Indian Journals of Science and Technology, *Vol 9(16), DOI: 10.17485/ijst/2016/v9i16/92298, April 2016*

ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Continuous Queries for Streaming Data of Mobile Objects

Hoang Do Thanh Tung*

Institute of Information and Technology, Vietnam Academy of Science and Technology (VAST);
tunghdt@ioit.ac.vn

Abstract

As a result of the development of streaming data systems, a huge number of continuous queries from users on those systems should be managed efficiently. Recent proposed methods mostly focus on handling problems of system performance that can be degraded by overloading server and the bottle neck problem of the wireless network. Specially, continuously updating reports when objects change positions may consume battery energy of mobile devices significantly. In this paper, we propose an efficient method to manage continuous queries for streaming data of mobile objects in GPS as well as Indoor positioning systems. Our idea is to use indices and in-memory data structures for objects, queries together in order to reduce cost of communication, updates on servers and battery burden of mobile device. But the method still supports the most types of queries as possibly. Our experiments show that the proposed method can manage continuous queries in large number quickly and efficiently in a wireless environment.

Keywords: Continuous Query, GPS, Indexing, IPS, Mobile Objects, Streaming Data

1. Introduction

Streaming data could be data items that can be in relational database, XML messages, log records of systems, networks, or webpage visits, sensor data, and so on. They come from many sources to special databases to serve end users through LAN, wireless, internet. Particularly, those data arrive as streams continuously in numerous, quick, possibly uncontrollable and unlimited way. As a result, streaming data appears to introduce some new research problems. Several applications only create data streams instead of data sets, such as financial indices, activities in network monitoring and traffic management, log records or clicks, views in web tracking and data feeds from sensor networks, and others.

A Data-Stream Management System (DSMS) must be a computer software that maintains and manages continuous queries of data in streams. A continuous query has to keep executing over time whenever the stream receives new data. Thus, the results of the continuous query are up-to-date in nearly real time. There are some data-stream management systems include Spark Streaming, Hadoop Streaming, Microsoft StreamInsight, Microsoft Biztalk, IBM InfoSphere.

In contrast to traditional queries of which the results are fetched just one time for completion on the current data sets, continuous queries are continuously executing to have the result of them up-to-date over the streaming database. In applications, continuous queries can be used to monitor traffic network behavior in order to detect anomalies such as hardware errors, attacks from hackers); In BI (Business Intelligent) applications, continuous queries can be used to monitor KPIs, performance of business system and alarm abnormal signals. In LBS (Location Based Services) applications, continuous queries can be used to monitor mobile objects/users moving in static areas or moving areas. These applications are all in need of continuous queries for users who require real-time updated answers in continuous way to make decisions. It is impossible for traditional queries to work on rapid data streams. In this paper, our applications

*Author for correspondence

is to be applied that LBS applications use GPS (Global Positioning System) to locate outdoor objects and IPS (Indoor Positioning System) to locate indoor objects.

There are many methods proposed for managing continuous queries of mobile/moving objects with assumption that every object carries mobile devices such as smart phones. Within our application domain, the objects only generate new-value events without modifying, deleting events. Thus, an application's major research issue is how to efficiently manage massive amounts of mobile objects. As objects are continuously moving, the query results also may continuously change and require continuous communication between objects and a system. The challenges are as follows:

- First, the constant transmission between mobile objects and a database server can not only cause an overhead communication but also quickly make the mobile battery power exhausted.
- Second, when the number of mobile devices is very large, that excessively manipulating continuous queries to be updated possibly overwhelm the database system.
- Third, there may be diversity of continuous queries, either stationary or moving, either current or future and so on.

Greedy to address most of the above challenges, in this paper, we propose an efficient method that can gain results of continuous queries in efficient and speedy way. Our method has the properties as follows:

- To reduce the number of transmission, the method takes advantage of motion function data for moving objects. This kind of data does not require too many location updates from users and gives a database server many chances to work off line.
- To accelerate continuous query revaluating performance, we deploy indices together with linked data structures in main memory. As a result, our method can help a service server less dependent on the connection between the server and mobile devices and improve query performance.
- To deploy various types of continuous query that includes stationary (range) queries, moving queries, current and future anticipated queries.

In the rest of this paper, we reviews some related works on continuous queries in section 2. Then, we present our proposed method in section 3. Section 4 shows our experimental results. We conclude our method in section 5.

2. Related Works

Recently, motivated by LBSs, processing Continuous (range) Query (CQ) over moving objects has been a hot topic instead of processing (traditional) snapshot queries, which retrieves their results only once during their life cycle. CQs for mobile objects assume that those objects continuously send new locations to the server via wireless connections, and the server stores and updates all the results of the active queries immediately. However, in case the number of CQs as well as objects becomes enormous, the system can be overloaded because it possibly has to re-calculate all CQs for new results and the wireless network can get jam due to overwhelming communication. Thus, many methods have been proposed for processing a continuous query as well as a large number of queries efficiently.

The papers[1,2] are with the validity of the results approach. With each query answer, the server returns a valid region[1] or a valid time[2] of the answer. The valid time and the valid region indicate the temporal and the spatial validity of the returned answer, respectively. Once the valid time is expired or the client goes out of the valid region, the client resubmits the continuous query for complete reevaluation. The papers[3,4] are with the results caching approach. The main idea is to resubmit the continuous query every fixed time interval T. The recent query result is cached either in the client side[3] or in the server side[4]. Upon resubmission, the previously cached results are used to prune the search for the new results of k-nearest-neighbor queries[3] and range queries[4].

To avoid redundant location-update reports from each moving object, the safe region method was proposed in[5]. Similarly, MQM[6] was proposed to aim to reduce the communication cost and the server workload by allowing moving objects to be able to estimate their effect on the CQ's results through the concept of resident domain, containing current location of an object. SPQI[7] was proposed to index query regions instead of monitor regions in MQM. However, a moving object must know other queries through which it may go and use this information to evaluate and inform a server whether or not it moves to other queries. In the worst case, there are a lot of CQs active in the server, changing overtime. That requires large computing in mobile devices and heavy burden on the network.

The above methods can't support moving range queries. They either support very specific continuous queries or focus on processing a continuous query at a time. They do not deal with the problem of a large number of queries that is more meaning than the dealing with just one query. Q-index[8] is to manage a large number of continuous queries as a data. the Q-index uses an R-tree index structure to manage CQs. Whenever any moving object has a new location, the object will probe the Q-index to find the queries to which the object should belong. The Q-index has two disadvantages: (1) it supports only point represented data of which updates trigger reevaluation for queries; (2) It is applicable only for stationary queries. The work[9] assumed that trajectories of moving objects may be known in advance. It made Grid indices for objects and queries. The disadvantage of this method is that a CQ also must know object's trajectories/moving functions. As a result, at every time t, the CQ has to re-calculate all objects whether or not still inside it in order to request updates on a server. This work is not necessary because an object and a query only intersect each other only a certain period of time. Moreover, it is not easy to know future trajectories of objects.

3. Proposed Method

In this paper, we focus on an approach to manage high amount of continuous queries efficiently, reducing redundant updates on active CQs, heavy communication burden between mobile objects and a database system. Our approach is to use two HTPR-tree indices[10] that manage data of mobile objects and continuous queries, respectively. In particular, we propose In-memory structures that keep results of queries up-to-date when objects are moving, new objects or continuous queries are arriving, efficiently. Based on them, we have designed update algorithms with substantially lower costs than the most general case for an update.

Our contributions are as follows:

- We assume moving object's motion functions are known. With our approach, the method can support several types of continuous queries, including moving continuous queries.
- We use a combination of indices with In-memory structures for efficiently managing relationship between objects and queries. This reduces query update and communication cost significantly.

In this paper, we assume that the moving objects are moving in a two/three-dimensional plane. For each object o, a motion function f_o is associated. Normally, the current position of each object has to be transmitted whenever the object changes its position to update the related CQs in the system. In our approach, only the change of an object's velocity (direction and speed) is in need of transmission. We assume that entire of streaming data and computation for result sets of continuous queries are stored and processed on a server side.

The basic idea of our method is that while objects are moving, there is not any query-update request from the objects if none of them change their velocities. Therefore, we need a mechanism to update CQs automatically even when they don't receive any request from any object. Because we don't want mobile objects to consume their energy to compute like MQM[6], the server has to keep track of both, the moving objects and the continuous queries. Now, we should consider some following scenarios in which CQs has to be updated automatically even when there is no update request form moving objects by changing their velocities.

(1) First, a new continuous query arrives and the query result has to be computed from all current objects.
(2) Second, a new object arrives or go out of (is removed from) the plane, then continuous queries need checking the object whether become in their results or not.
(3) Third, the most generally frequent case, changes of the positions of some mobile objects can change the result of some continuous queries when they move in or out the query region.

In our approach, we use two independent indices to manage (1) and (2). Objects and queries are managed by HTPR-tree indices, respectively. We use two linked data structures to manage (3). The first structure is used for keeps track of all objects moving among CQs in the future. The second structure is used for the current result of all continuous queries. These simple data structures will automatically generate query update requests as well as re-compute several CQs when some moving objects move in/out any query. With this approach, we expect that the costs for update operations can be kept rather low. Additionally, mobile objects don't need spending power on storing query data, computing CQs as well as communicating to a server to update CQs.

In the following sub sessions, we present our proposed algorithm to manage continuous queries completely in main memory. Therefore, we have to assume that the main memory of a server stores entire the data needed for the method. Technically, the data consists of only moving objects related to queries and query's results so that it is feasible to deploy our method for real applications. Because we just apply the algorithms of HTPR-tree to index objects and queries, we will not introduce the HTPR-tree algorithms in this paper. Instead, we present the method's data structures and related algorithms that we propose.

3.1 In Main Memory Structure

Different from the managing data of current positions as points, the managing data of motion functions cannot take the advantage of position-updated events from users in order to trigger re-evaluating continuous queries active in server. In other words, in a database server managing motion functions, the position changes that impact query's results does not alert the server to re-evaluate continuous queries active in it without the change of velocities. Because of the difference, we employ two main memory structures to manage continuous queries as follows:

- **Self-triggering structure** is linked-list data stored in main memory. We call it *outerQ(ueries)*. It stores all moving objects that are going to enter any queries at any time after now as potential objects. It is used at every timestamp to detect new results for the active continuous queries. The purpose of this structure is to trigger updates on the continuous queries. Its data is an object linked list of *(Oid, mint, maxt, queries_list)* records in which *Oid* is an potential object's identifier; *mint* is the earliest time when the object enters any query in queries_list; *maxt* is the latest time when the object enters any query; *queries_list* is a query linked-list of *(Qid, int, outt)* records in which *Qid* is a query's identifier; *int/outt* is the time when the object *Oid* enters/goes out the query *Qid*. Figure 1 is a visual picture of self-triggering structure. It shows that each node of an object *Oid* is connected to a list of queries which the object is going to intersect during an interval *[int, outt]*. The figure also shows that the object list is sorted by *Oid* and the query list is sorted by *{int, outt}*. The sorting minimizes inspection processing time in following algorithms.
- **Queries-result structure** is a main memory array that keeps all results of all continuous queries valid until now or the future. We called it *innerQ(ueries)*. Its data is a query linked list of *(Qid, mint, Objects-list)* recodes in which *Qid* is a query's identifier; *mint* is the earliest time when any object goes out the query; *Objects-list* is an object linked list of *(Oid, outt)* records as the query's result in which *Oid* is an object's identifier; *outt* is the time when the object goes out the query. Figure 2 is a visual picture of Queries result structure. It shows that each node of a query *Qid* is connected to a list of objects that already stay inside the query until a time *outt*. The figure also shows that the query list is sorted by *Qid* and the object list is sorted by time *outt*. The sorting minimizes inspection processing time in following algorithms.

We keep innerQ and outerQ entirely in main memory. In case the number of potential objects is too large, we can improve the speed of inspecting potential objects in outerQ by simply organizing outerQ as a binary-tree of Oid.

3.2 The Method Algorithms

Given a set of active continuous queries, the work of our algorithms is composed of two main steps, first setting up data for outerQ and innerQ structures that are needed to re-evaluate the queries at subsequent timestamps by using an Initial-conQuery Algorithm, secondly using those structures to re-evaluate the queries without disk accesses while time are running by using a conQuery Algorithm. The former work needs repeating every after a long time period but the second need repeating at every timestamp.

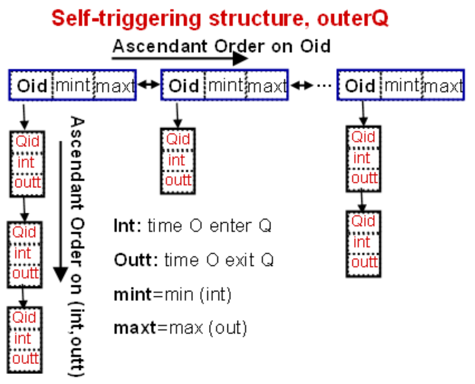

Figure 1. Self-triggering Structure, outerQ.

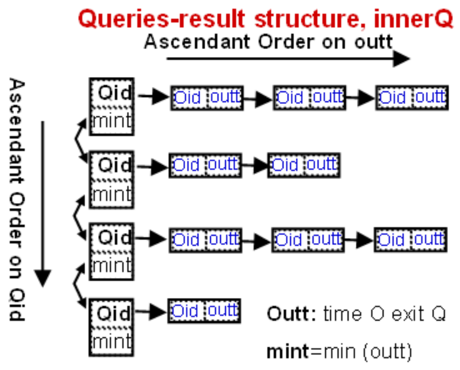

Figure 2. Queries result Structure, innerQ.

3.2.1 Initial-conQuery Algorithm

Given a number of continuous queries, we first scan the HTPR-tree to gain results in which objects already are inside the queries, and anticipated future (or potential) results in which objects potentially can intersect the queries in the future. To limit the cost of collecting the results, we employ an *until_t* time parameter. After *until_t*, we have to re-do this algorithm again in order to calculate further results. The value of *until_t* depends on a specific situation. Figure 3 shows a process in which we have a number of queries inspect/scan a HTPR-tree. At leaf nodes, every query is measured with objects to calculate *int* and *outt*, *int* is the time an object enters a rectangle and *outt* is the time the object goes out the rectangle. We can find the algorithm to calculate an interval of an intersection between a moving object and a moving query in10. In this algorithm, after inspecting the HTPR-tree, the rest work is simply to deliver the results to innerQ and the potential results to outerQ.

3.2.2 conQuery Algorithm

This algorithm operates at every timestamp to update query's results that are active in a database server. Using this algorithm, a system can inform a user new objects that need adding to or obsolete objects that need deleting form his/her query. Thank to that, the system can avoid re-sending entire content of the query to the user at every timestamp.

In this algorithm, we have to deal with two important situations. The first situation is when a database server receives a change from a moving object. Thus, an update occurs and invokes the server to process it. Our queries do not wait to be updated until the secondary storage of the server is updated as usual. The algorithm deals with the new change by itself. The Figure 4 shows the process used for the situation. The process mainly takes three steps. At first, because the object change its state (velocities), the algorithm fetch the object in outerQ and delete it as well as its query list from outerQ. At second, the algorithm inspects all queries in innerQ except their object lists in order to calculate intervals (int, outt) of intersections between the new state of the object as long as all continuous queries still active. At third, after calculating the intervals and updating queries in innerQ, the algorithm inserts the object as well as its query list into outerQ. The system can use this information to send {Q, +O} or {Q, -O} to each user of the queries. +O means the object O needs adding; -O means the object O needs deleting.

The second situation is that the server does not receive any velocity-update from objects. Despite that, some continuous queries still possibly change their results because objects are changing their positions. In this case, the algorithm has to trigger updating query results by itself. Figure 4 shows the process used for this situation. The process is composed of two periods. The first period is to find new objects that entered any query already at a time t. Given a time t, this work inspects objects in outerQ. If any object has *mint* less than t, the object must be inside any query in its query list and the work will dig down the query list to find out those queries. Searching the queries containing the object at t is very quick because the query lists are all sorted by *{int, out}*, thereby stopping as soon as meeting either a query with *int* greater than t or an object with *mint* greater than t. The other parameters *outt* and *maxt* are used to check whether or not an object goes out a query. In other words, *"maxt < t"* means the object goes out all queries and *"outt < t"* means the object only goes out the query. If new objects are found in outerQ, the objects will be deleted from outerQ and added into innerQ. The system can use this information to send {Q, +{O1,O2, .., Ok} } to each user of the queries.

The second period is to find obsolete objects that went out of any query already at t. The work inspects all queries in *innerQ*. If meeting a query that has *mint* less than t, there must be any object in its object list that has gone already, the work will dig down the object list to find out those objects. Searching obsolete objects at t is very quick because the object lists are all sorted by *outt*, thereby stopping as soon as meeting either a query with *mint* greater or equal to t or an object with *outt* greater or equal to t. if obsolete objects are found in innerQ, the objects will be deleted from inner Q forever. The system can use this information to send {Q, -{O1,O2, .., Ok} } to each user of the queries.

4. Experiments

We built up entire data structures in main memory. Besides that, we used a fundamental index to support **initial-conQuery** algorithm in fetching result and potential objects. The index is HTPR-tree[10]. The disk page size is set to 1k bytes, and the maximum number of entries in a node is 27 for all indexes. Our goal is to compare (continuous query + update) performance between our method and a similar method, Q-index[8]. However, because Q-index doesn't index on motion-function data, we have to change its algorithms to index motion-function data by replacing Rtree with TPR-tree, we call TPR-Q index.

We use a real spatial dataset LA[11] to initiate positions and directions of MOs at timestamp 0. We assume that mobile cars/objects randomly distributed in 2d area where each axis of the area is normalized to [0, 10.000 meters]. Then, every object is associated with a VBR such that (a) object does not change spatial extents during its movement, (b) every car can run at most of about 90km/h, velocity can be positive or negative as two opposite directions with equal probability. A time unit is 3 seconds. At each time unit, every object can randomly change its speed, direction if it reaches an intersection, or even nothing.

Each query q has three parameters: $q_R$len, $q_V$len, and $q_T$len, such that (a) its MBR q_R is a spare, with length

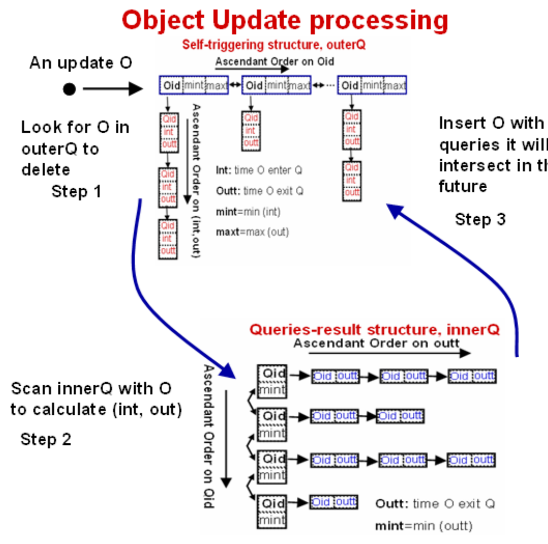

Figure 3. Object Update Processing.

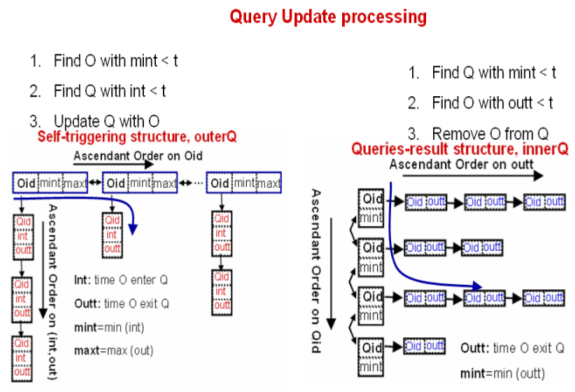

Figure 4. Query Update Processing.

$q_R$len, uniformly generated in the data space, (b) its VBR is q_V = { -$q_V$len, $q_V$len, -$q_V$len, $q_V$len }, (c) its query interval is q_T = [0, $q_T$len]. Query cost is measured as the average number of CPU clock ticks

4.1 Comparing Performance at different timestamps Performance of Continuous Moving Range Queries

In these experiments, we examined 200 continuous moving range queries with properties: spatial extents of the queries $q_R$len = {100,100}; velocities of the queries $q_V$len = {-10, 10, -10, 10} that means our queries run about 12km/h on a direction. However, because we let the queries run on all directions, the speed of their enlargement was about 24km/h. the positions of those queries were randomly located on the terrain. The horizon time to re-set up our data structures periodically was 10. We made two experiments: one was running the continuous queries without interruption of updates, in other words, we assumed the database server was running off line from moving objects, see Figure 5; the other was the running continuous queries but at every timestamp, they were interrupted by 1000 updates from moving objects. In other words, we assumed the database server was running on line with moving objects, see Figure 6. Figure 5 shows performance of queries running from timestamp 7 to timestamp 8 continuously. At timestamp 7, the performance of conQ (our method's continuous queries) is lower than that of TPR-Q (TPR-tree's continuous queries) because we get total cost of setting up our structure data and querying data. Nevertheless, at the other timestamps, conQ's performance is much higher than TPR-Q's because the TPR-tree has to re-evaluate all queries at every timestamp. Therefore, if we compare total cost of running continuous queries during 10 timestamps, conQ will spends about 2 seconds; meanwhile TPR-tree will spends about 10 seconds.

Figure 6 also shows the performance of queries running from timestamp 7 to timestamp 8 continuously. However, in this experiment, the continuous queries have to be updated on not only by objects that are linearly moving but also by objects that have changed their velocities, directions or speeds. Because TPR-Q's continuous queries have to wait until the database server finishes processing 1000 updates, the total cost to re-gain results of TPR-Q is very high. Figure 6 shows that even at timestamp 7, the performance of TPR-Q is worse than that of conQ. From timestamp 9 to 10, the cost of TPR-Q performance is higher than 3 seconds that means the results of those queries are really obsolete for the users of those queries at time they receive.

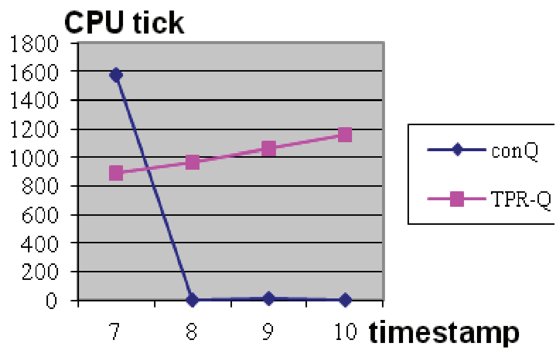

Figure 5. Without object updates, work off line.

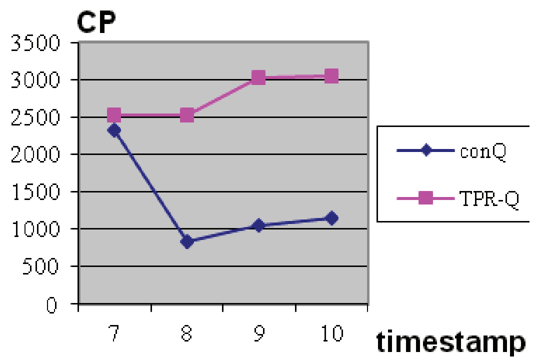

Figure 6. 1000 updates at every t, work online.

4.2 Performance of Continuous Stationary Range Queries

In these experiments, we examined 200 continuous stationary range queries with properties: spatial extents of the queries qRlen = {100,100}. Those range queries would be stationary just like crossroads. The positions of those queries were randomly located on the terrain. The horizon time to re-set up our data structures periodically was 10. We also made two experiments: one was running the continuous queries without interruption of updates, in other words, we assumed the database server was running off line from moving objects, see Figure 7; the other was running continuous queries but at every timestamp, they were interrupted by 1000 updates from moving objects. In other words, we assumed the database server was running on line with moving objects, see Figure 8. Figure 7 shows performance of queries running from timestamp 7 to timestamp 8 continuously. At timestamp 7, the performance of conQ (our method's continuous queries) is lower than that of TPR-Q (TPR-tree's continuous queries). However, in this experiment, the cost of conQ is not substantially higher than that of TPR-Q. At the other timestamps, conQ's performance is much higher than TPR-Q's. In spite of that, total cost of conQ in running continuous queries during 10 timestamps is much cheaper than that of TPR-Q.

Figure 8 also shows the performance of queries running from timestamp 7 to timestamp 8 continuously. However, in this experiment, the continuous queries have to be updated on not only by objects that are lin-

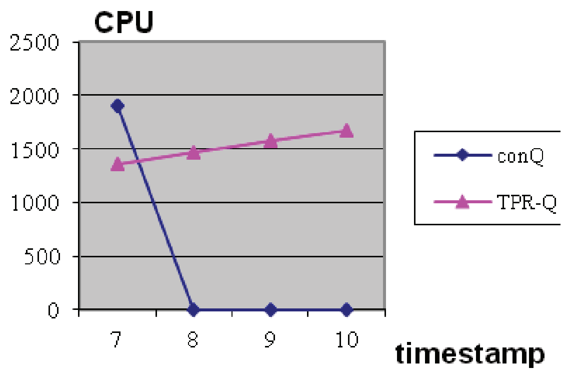

Figure 7. Without object updates, work off line.

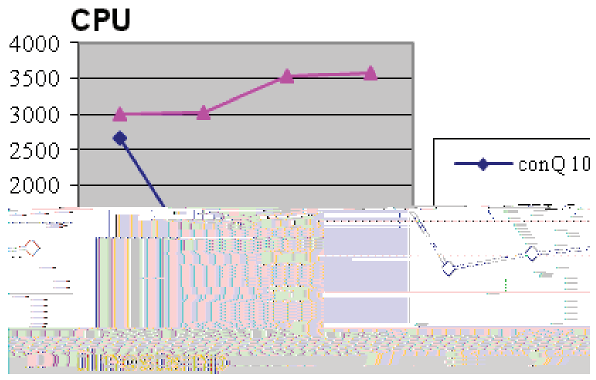

Figure 7. 1000 updates at every t, work online.

early moving but also by objects that have changed their velocities, directions or speeds. Because TPR-Q's continuous queries have to wait until the database server finishes processing 1000 updates, the total cost to re-gain results of TPR-Q is very high. Figure 8 shows that even at timestamp 7, the performance of TPR-Q is worse than that of conQ. Especially, in this experiment, at all timestamps 7, 8, 9, and 10, the cost of TPR-Q performance is always higher than 3 seconds that means the results of those queries make no sense for the users of those queries at time they receive at all.

5. Conclusions

By building up main memory structures to manage continuous queries and taking advantage of data indexes like the HTPR-tree to manage object movements, our proposed method has some advantages as follows

- Low Communication Cost: our data is of motion functions so that a small number of updates occurs over time. Without updates, our method still gains new results of continuous queries offline from moving objects. Moreover, if a query has some changes, only changes need transmitting to its users.
- Low Query cost: continuous queries don't need to re-access the based index on disk to gain new results every time. By using two main memory structures, updating queries is very fast.
- Dynamically Operating: Different from the most of others, our method spends low cost for new coming queries and objects. It supports variety of continuous queries.

In our knowledge, this is the first method managing continuous queries on motion functions data in main memory. We hope this method to be useful for real applications such as traffic systems. Nevertheless, in this method, storing potential objects can make us rather difficult to estimate main memory occupation, although we think that the occupation is almost less than that of Q-index, which stores all safe regions of all objects.

6. Acknowledgment

This paper is supported by project VAST01.09/14-15, Vietnam Academy of Science and Technology (VAST), Hanoi, Vietnam.

7. References

1. Zhang J, Zhu M, Papadias D, Tao Y, Lee DL. Address based Spatial Queries. Proceedings of the ACM International Conference on Management of Data, SIGMOD, San Diego, CA. 2003 Jun. p. 443–54.
2. Zheng B, Lee DL. Semantic Caching in Location-Dependent Query Processing. Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD, Redondo Beach, CA. 2001 Jul; 97–116.
3. Song Z, Roussopoulos N. K-Nearest Neighbor Search for Moving Query Point. Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD, Redondo Beach, CA. 2001 Jul; 79–96.
4. Lazaridis I, Porkaew K, Mehrotra S. Dynamic Queries over Mobile Objects. Proceedings of the International Conference on Extending Database Technology, EDBT, Prague, Czech Republic. 2002 Mar. p. 269–86.

5. Saltenis S, Jensen CS, Leutenegger ST, Lopez MA. Indexing the Positions of Continuously Moving Objects. Proceedings of SIGMOD, NY, USA. 2000 Jun; 331–42.
6. Cai Y, Hua KA, Cao G, Xu T. Real-time processing of range-monitoring queries in heterogeneous mobile databases. Journal of IEEE Transactions on Mobile Computing. 2006 Jul; 931–42.
7. Jung H, Kim Y-S, Chung YD. SPQI: An Efficient Index for Continuous Range Queries in Mobile Environments. Journal of Information Science and Engineering. 2013 May; 557–68.
8. Prabhakar S, Xia Y, Kalashnikov DV, Aref WG, Hambrusch SE. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. Journal of IEEE Transaction on Computers. 2002 Oct; 1124–40.
9. Schmiegelt P, Seeger B, Behrend A, Koch W. Continuous queries on trajectories of moving objects. Proceedings of the 16th International Database Engineering and Applications, NY, USA. 2012; 165–74.
10. Tung HDT, Jung YJ, Lee EJ, Ryu KH. Moving Point Indexing for Future Location Query. Proceedings of the International Workshop on Conceptual Modeling for GIS, 23th ER2004 conference, Shanghai, China. 2004 Nov; 79–90.
11. Datasets at Available from: Http://www.census.gov/geo/www/tiger/. Date accessed: 01/12/2015
12. Li X, Karras P, Shi L, Tan K-L, Jensen CS. Cooperative Scalable Moving Continuous Query Processing.Proceedings of IEEE 13th International Conference on Mobile Data Management, Bengaluru, India. 2012 Jul. p. 69–78.