

PLACE* is a prototype distributed spatio-temporal data stream management system developed at Purdue University. In the experiments, the space in which the objects move is a unit square that is evenly divided into nine (3×3) square regions and each region is covered by a regional PLACE server. Each regional PLACE server runs on a dedicated Intel Pentium IV machine with dual 3.0GHz CPUs and 512MB RAM. Regional servers are connected with each other through TCP connections.

Within each regional server, a number of 50,000 local objects are uniformly generated. Local objects move only inside the coverage region of the corresponding regional server. To simulate moving objects that travel among regional servers, a *global object generator* generates 50,000 global objects that are randomly distributed and move in the entire data space. The global object generator sends location updates of global objects to regional PLACE servers according to the coverage regions of the servers. When sending an update of a global object o, the global object generator attaches the identifier of o's last regional server to the update. If o moves to a new regional server, o's new server can directly contact o's old server and start the handoff procedures given in Section 4.4. Object locations are updated every 30 seconds.

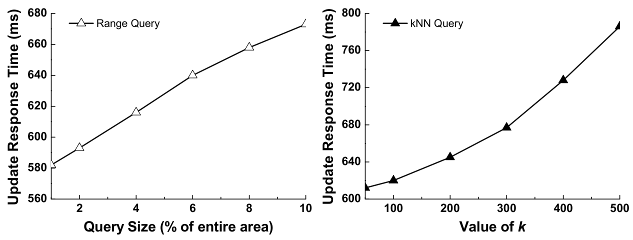
We evaluate both continuous range queries and continuous kNN queries in PLACE*. For range queries, 1,000 square-shaped range queries are generated at each regional server. We test with various query sizes ranging from 1% to 10% of the area of the entire space. For kNN queries, 1,000 kNN queries are generated on each regional server. Various values of k ranging from 50 to 500 are evaluated in the experiments. The expanding factor α of kNN queries is set as 20% and the time span to periodically adjust search region is set as 5 seconds. Focal objects and query issuer objects of both range queries and kNN queries are randomly selected from global objects residing currently in corresponding servers.

We investigate both the system response time and the number of communication messages. To reduce communication overhead, each participating server sends local results every second. Multiple result tuples can be packed in one message. The maximum size of a message is set to 1,024 bytes.



(a) Range Query (b) kNN Query

Fig. 8. **Plan Initialization Time**



(a) Range Query (b) kNN Query

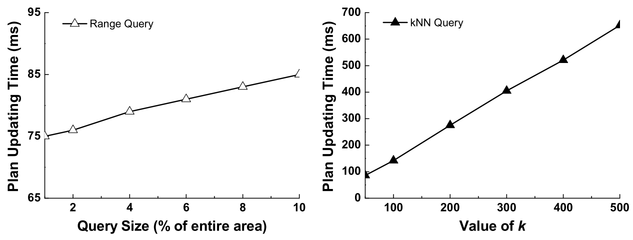
Fig. 9. **Answer Update Response Time**

6.1. System Response Time

We first evaluate the response time of PLACE*. The following four aspects are evaluated: (1) Plan initialization time, (2) Answer update response time, (3) Query update response time, and (4) Server handoff time.

6.1.1. Plan Initialization Time evaluates the time spent to establish a query plan distributed in regional servers since the query was issued by a user. Figures 8(a)-8(b) give the plan initialization time, respectively, for continuous range queries and continuous kNN queries. For range queries, the initialization time increases very slightly along with the query size. A larger query is apt to overlap more regional servers and a querying server needs to contact more participating servers. However, since local plans at participating servers are established concurrently, having more participating servers does not increase the plan initialization time apparently. For kNN queries, the initialization time increases with the value of k. According to the algorithms in Section 5.1, an initial search region is calculated by the server containing focal object. A larger k incurs higher processing time in calculating the initial search region. In both case, this setup time lasts for less than 1 sec.

6.1.2. Answer Update Response Time evaluates the elapsed time between the moment when an object update u is received at a regional server and the moment when u affects final query answer at a querying server. Figures 9(a)-9(b) give the answer update response time for range queries and kNN queries, respectively. Note that to reduce the number of messages, participating servers update local results every second. For range queries, the time increases with query size. The main reason is because when query size becomes larger, a querying server receives more answer updates per second and yields a longer processing time. Similarly for kNN queries, a querying server processes more answer updates when k becomes larger, and thus incurs a longer answer update response time. However, in both query types, this response time is also less than 1 sec for the realistic parameter ranges used, which is acceptable.

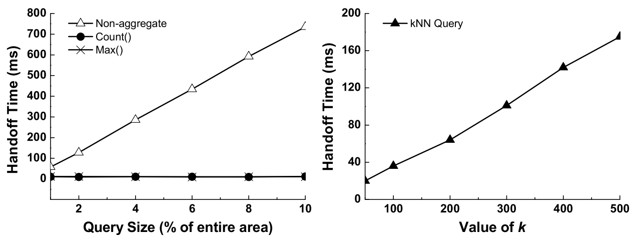


(a) Range Query (b) kNN Query

Fig. 10. **Query Update Response Time**

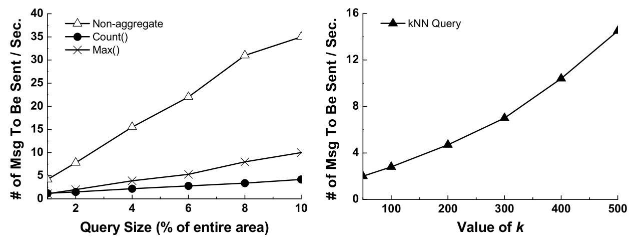
6.1.3. Query Update Response Time evaluates the elapsed time between the moment when a query q's focal object reports an update u and the moment when q's plan has been updated based on u. Figures 10(a)-10(b) give the query update response time for range queries and kNN queries, respectively. For range queries, the time increases very slightly (from 75ms to 85ms) when the query size increases from 1% to 10% of the entire space. This is because all participating servers can update local plans simultaneously after a querying server issues an update request. For kNN queries, however, a larger k results in larger query update response time. The main reason is because when focal object moves, the $objList$ must be re-sorted to find the new k nearest objects. A larger k implies more processing time to re-sort the whole list.

6.1.4. Server Handoff Time evaluates the time for a complete handoff that occurs when an object moves from an old regional server to a new region server. In these experiments,



(a) Range Query (b) kNN Query

Fig. 11. **Server Handoff Time**



(a) Range Query (b) kNN Query

Fig. 12. **Local Result Communication Cost**

we focus on query issuer objects as the most costly *assembler operator shipping* is carried out for these objects.

Figure 11(a) gives the average server handoff time for objects that have issued range queries. Three types of range queries are studied: (1) Non-aggregate queries asking for locations of objects inside the query range (referred to as *Non-aggregate* in Figure 11(a)), (2) Count() aggregate queries asking for the total number of objects inside the query range (referred to as *Count()* in Figure 11(a)), and (3) Max() aggregate queries asking for the east-most object inside the query range (referred to as *Max()* in Figure 11(a)). As indicated in Figure 11(a), the handoff time for non-aggregate queries increases steadily with query size. This is mainly because when the query size increases, the PST of the assembler operator contains more answer objects. Transferring a larger-sized PST old server to new server requires longer communication time. On the contrary, the handoff times for Count() and Max() queries are negligible compared to the handoff time of non-aggregate queries. This is because the PST for an aggregate query contains only aggregated results from participating servers and thus it is quite small.

Figure 11(b) gives the handoff time for objects that have issued kNN queries. Similar to non-aggregate range queries, the handoff time for kNN queries increases along with the value of k. When k becomes larger, a larger-size PST needs to be transferred, which incurs more handoff time.

6.2. Communication Cost

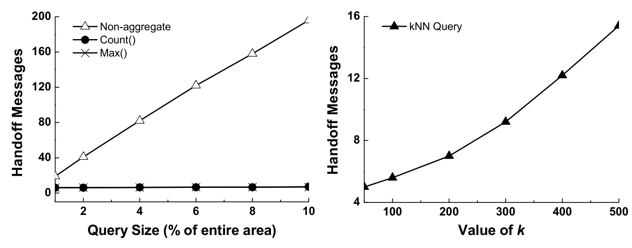
In this section, we investigate the communication cost in PLACE*. We focus on two aspects: (1) Local result communication cost, and (2) Server handoff communication cost.

6.2.1. Local Result Communication Cost In these experiments, we evaluate the number of messages sent from participating servers to querying servers when reporting local query results. Figures 12(a) and 12(b) give the average number of messages sent per second for range queries and kNN queries, respectively. As indicated in Figure 12(a), the number of messages sent for non-aggregate range queries increases with the query size. This is because more objects reside in the query range when the query size increases. For Count() queries and kNN() queries, the numbers of sent messages are much smaller than that for non-aggregate range queries. This is because only aggregated results are sent by participating servers for aggregate queries. For kNN queries, the number of messages increases with the value of k. This is because a larger k results in a larger search region. Consequently, more objects are evaluated as answer candidates and are sent to the querying server.

6.2.2. Server Handoff Communication Cost We evaluate the total number of messages incurred during a server handoff operation. Similar to Section 6.1.4, we focus on objects that have issued queries because assembler operator shipping is carried out for those objects. Figure 13(a) and Figure 13(b) give the number of messages for range queries and kNN queries, respectively. These results are consistent with the server handoff time given in Figure 11. During a handoff, non-aggregate range queries and kNN queries incur larger communication costs when the query size or the value of k increases. On the other hand, the number of handoff messages for aggregate range queries remains constantly small regardless of query size.

7. Conclusions

In this paper, we presented PLACE*, a distributed data stream management system for moving objects. PLACE* supports continuous spatio-temporal queries over multiple regional servers through the *Query-Track-Participate* model. Specifically, we have presented the algorithms for answering continuous range queries and continuous k-nearest-neighbor queries. Experimental evaluations have



(a) Range Query (b) kNN Query

Fig. 13. Handoff Communication Cost

been presented to demonstrate the scalability and effectiveness of PLACE*.

References

[1] D. J. Abadi and et al. Aurora: a new model and architecture for data stream management. *VLDB Journal*, 12(2), 2003.

[2] D. J. Abadi and et al. The design of the borealis stream processing engine. In *CIDR*, 2005.

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

[4] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3), 2001.

[5] R. Cáceres and V. N. Padmanabhan. Fast and scalable handoffs for wireless internetworks. In *MOBICOM*, 1996.

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

[7] M. Cammert, C. Heinz, J. Krämer, T. Riemenschneider, M. Schwarzkopf, B. Seeger, and A. Zeiss. Stream processing in production-to-business software. In *ICDE*, 2006.

[8] S. Chandrasekaran and et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[9] S. Chandrasekaran and M. J. Franklin. Psoup: a system for streaming queries over streaming data. *VLDB Journal*, 12(2), 2003.

[10] J. Chen and et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.

[11] M. Cherniack and et al. Scalable Distributed Stream Processing. In *CIDR*, Asilomar, CA, January 2003.

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

[13] M. A. Hammad and et al. Nile: A query processing engine for data streams. In *ICDE*, 2004.

[14] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD*, 2005.

[15] G. S. Iwerks, H. Samet, and K. Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In *VLDB*, 2003.

[16] B. Liu and et al. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, 2005.

[17] M. F. Mokbel and W. G. Aref. Gpac: generic and progressive processing of mobile queries over mobile data. In *MDM*, 2005.

[18] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, 2004.

[19] M. F. Mokbel, X. Xiong, and W. G. Aref. Continuous Query Processing of Spatio-temporal Data Streams in PLACE*. *GeoInformatica*, 9(4), 2005.

[20] M. F. Mokbel, X. Xiong, W. G. Aref, S. E. Hambrusch, S. Prabhakar, and M. A. Hammad. PLACE: A Query Processor for Handling Real-time Spatio-temporal Data Streams. In *VLDB*, 2004.

[21] R. Motwani and et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[22] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.

[23] S. Prabhakar and et al. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10), 2002.

[24] E. A. Rundensteiner and et al. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, 2004.

[25] S. Seshan, H. Balakrishnan, and R. Katz. Handoffs in cellular wireless networks: The daedalus implementation and experience, 1996.

[26] M. A. Shah and et al. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.

[27] A. P. Sistla and et al. Modeling and Querying Moving Objects. In *ICDE*, 1997.

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

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

[30] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, 2003.

[31] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In *ICDE*, 2005.

[32] X. Xiong, M. F. Mokbel, W. G. Aref, S. Hambrusch, and S. Prabhakar. Scalable Spatio-temporal Continuous Query Processing for Location-aware Services. In *SSDBM*, 2004.