Continuous Query Processing of Spatio-temporal Data Streams in PLACE

Mohamed F. Mokbel Xiaopeng Xiong Moustafa A. Hammad Walid G. Aref*

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

Abstract

The tremendous increase of cellular phones, GPS-like devices, and RFIDs results in highly dynamic environments where objects as well as queries are continuously moving. In this paper, we present a continuous query processor designed specifically for highly dynamic environments (e.g., location-aware environments). We implemented the proposed continuous query processor inside the PLACE server (Pervasive Location-Aware Computing Environments); a scalable location-aware database server currently developed at Purdue University. The PLACE server extends data streaming management systems to support location-aware environments. Such environments are characterized by the wide variety of continuous spatio-temporal queries and the unbounded spatio-temporal streams. The proposed continuous query processor mainly includes: (1) Developing new *incremental* spatio-temporal operators to support a wide variety of continuous spatio-temporal queries, (2) Extending the semantic of sliding window queries to deal with spatial sliding windows as well as temporal sliding windows, and (3) Providing a shared execution framework for scalable execution of a set of concurrent continuous spatio-temporal queries. Preliminary experimental evaluation shows the promising performance of the continuous query processor of the PLACE server.

*This work was supported in part by the National Science Foundation under Grants IIS-0093116, EIA-9972883, IIS-9974255, IIS-0209120, 0010044-CCR, and EIA-9983249.

Copyright held by the author(s).

Proceedings of the Second Workshop on Spatio-Temporal Database Management (STDBM'04), Toronto, Canada, August 30th, 2004.

1 Introduction

The wide spread of cellular phones, handheld devices, and GPS-like technology enables environments where virtually all objects are aware of their locations. Such environments call for new query processing techniques to efficiently support location-aware servers. Unlike traditional database servers, location-aware servers have the following distinguished characteristics: (1) Data are received from moving and stationary objects in the form of unbounded spatio-temporal streams, (2) Large number of continuous stationary and moving spatio-temporal queries, and (3) Any delay of the query response results in an obsolete answer. Consider a query that asks about the moving objects that lie in a certain region. If the query answer is delayed, the answer may be outdated where objects are continuously changing their locations.

Existing techniques for handling continuous spatio-temporal queries in location-aware environments (e.g., see [3, 16, 18, 31, 34, 36, 39, 40]) focus on developing specific high level algorithms that utilize traditional database servers. In this paper, we go beyond the idea of high level algorithms, instead, we present a continuous query processor that aims to modify the database engine to support a wide variety of continuous spatio-temporal queries. Our continuous query processor is implemented inside the PLACE (Pervasive Location-Aware Computing Environments) server; currently developed at Purdue University [2, 24]. The PLACE server extends both the PREDATOR relational database management system [30] and the NILE streaming database management system [15] to support efficient continuous query processing of spatio-temporal streams. In particular, the continuous query processor of the PLACE server has the following distinguishing characteristics:

1. **Incremental evaluation.** The PLACE continuous query processor employs an *incremental* evaluation paradigm by continuously updating the query answer. We distinguish between two types of updates; namely *positive* and *negative* updates [23]. A positive/negative update

indicates that a certain object needs to be added to/removed from the query answer.

2. **Spatio-temporal operators.** The PLACE continuous query processor employs a new set of spatio-temporal incremental operators (e.g., INSIDE and kNN operators) that support incremental evaluation of a wide variety of continuous spatio-temporal queries.

3. **Predicate-based Sliding Windows**: We extend the notion of sliding windows beyond time-based and tuple-count windows to accommodate for predicate-based windows (e.g., an object expires from the window when it appears again in the stream).

4. **Scalability.** We use a *shared execution* paradigm as a means of achieving scalability in terms of the number of outstanding continuous spatio-temporal queries.

The rest of the paper is organized as follows: Section 2 highlights the challenges we faced in building the continuous query processor of the PLACE server along with the related work of each challenge. In Section 3, we present an overview of the data model and SQL language used by the PLACE server. Section 4 presents different methods of expiring incoming tuples in the PLACE server. The incremental processing of continuous queries is discussed in Section 5. Section 6 discusses the shared execution of concurrent continuous queries. The graphical user interface (GUI) of the PLACE server is presented in Section 7. Section 8 introduces preliminary experimental results from the PLACE server. Finally, Section 9 concludes the paper.

2 Challenges and Related Work

In this section, we go through some of the challenges we faced while building the continuous query processor of the PLACE location-aware server. With each challenge, we present its related work.

2.1 Challenge I: Incremental Evaluation of Continuous Queries

Most of spatio-temporal queries are continuous in nature. Unlike snapshot queries that are evaluated only once, continuous queries require continuous evaluation as the query result becomes invalid with the change of information [37]. A naive way to handle continuous queries is to abstract the continuous query into a series of snapshot queries executed at regular interval times. Existing algorithms for continuous spatio-temporal queries aim to optimize the time interval between each two instances of the snapshot queries. Mainly, three different approaches are investigated: (1) The validity of the results [39, 40]. With each query answer, the server returns a *valid time* [40] or a *valid region* [39] of the answer. Once the valid time is expired or the client goes out of the valid region, the client resubmits the continuous query for reevaluation. (2) Caching the results. The main idea is to cache the previous result either in the client side [31] or in the server side [18]. Previously cached results are used to prune the search for the new results of k-nearest-neighbor queries [31] and range queries [18]. (3) Precomputing the result [18, 34]. If the trajectory of query movement is known apriori, then by using computational geometry for stationary objects [34] or velocity information for moving objects [18], we can identify which objects will be nearest-neighbors [34] to or within a range [18] from the query trajectory. If the trajectory information changes, then the query needs to be reevaluated.

With the large number of continuous queries, reevaluating a continuous spatio-temporal query, even with large time intervals, poses a redundant processing for the location-aware servers. In the PLACE continuous query processor, we go beyond the idea of reevaluating continues spatio-temporal queries. Instead, we provide an incremental evaluation paradigm, where only the updates of the result are reported to the user.

2.2 Challenge II: Wide Variety of Continuous Queries

Most of the existing query processing techniques focus on solving special cases of continuous spatio-temporal queries, e.g., [31, 34, 39, 40] are valid only for *moving queries on stationary objects*, [5, 9, 11, 25] are valid only for *stationary range queries*. Other work focus on aggregate queries [11, 32, 33] or k-NN queries [16, 31]. Trying to support such wide variety of continuous spatio-temporal queries in a location-aware server results in implementing a variety of specific algorithms with different access structures.

In the PLACE continuous query processor, we avoid using tailored algorithms for each kind of continuous spatio-temporal queries. Instead, we furnish the PLACE server with a set of primitive pipelined query operators that can support a wide spectrum of continuous spatio-temporal queries.

2.3 Challenge III: Large Number of Concurrent Continuous Queries

Most of the existing spatio-temporal algorithms focus on evaluating only one spatio-temporal query (e.g., [3, 16, 18, 31, 34, 36, 39, 40]). In a typical location-aware server [2, 21, 24], there is a huge number of concurrently outstanding continuous spatio-temporal queries. Handling each query as an individual entity dramatically degrades the performance of the location-aware server.

Although there is a lot of research in sharing the execution of continuous web queries (e.g., see [8]) and

continuous streaming queries (e.g., see [6, 7, 14]), optimization techniques for evaluating a set of continuous spatio-temporal queries are recently addressed for centralized [25] and distributed environments [5, 9]. The main idea of [5, 9] is to ship part of the query processing down to the moving objects, while the server mainly acts as a mediator among moving objects. In centralized environments, the Q-index [25] is presented as an R-tree-like [10] index structure to index the queries instead of objects. However, the Q-index is limited in two aspects: (1) It performs reevaluation of all the queries (through the R-tree index) every T time units. (2) It is applicable only for stationary queries. Moving queries would spoil the Q-index and hence dramatically degrade its performance.

2.4 Challenge IV: Indexing Moving Objects/Queries

Most of the existing spatio-temporal index structures [22] aim to modify the traditional R-tree [10] to support the highly dynamic environments of location-aware servers. In particular, two main approaches are investigated: (1) Indexing the future trajectories such that the existing tree would last longer before an update is needed. Examples of this category are the TPR-tree [29], R^{EXP}-tree [28], and the TPR*-tree [35]). (2) Modifying the deletion and insertion algorithms for the original R-tree to support frequent updates. Examples of this category include the Lazy-update R-tree [17] and the Frequently-updated R-tree [19]

Even with the proposed modifications of the R-tree structures, highly dynamic environments degrades the performance of the R-tree and results in a bad performance. In the PLACE continuous query processor, we avoid using R-tree-like structure. Instead, we use a grid-like index structure [23] that is simple to update and retrieve. Moreover, fixed grids are space-dependent, thus there is no need to continuously change the index structure with the continuous insertion and deletion.

3 The PLACE Server

In this section, we present the data modelling and SQL language used by the PLACE server.

3.1 Data Model

By subscribing with the PLACE server, moving objects are required to send their location updates periodically to the PLACE server. A location update from the client (moving object) to the server has the format (OID, x, y), where OID is the object identifier, (x, y) is the location of the moving object in the two-dimensional space. An update is timestamped upon its arrival at the server side. Once an object stops moving (e.g., an object reaches to its destination or the cellular phone is shut down) it sends to the server a *disappear* message which indicates that the object is no further moving.

Due to the highly dynamic nature of location-aware environments and the infinite size of incoming spatio-temporal streams, we cannot store all incoming data. Thus, the PLACE server employs a three-level storage hierarchy. First, a subset of the incoming data streams is stored in in-memory buffers. In-memory buffers are associated with the outstanding continuous queries at the server. Each query determines which tuples are needed to be in its buffer and when these tuples are expired, i.e., deleted from the buffer. Second, we keep an in-disk storage that keeps track with only one reading of each moving object and query. Since, we cannot update the disk storage every time we receive an update from moving objects, we sample the input data by chosing every kth reading to flush to disk. Moreover, we cache the readings of moving objects/queries and flush them once to the secondary storage every T time units. Data on the secondary storage are indexed using a simple grid structure [23]. Third, every $T_{archive}$ time units, we take a snapshot of the in-disk database and flush it to a repository server. The repository server acts as a multi-version structure of the moving objects that supports historical queries. Stationary objects (e.g., gas stations, hospitals, restaurants) are preloaded to the system as relational tables that are infrequently updated.

3.2 Extended SQL Syntax

As the PLACE server [24] extends both PREDATOR [30] and NILE [15], we extend the SQL language provided by both systems to support spatio-temporal operators. Mainly, we add the $INSIDE$ and kNN operators to support continuous range queries and k-nearest-neighbor queries respectively. A continuous query is registered at the PLACE server using the SQL:

```
REGISTER QUERY query_name AS
SELECT select_clause
FROM from_clause
WHERE where_clause
INSIDE inside_clause
kNN knn_clause
WINDOW window_clause
```

The `REGISTER QUERY` statement registers the continuous query at the PLACE server with the *query_name* as its identifier. The *select_clause*, *from_clause*, and *where_clause* are inherited from the PREDATOR [30] database management statement. The *window_clause* is inherited from the NILE [15] stream query processor to support continuous sliding window queries [14]. A continuous query is dropped form the system using the SQL: `DROP QUERY` *query_name*.

The *inside_clause* can represent stationary rectangular or circular range queries by specifying the two corners or the center and radius of the query region, respectively. If the first parameter to the *inside_clause* is set to M, then the query is moving and the second parameter represents the ID of the *focal* object of the query. Similarly, the *knn_clause* can represent stationary as well as moving k-nearest-neighbor queries.

4 Tuple Expiration

With the unbounded incoming spatio-temporal streams, it becomes infeasible to store all incoming tuples. However, some input tuples may be buffered in memory for a limited time. The choice of the stored tuples are mainly query dependent, i.e., we store only the tuples of interest. Since the queries are continuously changing, there should be a mechanism to expire (delete) some of the stored tuples and replace them with other tuples that becomes more relevant to the outstanding continuous spatio-temporal queries. In the PLACE continuous query processor, we employ three types of tuple expiration, namely, *temporal* expiration, *spatial* expiration, and *predicate-based* expiration.

4.1 Temporal Expiration

Most of the data stream management systems use the concept of temporal expiration as a mechanism to answer continuous sliding window queries. A sliding widow query involves a time window w. Any object that has a timestamp within the current sliding window of any outstanding query Q is in-memory buffered with the associated buffer of Q.

An example for a sliding window query submitted to the PLACE server is: Q_1: *"Continuously, report the number of cars that passed by region R in the last hour"*.

```
SELECT COUNT(ObjectID)
FROM MovingObjects
WHERE type = Car
INSIDE R
WINDOW 1 hour
```

Notice that Q_1 buffers all incoming tuples during the previous hour. A tuple is expired (i.e., deleted from the query buffer) once it goes out of the sliding time window (i.e., if it becomes more than one hour old).

4.2 Spatial Expiration

The PLACE server introduces a new type of expiration that depends on the spatial location of the moving objects instead of their timestamps. An incoming tuple o is stored in the in-memory buffer associated with a query Q only if o satisfies the spatial window (e.g., region) of Q.

An example of spatial expiration query is: Q_2: *"Continuously, report the number of cars in a certain area."*. Notice that unlike Q_1, in Q_2, we are concerned about the actual current number of cars not the number of cars in the recent history. The SQL of Q_2 is similar to that of Q_1 with only the removal of the window statement.

4.3 Predicate-based Expiration

Due to the nature of spatio-temporal streams, other forms of tuple expiration may arise. For example, consider the query Q_3: *"For each moving object, continuously report the elapsed time between each two consecutive readings"*. Such a query contains a self join where objects from the stream of moving objects are self joined based on the object identifier. The query buffer needs to maintain only the latest reading of each moving object. Once the reading of a certain object is reported, the previous reading is expired. We call such kind of expiration as *predicate-based* where it is mainly dependent on the query semantic.

5 Incremental Evaluation

To avoid reevaluating continuous spatio-temporal queries, we employ an *incremental evaluation* paradigm in the PLACE continuous query processor. The main idea is to only report the changes of the answer from the last evaluation time. By employing *incremental evaluation*, the PLACE server achieves the following goals: (1) Fast query evaluation, since we compute only the updates of the answer not the whole answer. (2) In a typical location-aware server, query results are sent to the users via satellite servers [1, 12]. Thus, limiting the amount of transmitted data to the updates only rather than the whole query answer saves in network bandwidth. (3) When encapsulating incremental algorithms into physical pipelined query operators, limiting the tuples that go through the whole query pipeline to only the updates reduces the flow in the pipeline. Thus, efficient query processing is achieved.

To realize the incremental evaluation processing in the PLACE server, we go through three main steps. First, we define the high level concept of incremental updates, by defining two types of updates; *positive* and *negative* updates [20, 23]. Second, we encapsulate the processing of incremental algorithms into pipelined query operators. Third, we modify traditional pipelined query operators (e.g., distinct and join) to deal with the concept of negative tuples [13].

5.1 Positive/Negative Updates

Incremental evaluation is achieved through updating the previous query answer. Mainly, we distinguish between two types of updates; *positive* updates and *negative* updates. A positive/negative update indicates

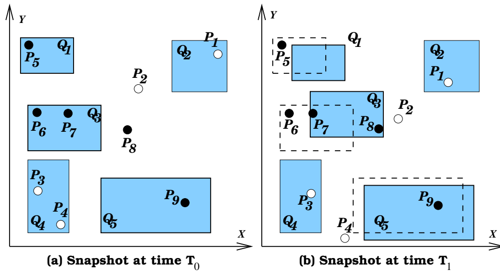


Figure 1: Incremental evaluation of range queries

that a certain object needs to be added to/removed from the query answer. A query answer is represented in the form $(QID, OList)$, where QID is the query identifier and $OList$ is the query answer. The PLACE server continuously updates the query answer with updates of the form $(QID, \pm, OID)$ where $\pm$ indicates the type of the update and OID is the object identifier.

Figure 1 gives an example of applying the concept of positive/negative updates on a set of continuous range queries. The snapshot of the database at time T_0 is given in Figure 1a with nine moving objects, p_1 to p_9, and five continuous range queries, Q_1 to Q_5. The answer of the queries at time T_0 is represented as (Q_1, P_5), (Q_2, P_1), (Q_3, P_6, P_7), (Q_4, P_3, P_4), and (Q_5, P_9). At time T_1 (Figure 1b), only the objects p_1, p_2, p_3, and p_4 and the queries Q_1, Q_3, and Q_5 change their locations. As a result, the PLACE server reports the following updates: $(Q_1, -P_5)$, $(Q_3, -P_6)$, $(Q_3, +P_8)$, and $(Q_4, -p_4)$.

5.2 Spatio-temporal Incremental Pipelined Operators

Two alternative approaches can be utilized in implementing spatio-temporal algorithms inside the PLACE server: using SQL *table functions* [26] or encapsulating the algorithms in physical query operators. Since there is no straightforward method for pushing query predicates into table functions [27], the performances is limited and the approach does not give enough flexibility in optimizing the issued queries. In the PLACE server we encapsulate our algorithms inside physical pipelined query operators that can be part of a query execution plan. By having pipelined query operators, we achieve three goals: (1) Spatio-temporal operators can be combined with other operators (e.g., distinct, aggregate, and join operators) to support incremental evaluation for a wide variety of continuous spatio-temporal queries. (2) Pushing spatio-temporal operators deep in the query execution plan reduces the number of tuples in the query pipeline. This reduction comes from the fact that spatio-temporal operators act as filters to the above operators. (3) Flexibility in the query optimizer where multiple candidate execution plans can be produced.

The main idea of spatio-temporal operators is to keep track of the recently reported answer of each query Q in a query buffer termed $Q.Answer$. Then, for each newly incoming tuple P, we perform two tests: Test I: Is P part of the previously reported $Q.Answer$? Test II: Does P qualify to be part of the current answer? Based on the results of the two tests, we distinguish among four cases:

- **Case I:** P is part of $Q.Answer$ and P still qualify to be part of the current answer. As we process only the updates of the previously reported result, P will not be processed.
- **Case II:** P is part of $Q.Answer$, however, P does not qualify to be part of the answer anymore. In this case, we report a *negative* update P^- to the above query operator. The *negative* update indicates that P is *spatially* expired from the answer.
- **Case III:** P is not part of $Q.Answer$, however, P qualifies to be part of the current answer. In this case, we report a *positive* update to the above query operator.
- **Case IV:** P is not part of $Q.Answer$ and P still does not qualify to be part of the current answer. In this case, P has no effect on Q.

5.3 Traditional Operators

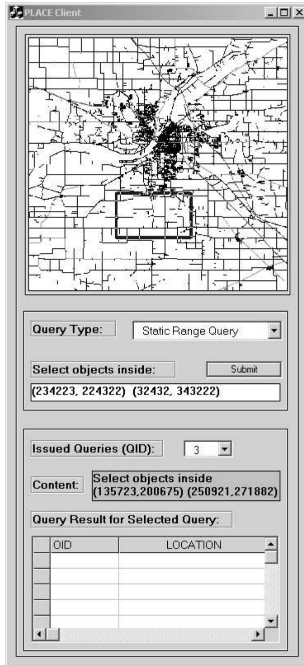
Having the spatio-temporal operators at the bottom or at the middle of the query evaluation pipeline requires that all the above operators be equipped with special handling of *negative* tuples. The NILE query processor [15] handles *negative* tuples in pipelined operators as follows:

- *Selection* and *Join* operators handle *negative* tuples in the same way as *positive* tuples. The only difference is that the output will be in the form of a *negative* tuple.
- *Aggregates* update their aggregate functions by considering the received *negative* tuple.
- The *Distinct* operator reports a *negative* tuple at the output only if the corresponding *positive* tuple is in the recently reported result.

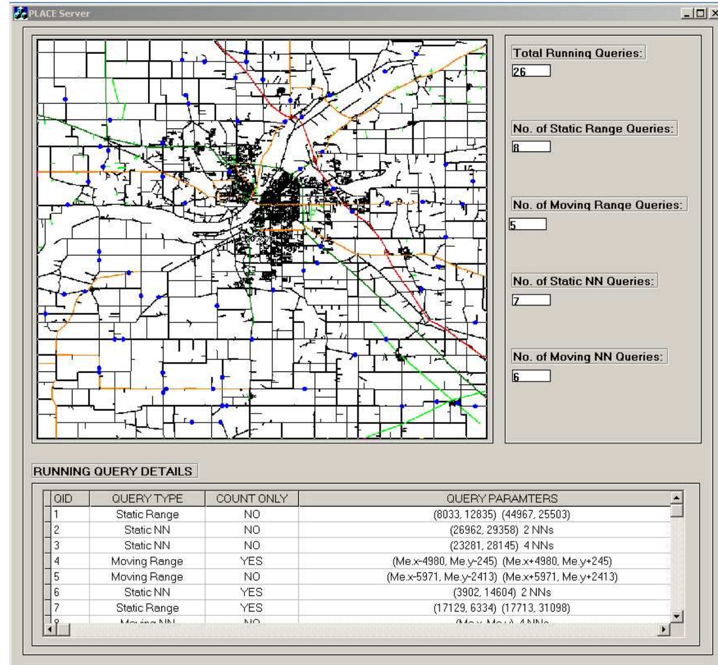
For more details about handling the *negative* tuples in various query operators, the reader is referred to [13].

6 Scalability

The PLACE continuous query processor exploits a *shared execution* paradigm [21, 23, 38] as a means for achieving scalability in terms of the number of concurrently executing continuous spatio-temporal queries.



(a) Client GUI



(b) Server GUI

Figure 3: Snapshot of the PLACE client and server

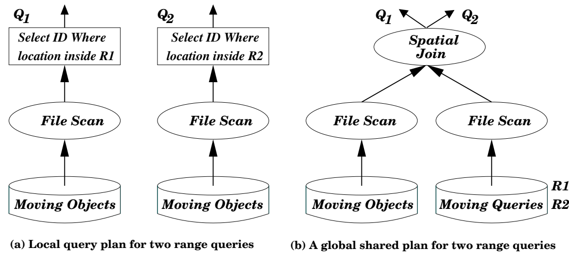


Figure 2: Shared execution of continuous queries.

The main idea is to group similar queries in a query table. Then, the evaluation of a set of continuous queries is modelled as a spatial join between moving objects and moving queries. Similar ideas of shared execution have been exploited in the NiagaraCQ [8] for web queries and PSoup [6, 7] for streaming queries.

Figure 2a gives the execution plans of two simple continuous spatio-temporal queries, Q_1: *"Find the objects inside region R_1"*, and Q_2: *"Find the objects inside region R_2"*. With *shared execution*, we have the execution plan of Figure 2b. Shared execution for a collection of spatio-temporal range queries can be expressed in the PLACE server by issuing the following continuous query:

```
SELECT Q.ID, O.ID
FROM QueryTable Q, ObjectTable O
WHERE O.location inside Q.region
```

7 User interface in PLACE

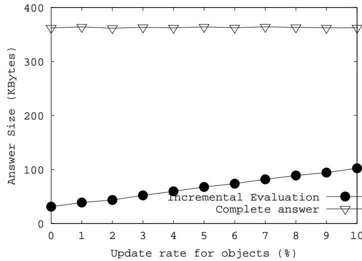
Figure 3 gives snapshots of the client and server graphical user interface (GUI) of PLACE. The client GUI simulates a client end device used by the users. Users can choose the type of query from a list of available query types. The spatial region of the query can be determined using the map of the area of interest[1] (the bold plotted rectangle on the map). By pressing the *submit* button, the client translates the query into SQL language and transmits it to the PLACE server. Once the query is submitted to the server, the result appears to the query as a list at the bottom of Figure 3a. A client can send multiple queries of different types to the PLACE server.

The PLACE server GUI is for the purpose of administration at the server side. The main idea is to keep track of the concurrently executing continuous queries from each type. All the processed queries along with their parameters are displayed in the bottom left of the screen. In addition, the server GUI contains a regional map showing the movement of objects, and the parameters of the selected queries.

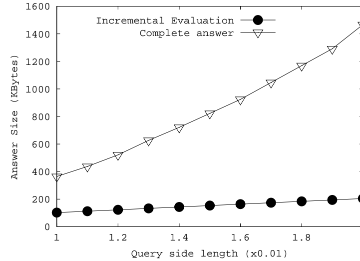
8 Performance Evaluation

In this section, we present preliminary experiments that show the promising performance of the continuous query processor in the PLACE server. We use the

[1]The map in Figure 3 is for the Greater Lafayette, IN, USA.

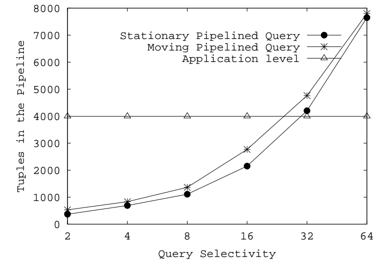


(a) Moving objects (%)



(b) Query size

Figure 4: The answer size



(a) $INSIDE$ Operator

Figure 5: Pipelined operators.

Network-based Generator of Moving Objects [4] to generate a set of 100K moving objects and 100K moving queries. The output of the generator is a set of moving objects that move on the road network of a given city. We choose some points randomly and consider them as centers of square range queries.

8.1 Size of Incremental Answer

Figure 4 compares between the size of the incremental answer returned by utilizing the incremental approach and the size of the complete answer. The location-aware server buffers the received updates from moving objects and queries and evaluates them every 5 seconds. Figure 4a gives the effect of the number of moving objects that reported a change of location within the last 5 seconds. The size of the complete answer is constant and is orders of magnitude of the size of the worst-case incremental answer. In Figure 4b, the query side length varies from 0.01 to 0.02. The size of the complete answer increases dramatically to up to seven times that of the incremental result. The saving in the answer size directly affects the communication cost from the server to the clients.

8.2 Pipelined Spatio-temporal Operators

Consider the query Q: *"Continuously report all trucks that are within MyArea"*. *MyArea* can be either a stationary or moving range query. A high level implementation of this query has only a selection operator that selects only the *"trucks"*. Then, a high level algorithm implementation would take the selection output and incrementally produce the query result. However, an encapsulation of $INSIDE$ algorithm into a physical operator allows for more flexible plans.

Figure 5 compares the high level implementation of the above query with pipelined operators for both stationary and moving queries. The selectivity of the queries varies from 2% to 64%. The selectivity of the selection operator is 5%. Our measure of comparison is the number of tuples that go through the query evaluation pipeline. When algorithms are implemented at the application level, the performance is not affected by the selectivity. However, when $INSIDE$ is pushed before the *selection*, it acts as a filter for the query evaluation pipeline, thus, limiting the tuples through the pipeline to only the incremental updates. With $INSIDE$ selectivity less than 32%, pushing $INSIDE$ before the selection greatly affects the performance.

9 Conclusion

In this paper, we present the continuous query processor of the PLACE (Pervasive Location-Aware Computing Environments) server; a database server for location-aware environments currently developed at Purdue University. The PLACE server extends both the PREDATOR database management system and the NILE stream query processor to deal with unbounded spatio-temporal streams. In addition to the temporal tuple expiration defined in sliding window queries, we maintain other forms of tuple expirations (e.g., spatial expiration). To efficiently handle large number of continuous queries, we employ an incremental evaluation paradigm that contains: (1) Defining the concept of *positive* and *negative* updates, (2) Encapsulating the algorithms for incremental processing into pipelined spatio-temporal operators, and (3) Modifying traditional query operators (e.g., distinct and join) to deal with the *negative* updates that comes from the spatio-temporal operators. Shared execution is employed by the continuous query processor as a means of achieving scalability in terms of the number of concurrently continuous queries. Preliminary experimental results show the promising performance of the PLACE continuous query processor.

References

[1] S. Acharya, M. J. Franklin, and S. B. Zdonik. Disseminating Updates on Broadcast Disks. In *VLDB*, 1996.

[2] W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Pervasive Location Aware Computing Environments (PLACE). http://www.cs.purdue.edu/place/, 2003.

[3] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *IDEAS*, 2002.

[4] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2), 2002.

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

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

[7] S. Chandrasekaran and M. J. Franklin. PSoup: a system for streaming queries over streaming data. *VLDB Journal*, 12(2):140–156, 2003.

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

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

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

[11] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. J. Tsotras. On-Line Discovery of Dense Areas in Spatio-temporal Databases. In *SSTD*, 2003.

[12] S. E. Hambrusch, C.-M. Liu, W. G. Aref, and S. Prabhakar. Query Processing in Broadcasted Spatial Index Trees. In *SSTD*, 2001.

[13] M. A. Hammad, W. G. Aref, M. J. Franklin, M. F. Mokbel, and A. K. Elmagarmid. Efficient execution of sliding-window queries over data streams. Technical Report TR CSD-03-035, Purdue University Department of Computer Sciences, Dec. 2003.

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

[15] 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. Marzouk, and X. Xiong. Nile: A Query Processing Engine for Data Streams (Demo). In *ICDE*, 2004.

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

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

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

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

[20] M. F. Mokbel. Continuous Query Processing in Spatio-temporal Databases. In *Proceedings of the ICDE/EDBT PhD Workshop*, 2004.

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

[22] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal Access Methods. *IEEE Data Engineering Bulletin*, 26(2), 2003.

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

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

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

[26] B. Reinwald and H. Pirahesh. Sql open heterogeneous data access. In *SIGMOD*, 1998.

[27] B. Reinwald, H. Pirahesh, G. Krishnamoorthy, G. Lapis, B. T. Tran, and S. Vora. Heterogeneous query processing through sql table functions. In *ICDE*, 1999.

[28] S. Saltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *ICDE*, 2002.

[29] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, 2000.

[30] P. Seshadri. Predator: A Resource for Database Research. *SIGMOD Record*, 27(1):16–20, 1998.

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

[32] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the Past, the Present and the Future in Spatio-Temporal Databases. In *ICDE*, 2004.

[33] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-Temporal Aggregation Using Sketches. In *ICDE*, 2004.

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

[35] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In *VLDB*, 2003.

[36] Y. Tao, J. Sun, and D. Papadias. Analysis of Predictive Spatio-Temporal Queries. *TODS*, 28(4), 2003.

[37] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-Only Databases. In *SIGMOD*, 1992.

[38] 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*, June 2004.

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

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