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

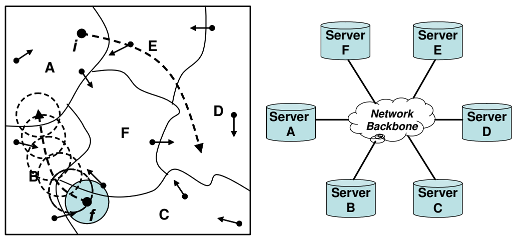
Abstract

In this paper, we introduce PLACE, a distributed spatio-temporal data stream management system for moving objects. PLACE* supports continuous spatio-temporal queries that hop among a network of 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 query plan generation, execution and update algorithms for continuous range queries in PLACE* using QTP. An extensive experimental study demonstrates the effectiveness of the proposed algorithms in PLACE*.*

1. Introduction

With the advances of location-detection technologies and mobile devices, moving objects are able to report their locations periodically to data stream servers as the objects move in space. Based on the collected location information, spatio-temporal data stream management systems (*ST-DSMS* for short) have the ability to answer continuous queries over moving objects.

Due to the pervasiveness of moving objects, one single data stream server cannot sustain excessive numbers of moving objects and continuous queries for large or dense areas. As a result, a large or a dense area is usually divided into smaller geographical regions each of which is covered by a regional data stream server. A regional server communicates with only local objects and processes only local queries within its coverage region. 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 – F.



(a) Spatial Partitions (b) Regional Servers

Fig. 1. A Network of Regional Servers

Figure 1(b) illustrates 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 the object's current location. As it moves, an object may switch the server it reports to based on the object's location.

To illustrate the new challenges of query processing in a data stream server network, we consider the continuous range query plotted in Figure 1(a). Assume that in a battlefield, a commander issues the following query q: *"Continuously, inform Commander i with all friendly units that are within ten miles from Soldier f"*. In Figure 1(a), the circles represent the query region at different times as f moves. q has the following characteristics: (1) q must be answered collectively and continuously by regional servers whose coverage regions overlap q's query region. (2) During f's move, the overlapping regions between q and regional servers continuously change. Further, the set of regional servers that q hops among dynamically changes. (3) The focal object f probably resides in a regional server 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. Handoff procedures must be designed to en-

[*] This work was supported in part by the National Science Foundation under Grants IIS-0093116 and IIS-0209120.

sure the continuity and correctness of query processing as objects move from one regional server to another.

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 set 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. Based on the QTP model, efficient distributed query processing and query updating algorithms are proposed. PLACE* supports objects and queries moving among regional servers by providing *query handoff* procedures to guarantee the continuity and correctness of query processing. 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.

The current PLACE* system supports distributed continuous range queries and k-Nearest-Neighbor (kNN) queries over a network of regional servers. Due to space limitation, in this paper we illustrate our ideas by focusing on continuous range query processing in PLACE*. For continuous kNN query processing, interested readers please refer to our technical report [31] for detailed algorithms and experimental evaluations.

The contributions of this paper are 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 processing in PLACE*. The algorithms cover initializing, executing and updating distributed query plans. Efficient handoff algorithms are also proposed to support server-switching of objects and queries.
- We present a comprehensive set of experiments that demonstrate the 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 distributed continuous range query processing in PLACE*. Experimental evaluations are presented in Section 5. Finally, Section 6 concludes the paper.

2. Related Work

There are many research prototypes of data stream management systems, for instances, TelegraphCQ [8], NiagaraCQ [10], PSoup [9], STREAM [4, 21], Aurora [2], NILE [13], PIPES [7], and CAPE [24]. The common characteristic of these systems is centralized processing of moving objects and continuous queries.

Distributed continuous query processing over data streams has been addressed in the literature. Distributed Eddies [30] studies policies for routing tuples between operators of an adaptive distributed stream query plan. Aurora* [11] 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 continuous execution. Borealis [1] addresses the issues of revising query results and query specifications during query execution. D-CAPE [16] works 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 cost. However, no previous work has addressed the challenges of processing continuous spatio-temporal queries over objects that move among a set of 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], continuous kNN queries [15, 22, 27, 28, 29, 32], and generic query processing [14, 18]). In contrast to PLACE*, these works assume that all object data and queries are processed in a centralized server.

PLACE* is part of the PLACE project [3] developed 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 regional servers where both queries and objects constantly move.

3. Overview of The PLACE* System

3.1. PLACE* Environment

The PLACE* environment consists of a set of regional servers. Each regional server covers certain geographical region. Regions covered by two regional servers may overlap.

Home Server Each moving 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

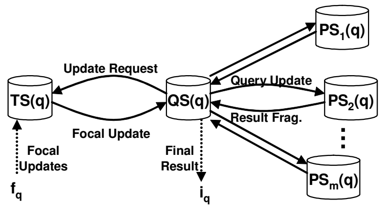


Fig. 2. The QTP Model

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 checking the prefix of o's global identifier.

Visited Server An object o moves freely and reports location periodically to the server covering its current location. 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, the home server of o ($HS(o)$) is notified about this switch so that $HS(o)$ is always aware of the current $VS(o)$.

3.2. Regional PLACE Servers

PLACE servers [19, 20] are employed in PLACE* as regional data stream servers. In a PLACE server, a query is processed in an incremental manner. A PLACE server continuously outputs *positive* or *negative* tuples. A positive tuple implies that the tuple is to be added into the previous query answer. A negative tuple implies that the tuple is no longer valid and is to be removed from the previous query answer. Incremental query processing algorithms in a single PLACE server have been extensively studied in [19, 20, 18, 17]. To simplify our discussion, in the paper we view each single regional PLACE server as a *black box* that accepts spatio-temporal query registrations and outputs *positive/negative* answer tuples according to the data streams in the local region.

Regional PLACE servers are connected with each other through high-speed reliable wired networks. Each server periodically advertises its presence along with its coverage region over the network. For each regional PLACE server, a *Regional Server Table* (RST) is maintained to keep the servers identifiers, the coverage regions and the network addresses of all the servers.

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., $QS(q) = VS(i_q)$.*

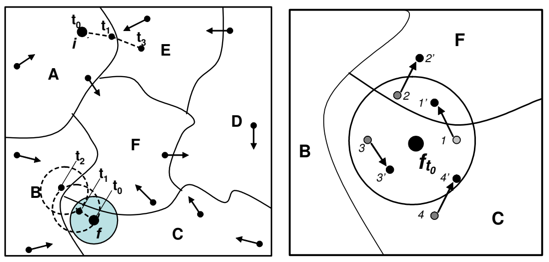
Definition 2 *For a query q, the tracking server $TS(q)$ is the regional server that q's focal object f_q currently belongs to, i.e., $TS(q) = VS(f_q)$.*

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 2. In Figure 2, $PS_1(q) \ldots PS_m(q)$ stand for m different participating servers for query q. $PS(q)$s process q locally within $PS(q)$s' local coverage regions. $PS(q)$s then transmit local query results to $QS(q)$. $QS(q)$ assembles local results from $PS(q)$s and transmits the final query result to i_q. $QS(q)$ is also responsible for updating the set of $PS(q)$s and for coordinating query updates with $PS(q)$s. $TS(q)$ is responsible for tracking updates of f_q and for forwarding the updates to $QS(q)$. Note that for a query q, a regional server may act as a combination of the above roles. Table 1 summarizes the notations used in the paper.

The QTP model is designed with the following desirable properties: (1) QTP supports dynamic query types where both the query focal object and the query issuer object may move continuously; (2) QTP avoids bottlenecks by pushing processing down to the participating servers; (3) QTP minimizes the communication cost: users issue queries to and obtain query answers from the currently visited server without message forwarding through other servers.

Example. Consider q in Figure 1(a). When q starts (refer to the shaded circle), $QS(q)$ is server A since q's issuer object i belongs to A at this time. $TS(q)$ is server C as q's focal object f belongs to C. $PS(q)$s include servers C and F as q overlaps the coverage space of these two servers. At the last timestamp (refer to the last dashed circle), $QS(q)$ changes to server D as i belongs to D then. $TS(q)$ changes to server A as f belongs to A. The $PS(q)$s consist of server A and B as q overlaps the coverage space of these two servers.



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

Fig. 3. Snapshots of Range Query Example

4. Continuous Range Query Processing

We focus on distributed continuous range query processing in PLACE*. To make our discussion generic, we assume that the query issuer is different from the query focal object. Throughout this section, q in Figure 1(a) is used to illustrate our ideas. Figure 3(a) re-plots q using discrete time points (t_0, t_1, etc.). At each time point, the focal object f reports a new location and thus causes a query update. We assume that q is issued at time t_0.

4.1. Initial Plan Generation

When an issuer i submits a query q to i's visited server $VS(i)$, $VS(i)$ assigns q a global query identifier. Then $VS(i)$ generates an initial execution plan for q in the following three phases.

Phase I: Focal Localization. q's query range can be determined only after the location of the focal object f is obtained. In focal localization, $VS(i)$ requests the most recent location of f from f's visited server $VS(f)$. 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. Then $HS(f)$ acknowledges with the server identifier of $VS(f)$.
2. $VS(i) \Longleftrightarrow VS(f)$: $VS(i)$ subscribes f's future updates from $VS(f)$. In the subscription, $VS(i)$ sends f's object identifer ($f.oid$) along with q's query identifier ($q.qid$) to $VS(f)$. Then $VS(f)$ stores the pair of ($f.oid$, $q.qid$) in a *forwarding request table* so that future updates of f can be forwarded to $VS(i)$. $VS(f)$ acknowledges with f's most recent location.

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. $QS(q)$ 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. 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 accepts local query results from all $PS(q)$s and generates a final query result. An assembler operator maintains a *participating server table* (PST). For each $PS(q)$, there is a corresponding entry in the PST table. A PST entry is of the form $(PSid, Result)$, where $PSid$ is the identifier of a $PS(q)$ and $Result$ is the most recent local result sent from the $PS(q)$.

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

Example. Figure 4(a) gives the initialized query plan for the query shown in Figure 3(a) at time t_0. The plan consists of four parts. 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. 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 (with respect to time t_0) with a radius of 10 miles.

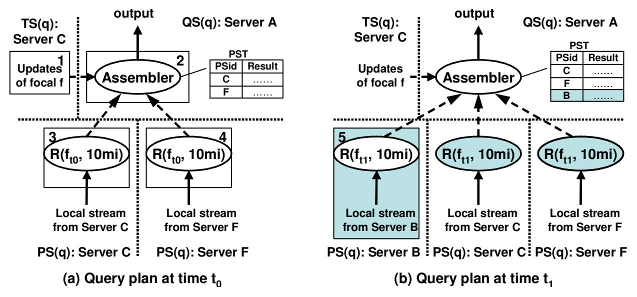


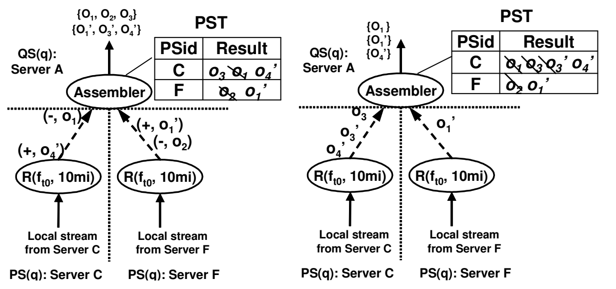
Fig. 4. Example: Plan Initialization/Updating

4.2. Distributed Query Execution

In this section, we describe the execution process for distributed range queries. 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. For non-aggregate range queries, the $PS(q)$s



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

Fig. 5. Example: Query Plan Execution

send positive and negative object tuples to $QS(q)$ directly. 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 aggregations over local results before sending to $QS(q)$. When $QS(q)$ receives a new aggregated answer tuple t from a $PS(q)$, the assembler operator stores t in PST. $QS(q)$ calculates the final query result by aggregating over local results of $PS(q)$s.

Example. Figure 3(b) gives a snapshot for the query shown in Figure 3(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 Figure 3(b) represents a non-aggregate query that reports the identifiers of all objects inside the query range. Figure 5(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 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. o'_4 is reported by server C as a positive tuple when o_4 moves to inside the query range. Finally, the query result is updated as o'_3, o'_4 from server C and o'_1 from server F.

Now assume that Figure 3(b) represents a MAX() aggregate query asking for the east-most object within the query range. Figure 5(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, servers C and F update their local results to o_3 and o'_1, respectively. Server A then calculates o'_1 as the new result. When o_2 moves, no update is sent as the movement does not affect the local result of server F. When o_3 moves, server C updates the local result to o'_3. Finally when o_4 moves into server C, o'_4 is reported by server C as local result. Then 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 section, we concentrate on updating a query plan when the query's focal object moves within the same server.

In PLACE*, updating an existing query plan follows the following three phases:

Phase I: Focal Update Forwarding. An object o periodically reports location updates to $VS(o)$. 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 o's updates.

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). 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 3 and 4).

Phase III: Local Plan Updating. 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. In the case that 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

```
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

the local query result, the updates to the query result are sent to $QS(q)$.

Example. Figure 4(b) gives the updated distributed query plan at time t_1 for the query q shown in Figure 3(a). In Figure 4(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. 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*.

Phase I: Forwarding Request Shipping. 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. The three-

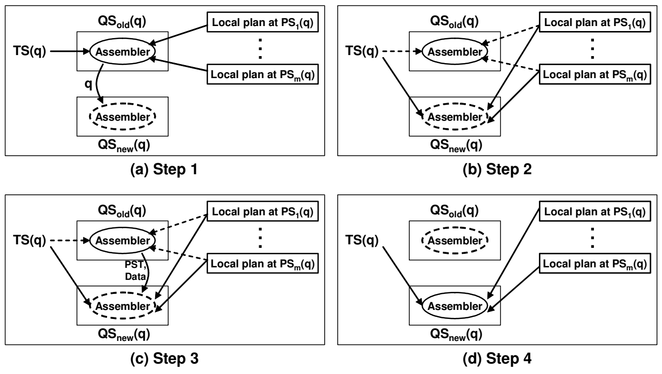


Fig. 6. Assembler Operator Shipping

step *forwarding request shipping* phase transfers the forwarding requests regarding 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 $VS_{new}(o)$.

Figure 6 illustrates the four-step *assembler operator shipping* process. 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.

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 results of q to $VS_{new}(o)$. Meanwhile, $VS_{old}(o)$ notifies $TS(q)$ to send future updates of f to $VS_{new}(o)$. Future results and focal updates sent to $VS_{new}(o)$ will be temporarily buffered in $VS_{new}(o)$. Next, $VS_{old}(o)$ waits for acknowledgements from the $PS(q)$s and $TS(q)$. The assembler operator in $VS_{old}(o)$ continues executing until the acknowledgements are received. As the underlying network provides reliable in-order delivery, it is guaranteed that no more 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. Performance Evaluation

In the experiments, the data space is a unit square that is evenly divided into nine (3 $\times$ 3) square regions. Each region is covered by a regional PLACE server running 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 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. Object locations are updated every 30 seconds.

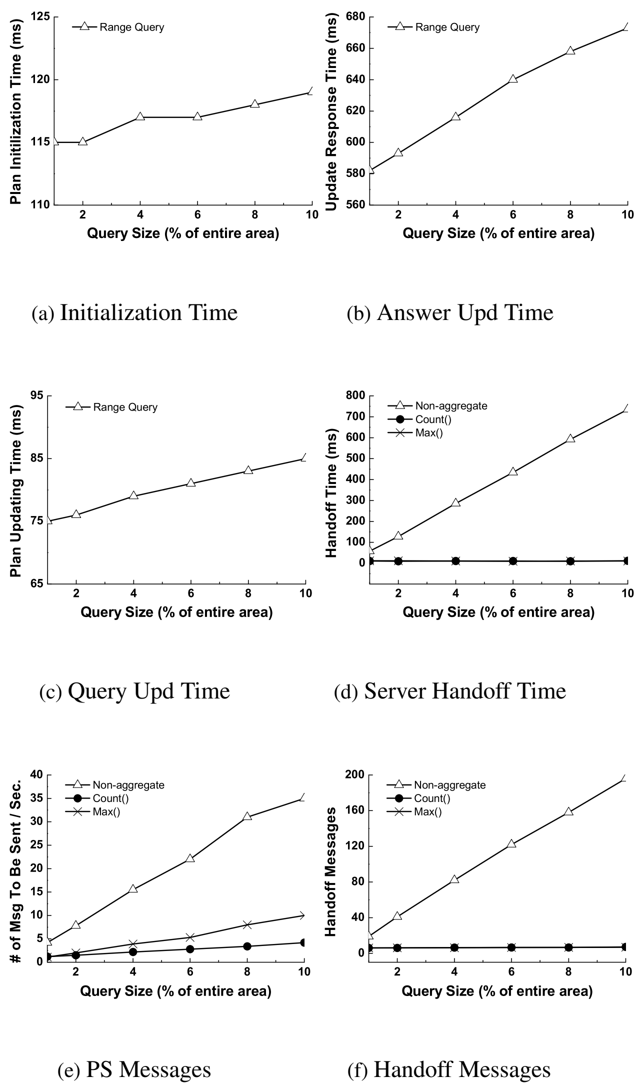
1,000 square-shaped range queries are generated at each regional server. Focal objects and issuer objects of range queries are randomly selected from global objects. We test with various query sizes ranging from 1% to 10% of the area of the entire space. To reduce communication overhead, each participating server sends local results every second. Multiple result tuples are packed in one message. The maximum size of a message is 1,024 bytes.

In our experiments, we observe that one single PLACE server cannot sustain the total loading as stated above, i.e., 500,000 objects and 9,000 queries in total. On the contrary, PLACE* exhibits satisfactory performance. This observation witnesses the scalability of the PLACE* system.

5.1. System Response Time

Plan Initialization Time evaluates the time spent to establish a distributed query plan since the query is issued. As given in Figure 7(a), the initialization time for range queries increases very slightly along with the query size. A larger query is apt to overlap more regional servers. However, since local plans of participating servers are established concurrently, having more participating servers only increases the plan initialization time slightly.

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 the final query answer at a querying server. As given in Figure 7(b), the time for range queries 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. For the worst case, the response time is less than 680 milliseconds for the realistic range parameters.



(a) Initialization Time (b) Answer Upd Time

(c) Query Upd Time (d) Server Handoff Time

(e) PS Messages (f) Handoff Messages

Fig. 7. Experimental Evaluations of PLACE*

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. As indicated in Figure 7(c), the update response time increases very slightly (from 75ms to 85ms) along with query size. This is because all participating servers can update local plans simultaneously after a querying server issues an update request.

Server Handoff Time evaluates the time for a complete handoff when an object switches its server. Three types of range queries are studied: (1) Non-aggregate queries (referred to as *Non-aggregate*), (2) Count() aggregate queries (referred to as *Count()*), and (3) Max() aggregate queries (referred to as *Max()*). As indicated in Figure 7(d), 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.

5.2. Communication Cost

Local Result Communication Cost evaluates the number of messages sent from participating servers to querying servers. Figure 7(e) gives the average number of messages sent per second for range queries. The number of messages sent for non-aggregate queries increases rapidly with the query size. This is because more objects reside in the query range when the query size increases. For Count() and Max() queries, the numbers of messages are much smaller as only aggregated results are sent by participating servers.

Server Handoff Communication Cost evaluates the total number of messages incurred during a server handoff operation. Figure 7(f) gives the number of messages for range queries. During a handoff, non-aggregate range queries incur linear increase in communication costs when the query size increases. On the other hand, the number of handoff messages for aggregate range queries remains constantly small regardless of query size.

6. 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. Experimental evaluations demonstrate that PLACE* is scalable and efficient in processing distributed continuous spatio-temporal queries.

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

[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*, pages 56–66, 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*, page 168, 2006.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. 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):140–156, 2003.

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

[11] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. 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, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. S. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *ICDE*, page 851, 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, Y. Zhu, M. Jbantova, B. Momberger, and E. A. Rundensteiner. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, pages 1338–1341, 2005.

[17] M. F. Mokbel and W. G. Aref. Gpac: generic and progressive processing of mobile queries over mobile data. In *MDM*, pages 155–163, 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*, pages 1377–1380, 2004.

[21] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. 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*, pages 634–645, 2005.

[23] 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 Transactions on Computers*, 51(10):1124–1140, 2002.

[24] E. A. Rundensteiner, L. Ding, T. M. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.

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

[26] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.

[27] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. 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*, pages 333–344, 2003.

[31] X. Xiong, H. G. Elmongui, X. Chai, and W. G. Aref. PLACE*: A Distributed Spatio-temporal Data Stream Management System for Moving Objects. *Purdue University Department of Computer Sciences Technical Report, CSD TR06-020*, Nov 2006.

[32] 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.