Optimizing Moving Queries over Moving Object Data Streams

Dan Lin[1], Bin Cui[2,⋆], and Dongqing Yang[2]

[1] National University of Singapore
lindan@comp.nus.edu.sg
[2] Peking University, China
{bin.cui,dqyang}@pku.edu.cn

Abstract. With the increasing in demand on location-based aware services and RFIDs, efficient processing of continuous queries over moving object streams becomes important. In this paper, we propose an efficient in-memory processing of continuous queries on the moving object streams. We model moving objects using function of time and use it in the prediction of usefulness of objects with respect to the continuous queries. To effectively utilize the limited memory, we derive several replacement policies to discard objects that are of no potential interest to the queries and design efficient algorithms with light data structures. Experimental studies are conducted and the results show that our proposed method is both memory and query efficient.

1 Introduction

With rapid advances in electronic miniaturization, wireless communication and position technologies, moving objects that acquire and transmit data are increasing rapidly. This fuels the demand for the location-based services and also deployment of Radio Frequency Identification (RFID) in tracking and inventory management applications. In inventory tracking like applications, disclosure of object positions forms spatio-temporal data streams with high arrival rate, and queries act upon them tend to be continuous and moving. Consequently, queries must be continuously updated and any delay of query response may result in an obsolete answer [7]. Moving object data stream management systems [3,8] have been designed to handle massive numbers of location-aware moving objects. Such systems receive their input as streams of location updates from the moving objects. These streams are characterized by their high input rate, and they cannot be stored and need to be processed on the fly to answer queries. Clearly, the disk-based structures are not able to support the fast updates and provide quick response time. The PLACE [8] extended data streaming management systems to support location-aware environments. However, the PLACE can only manage the snapshots of objects and queries at each timestamp, which inevitably increases the amount of data information. Additionally, it stores the entire dataset in the server for query processing, which may not be applicable for data stream management system. On the other hand, studies of real positional information obtained from GPS receivers installed in

⋆ Contact author. This work is supported by the NSFC under grant No. 60603045.

R. Kotagiri et al. (Eds.): DASFAA 2007, LNCS 4443, pp. 563–575, 2007.
© Springer-Verlag Berlin Heidelberg 2007

cars show that representing positions as linear functions of time reduces the numbers of updates needed to maintain a reasonable precision by as much as a factor of three in comparison to using constant functions [2]. Linear functions are thus much better than constant functions in the data streaming environment.

As with other data streams, processing of continuous spatio-temporal queries over the moving object stream requires the support of in-memory processing. Existing disk-based algorithms cannot be easily turned into in-memory methods, because the underlying structures tend to be bulky and index all data points due to the availability of cheap storage space. Existing tree-based indexing structures [4,10,11,14] for moving objects focus on reducing disk accesses since the execution time is dominated by the I/O operations. In fact, for some indexes, fast retrieval is achieved by preprocessing and optimization before insertion into the index [14]. In this paper, we propose an efficient approach which is able to handle moving objects and queries represented by functions. Due to the limited amount of memory, we design light data structures, based on hash tables and bitmaps. To manage the limited amount of buffer space, we design several replacement policies to discard objects that are of no potential interest to the queries. Experimental results demonstrate that our algorithms can achieve fast response time and high accuracy with a small memory requirement.

The rest of the papers is organized as follows. Section 2 defines the problem and reviews the related work. Section 3 introduces the overall mechanism. Section 4 presents the algorithms of continuous range queries. In section 5, we report the experimental results. Finally, Section 6 concludes with the paper.

2 Problem Statement and Related Work

2.1 Problem Statement

The moving object data stream is made up of a sequence of update information of moving objects. We assume that moving objects are capable of repeatedly transmitting their positions and velocities to a central server. Then, each tuple in the stream includes $\langle OID, \overline{Op}, \overline{Ov}, Ot\rangle$, where OID is the object ID, Ot is the update time, $\overline{Op}$ and $\overline{Ov}$ are the position and velocity at time Ot respectively. The incoming tuple is the update information of the existing tuple with the same OID. To reduce the update frequency, a linear function is used to model the trajectory of a moving object. A moving object is required to transmit a new location to the server when the deviation between its real location and its server-side location exceeds a threshold, dictated by the services to be supported. In keeping with this, we define the *maximum update time* (U) as a problem parameter. This quantity denotes the maximum time duration in-between two updates of the position of any moving object.

The query data stream is comprised of a sequence of two types of queries: continuous static range query and continuous moving range query. They are defined as follows:

- *Continuous static range query*: Given a static range R at time Qt, the query needs continuously reporting all the moving objects within the range R from time Qt.
- *Continuous moving range queries*: Given a range R moving at the velocity $\overline{Qv}$, and a time Qt, the query needs continuously reporting all the moving objects inside $R(t)$ from time Qt, where $R(t)$ denotes the range at time t after Qt.

In fact, the static query can be treated as the special case of the moving query where the velocity is equal to zero. Therefore, each tuple in the query data stream can be represented using the same format $\langle QID, R, \overline{Qv}, Qt \rangle$. Similar to the moving object data stream, the newly incoming query will replace the tuples with the same QID in the memory. Without loss of generalization, we consider the square range throughout the paper.

2.2 Related Work

There are numerous work in the area of spatio-temporal query processing on moving objects (e.g.,[1,4,6,12,10,11,14,16]). However, these disk-based approaches may not be suitable for scalable, real-time location based queries because of high I/O costs, even when sophisticate buffer management is employed. Although it is possible to tailor these methods and put the entire data and indexes into the main memory to speed up, this may consume too much memory. In this paper, we have proposed a main memory based index approach. Our approach does not intend to store the data of all moving objects, because only some objects will be included in the queries answers.

In [5,15], continuous range queries are made over moving objects. The queries being considered are however static. In [7,8,9], the problem of moving queries over moving objects are discussed. However, their approaches are to store snapshots of queries and objects at each timestamp making it necessary to store and process these snapshots on the disk. To ensure efficient processing, our work here try to address the same problem using only in-memory processing.

3 Continuous Query Processing on Moving Object Streams

3.1 System Architecture

In this section, we introduce the QMOS (Query Moving Object Stream) system. Figure 1 gives an overview of the QMOS system. There are four in-memory storages (shaded boxes): object pool, query pool, event queue and query filter; and two processors (white boxes): query processor and discarding processor.

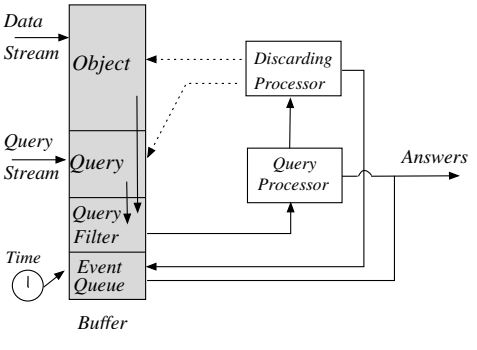


Fig. 1. An Overview of the System Architecture

An incoming object is first put into the *object pool*, and then will be sent to the *query processor* together with selected queries from the *query filter*. The query processor will generate three kinds of results: current answer, potential answer, and none answer. *Current answer* means that the incoming object is one of the answers of the query at the current time, which will be directly reported to the user. *Potential answer* means that the incoming object will be one of the answers of the query at some future time (within the maximum update time interval U). *None answer* means that the incoming object is neither a current answer nor a potential answer. Both of them will be further sent to the *discarding processor*. Since the memory is limited, potential answers and none answers need to be judged whether they are valuable to be stored. The discarding processors will provide a feedback to the object pool if the incoming object can be discarded. Valuable potential answers will then be stored as events in the *event queue*. As time passes, potential answers may turn into current answers and be reported to the users. In addition, the event queue also handles objects which leave the query answer sets. It is worth noting that the query answer set is maintained incrementally. There is an output only when the query result has been changed, due to adding or deleting an object from the answer set.

The processing of an incoming query is relatively simple. If the memory is enough, we store it in the *query pool*, and register its summary information in the *query filter*. Otherwise, the *Discarding processor* is triggered to find out whether there is some space can be used for the new query.

3.2 Storage Components

The *object pool* stores the information of the moving objects. Each tuple in the object pool is in the form of $\langle OID, \overline{Op}, \overline{Ov}, Ot, PA, Ca, Evt \rangle$, where OID, $\overline{Op}$, $\overline{Ov}$, Ot are used to represent the object, PA is the number of queries of which the object is a potential answer, Ca is a bit-map storing the entries to the queries of which the object is a current answer, and Evt is also a bit-map used to locate the related events of this object.

The *query pool* stores the information of the queries. Each tuple in the *query pool* consists of $\langle QID, R, Qt, Qa, Evt \rangle$, where QID is the query ID, R is the query range, Qt is the query starting time, Qa is a pointer to the query results, and Evt is used to locate the related events of the query (the same as the corresponding part in the object pool). Further, R is represented by $(\overline{Qp}, \overline{Qv}, L)$, where $\overline{Qp}$ stores the left bottom corner of the query window, $\overline{Qv}$ is the moving velocity of the query window, and L is the length of the query window.

The *event queue* stores future events when an object will join or leave current query answer set. Each tuple consists of four components: t, pO, pQ, and M. t is the time that the event may happen. pO is a pointer to the object stored in the object pool, and pQ refers to the query that this object may affect. M is a one-bit mark: if in the event the object will become one of the query answers, M is set to 1; if the object will no longer be the answer, M is set to 0.

Object pool, query pool and event queue are all organized as hash tables where the keys are OID, QID and t respectively. The lengths of the hash tables are determined by the memory size. The hash structure is preferred over other kinds of data structures

since (i) these data are usually retrieved by their key values and hashing techniques provide fast and direct access; (ii) the memory is limited and hash structures have less storage overhead.

The *query filter* is designed to accelerate the query processing. It is a grid structure which captures the current and future positions of moving queries. Basically, we partition the space into a regular grid where each cell is a bucket. Each bucket contains pointers to the queries passing this bucket.

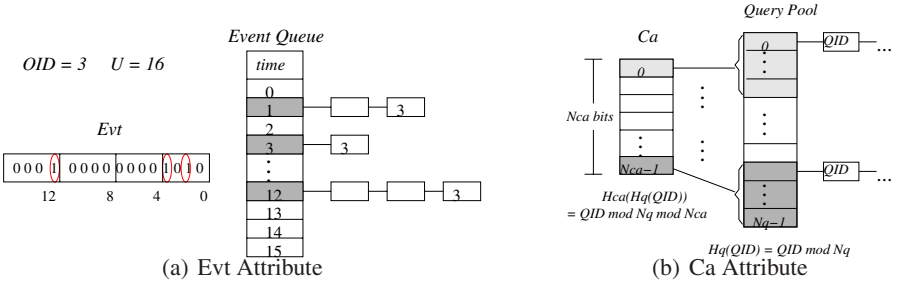
3.3 Data Processing

We proceed to present how the system manages the two kinds of incoming data (moving objects and queries) and the internal data – events. During all processes, whenever there is not enough memory, discarding policies are applied to remove useless data to collect memory. We defer the discussion of the discarding policies to the next subsection.

- **Moving Object Data Streams**

An incoming object O is processed as follows. First, we check whether the object ID has already existed in the object pool. If yes, we modify the corresponding tuple by using the new information including position $\overline{Op}$, velocities $\overline{Ov}$, update time Ot. Information with regards to this object O in the query results and the event queue is removed, since they become obsolete now. Then object O will be computed with queries selected by the query filter. We need to decide whether we store this object in the memory. We will store it only if it proves to be useful. In particular, the coming object is useful if it is a current/ potential answer of a query, and there is enough memory after applying discarding policies. After the object O is successfully stored in the memory, the pointer to the object will be inserted to the query answer set where it is a current answer, and the leaving events and potential answers will be added into the event queue.

The details of query and discarding processes will be addressed later. The deletion and insertion of the object in the object pool is done fast by hashing the OID, similarly to the insertion of the related events and query answers. While the deletion of related events and query answers is a little more complex. The straightforward way is to scan the whole event queue (query pool) to find the events (queries) related to the object, which is obviously inefficient and may result in an unbearable delay when the memory size is large. To avoid such a brute force method, we propose the following techniques.



(a) Evt Attribute (b) Ca Attribute

Fig. 2. Examples of Evt and Ca Attributes

The search of the related events is managed with the aid of the Evt attribute of the object. Specifically, one bit in the Evt is related with one timestamp in the event queue, and the bit will be set to 1 if there is an event with respect to the object happening at the corresponding timestamp. For example (see Figure 2(a)), assuming that $OID = 3$ and $U = 16$, the object has related events at time 1, 3 and 12. Then the 1st, 3rd and 12th bits in the Evt are set to 1, others are 0. By checking the Evt, we can easily find the entries to the related events of the object and avoid scanning the entire event queue.

The search of the related queries is accelerated by the Ca attribute of the object. Different from the Evt, the one-one map (i.e. one bit to one entry in the hash table of the query pool) may lead to a long Ca, because the number of queries in memory could be large when the memory scales up (i.e. the length of the hash table of the query pool may grow up). Therefore, we employ a second level hashing over the query IDs, where each bit of Ca corresponds to a series of entries in the hash table of the query pool. As shown in Figure 2, suppose that the length of the hash table of the query pool is N_q, and the number of the bits in Ca is N_{ca}. Queries are first hashed to the hash table of the query pool by the function $H_q(QID) = QID \ mod \ N_q$. Then the mapping function for the Ca is $H_{ca}(QID) = H_{ca}(H_q(QID)) = (QID \ mod \ N_q) \ mod \ N_{ca}$.

- **Queries**

For an incoming query Q, we insert $\langle QID, \overline{Qp}, \overline{Qv}, L, Qt, NULL \rangle$ into the query table to represent the new query. The trajectory of the new query will be registered in the query filter. The new query only considers the objects coming after it, which means it needs some time to "warm up". The "warming-up" time could be short since objects are updated frequently. If the query expires, we remove the entry from the query pool, and the events related to the old query (the procedure is similar to that in the previous section). Note that objects become none answers after the deletion of the query are automatically discarded from the memory.

- **Events**

As time passes, the event queue is checked to update current answers of queries. All events whose start time is less than or equal to current time are evaluated. Recall that, the events are stored in a hash table with the length equal to the maximum update interval U. By hashing the current timestamp t, we can find its entry in the hash table.

There are two kinds of events: objects leaving or entering the query range. According to the type of an event, different actions are performed. Given an event $\langle t, pO, pQ, M \rangle$, if the mark M equals to 1, the object pO pointing to should be inserted into the answer list of the corresponding query that pQ points to. If M equals to 0, which means the object O is no longer an answer of the query Q, then O is removed from the answer list of Q. In both situations, the Ca and Evt attributes of O should be adjusted. Finally, we delete the event itself.

3.4 Discarding Policy

Continuous queries over infinite streams may require infinite working memory. Thus, an essential solution to answer such queries in bounded memory is to discard some unimportant data when the memory is full. Our proposed discarding policies comply with the basic rule that discarding data of lowest priority first. In our scenario, we define

the priorities of the data as that: the query data is most important, followed by the current answer and the potential answer.

Each time the memory is full, we first attempt to discard objects which are neither current answers nor potential answers. If this operation fails, we further apply any of the following three policies.

1. Discard the oldest object according to its insertion time. The idea behind the Policy 1 is that the oldest object has the highest probability to be updated first, and thus the influence of discarding this object may be ended within the shortest time.
2. Discard the object whose first appearance in the event queue is latest than that of any other object and it is an entering event. The motivation of Policy 2 is to keep the query answers unaffected as long as possible. Therefore, it picks the object which is the last one to become a potential answer. Combined with the idea of Policy 1, we may have a variation of Policy 2: discard the object which has the longest time interval between its insertion time and the time it becomes an answer.
3. Discard the object that affects fewest queries. Different from the first two policies that both take into account the time effect, Policy 3 aims to minimize the number of queries that the object affects.

All the policies share the same purpose that minimizes the error rate of the query answers after the discarding. Note that the query data will be discarded only when the memory is fully occupied with queries.

Next, we introduce the discarding process. Any policy is realized by scanning the object pool once. Policy 1 compares the insertion time Ot of each object and discards the one with smallest Ot. Policy 2 is done by examining the attribute Evt of an object, where the lowest none-zero bit refers to the first event of the object. We then need to check whether the event is an entering event or a leaving event. For the Policy 3, the number of related queries can be approximated by the sum of none-zero bits in Ca and Evt. If the exact number is required, we can further access corresponding tuples in the event queue and query pool according to Ca and Evt.

4 Algorithms of Continuous Range Queries

4.1 Processing a Single Query

In the two-dimensional space, given a continuous range query $\langle QID, \overline{Qp}, \overline{Qv}, L, Qt\rangle$, the query range at time t ($Qt \leq t$) can be represented by the left-bottom and right-top corner, $[(Qp_x + Qv_x(t - Qt), Qp_y + Qv_y(t - Qt)), (Qp_x + Qv_x(t - Qt) + L, Qp_y + Qv_y(t_Q t) + L)]$. For an incoming object $\langle OID, \overline{Op}, \overline{Ov}, Ot\rangle$, we need to identify whether it is a current answer or a potential answer.

An object is a current answer to the range query if its position at current time t_c is inside the query range at time t_c. We first compute the query range at t_c by its moving function, and then compare the position of the object with the left-bottom and right-top corner of the query range directly.

$$\begin{cases} Qp_x + Qv_x(t_c - Qt) \leq Op_x \leq Qp_x + Qv_x(t_c - Qt) + L \\ Qp_y + Qv_y(t_c - Qt) \leq Op_y \leq Qp_y + Qv_y(t_c - Qt) + L \end{cases}$$

If the above conditions are satisfied, the object is a current answer to the range query and will be added into the answer list. The remaining task is to compute the time it leaves the query range, and insert the leaving event to the event queue. As the object is already inside the query range, its future trajectory will have only one intersection point with the query range, and the intersection time is the leaving time. The details of the computation will be explained shortly.

An object is a potential answer to the range query if its position at future time t_f (not later than the maximum update interval) is inside the query range at time t_f. Then we need to compute the time when the object enters the query range, and insert this future event to the event queue. As the object is currently outside the query range, its future trajectory may have at most two intersection points with the query range. The earlier intersection time is the entering time and the other one is the leaving time.

We proceed to present how to compute the intersection time. Figure 3 shows a continuous range query and an incoming moving object, where the solid rectangle presents the query range at the current time, the rectangles with broken line denotes the query ranges at near future, the black point is the moving object, and the connecting line with arrow shows the object's future trajectory. To check whether the object's future trajectory intersects with the query range, let us consider the four borders of the query range, AB,BC,CD,DA, one by one. The border AB moves at the speed of Qv_x, and thus the line at time t (denoted as L_{ab}) it resides in can be described by the equation: $x = Qp_x + Qv_x(t - Qt)$. If the object's trajectory intersects with AB, it must also intersects with L_{ab}. In other words, the object's x coordinate should be on L_{ab} at the intersection time. Assuming that the intersection time is t_{ab}, we have the equation: $Op_x + Ov_x(t_{ab} - Ot) = Qp_x + Qv_x(t_{ab} - Qt)$. By solving the equation, we obtain the following results:

$$t_{ab} = \begin{cases} \frac{(Qp_x - Ox) - (Qv_x \cdot Qt - Ov_x \cdot Ot)}{Ov_x - Qv_x}, & Ov_x \neq Qv_x; \\ +\infty, & Ov_x = Qv_x. \end{cases}$$

Note that, when $Ov_x = Qv_x$, i.e. the object and the border AB move at the same speed and same direction, they will never meet each other. Therefore, the t_{ab} is set to be the infinite large $+\infty$ in this case.

The resultant t_{ab} value is invalid if it does not satisfy the constraints: (i) $t_{ab} > Ot$, i.e. the intersection time should be later than the object insertion time; (ii) $t_{ab} > Qt$,

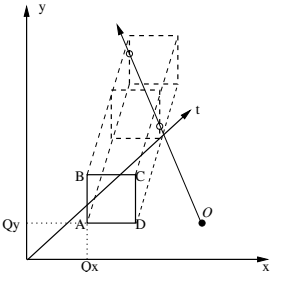


Fig. 3. An Example of a Continuous Range Query

i.e., the intersection time should be later than the query starting time; (iii) $t_{ab} < Ot+U$, i.e., the intersection time should not exceed the validity period of the object. Invalid t_{ab} will also be reset to the infinite large $+\infty$.

So far the t_{ab} we computed is only the intersection time of the object's trajectory and the line that the AB belongs to. We need to further check whether the intersection point lies in the line segment AB. Suppose that the t_{ab} is valid, we can use it to compute the intersection point $P(P_x, P_y)$, where $P_x = Op_x + Ov_x(t_{ab} - Ot)$, and $P_y = Op_y + Ov_y(t_{ab} - Ot)$. Then we compare the y coordinate of P and points A, B. If $A_y \leq P_y \leq B_y$, the intersection point is in the segment AB, which means we obtain one useful intersection time. Otherwise, we again set the t_{ab} to be $+\infty$.

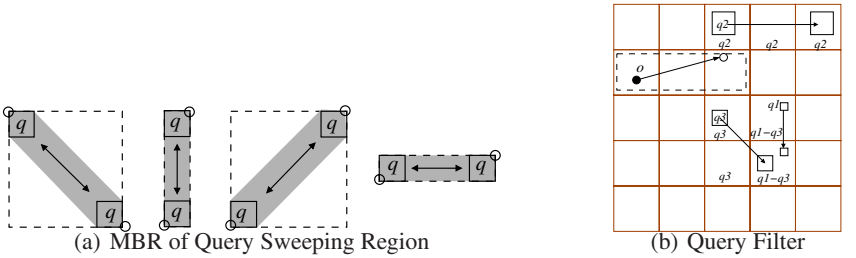
The similar computation is carried out for the other three borders BC, CD and DA. The entering time t_e is the minimum value of the four intersection times, and the leaving time t_l is the finite maximum value of the four intersection times. Note that we may not need to compute the four intersection times. If we have obtained two intersection times which are not $+\infty$, we do not need to process the remaining borders.

4.2 Processing Multiple Queries

An incoming object could be a current or potential answer of multiple queries in the memory. Comparing it with all the queries one by one may result in high query cost. Therefore, we propose a query filter to prune the searching space.

The query filter is a regular grid structure which partition the space into equal cells. Each cell stores pointers to queries which pass by the cell during the maximum update interval U. The pointer to one query may be stored several times in the grid due to the movement of the query. To reduce the number of duplications, we need to decide a reasonable grid cell size. For example, we can set the extent of cell to be slightly larger than $v_{max} \cdot U$, where the v_{max} is the maximum speed of a query. For a query with a long lifetime, we will update its information in the grid every U time interval. Moreover, in order to speed up the mapping process, we do not compute the exact cells that the queries intersects with. Instead, we map the minimum bounding rectangle of the query sweeping region (during U) to the grid as shown in Figure 4(a).

We are now ready to look at how the query filter works. For example, in Figure 4(b), given an incoming object O at time Ot, we will map it to the grid in the similar way as we have done to the query. First we compute its position at time $Ot + U$. The search



(a) MBR of Query Sweeping Region (b) Query Filter

Fig. 4. Query Filter Construction

space (the dashed rectangle in the figure) is the rectangle determined by the two positions at Ot and $Ot + U$ respectively. Then only queries registered inside this rectangle need to be computed.

5 Performance Study

All the experiments are conducted on a 2.6G Hz P4 machine with 1Gbyte of main memory. The memory for our application is limited from 100K to 2M. Moving objects are represented as points in the space domain of 1000×1000. The datasets were generated by a typical data generator [11]. The maximum interval between two successive updates of an object is equal to 30 time units. Queries follow the same distribution of the moving objects. The moving speed of the queries is half of the speed of objects. The query window size is 0.01% of the space. The number of queries existing at the same time varies from 100 to 500. Unless noted otherwise, we use 300K memory for 100K moving objects when there are 100 queries at each timestamp.

We evaluate the memory requirement, the accuracy and the response time of the proposed three policies. The memory requirement is compared with the B^x-tree. The accuracy function is $Accuracy = \frac{Number\ of\ answers\ produced\ by\ the\ algorithm}{Number\ of\ correct\ answers}$. The response time is defined as the time interval between the input of a data and the output of the result regarding to this data.

• **Effect of Memory Size.** The first round of experiments evaluate efficiency of the three discarding policies when varying the total available memory size from 100K bytes to 500K bytes. The number of moving objects are 100K, and the data streams of their update information is of size 217K tuples during 30 timestamps.

Figure 5(a) shows the results of the accuracy at timestamp 30. As shown, the performances of all the policies improve with the increasingly large memory size. The reason is straight forward: larger memory can hold more answers. When the memory size reaches beyond a certain point ($> 300K$), the accuracy of all the policies approaches 100%. Note that 300K is about only 13% of the space used to store all the objects. This is because our algorithm only catches query answers and the result demonstrates its space efficiency. We can also observe that Policy 2 always yields higher accuracy than the other two policies. The reason could be that Policy 2 maximizes the valid period of query results.

Figure 5(b) shows the average response time of the three policies during one maximum update interval. We can see that as the memory size increases, the response time of three policies first increases slightly and then almost keep constant. For an object, the response time is the sum of query processing time and the discarding processing time. The query processing time will not be affected by the memory size when the query number is fixed, and thus the variation of the response time is mainly due to the variation of the discarding processing time. As the memory increases, the time to find a replacement slows down, whereas the need to execute a discarding policy is reduced. When these two factors reach a balancing station, the performance becomes stable. In addition, the resultant three curves are close to one another possibly because that the discarding process is only different in the selection metric, and hence the processing time is similar.

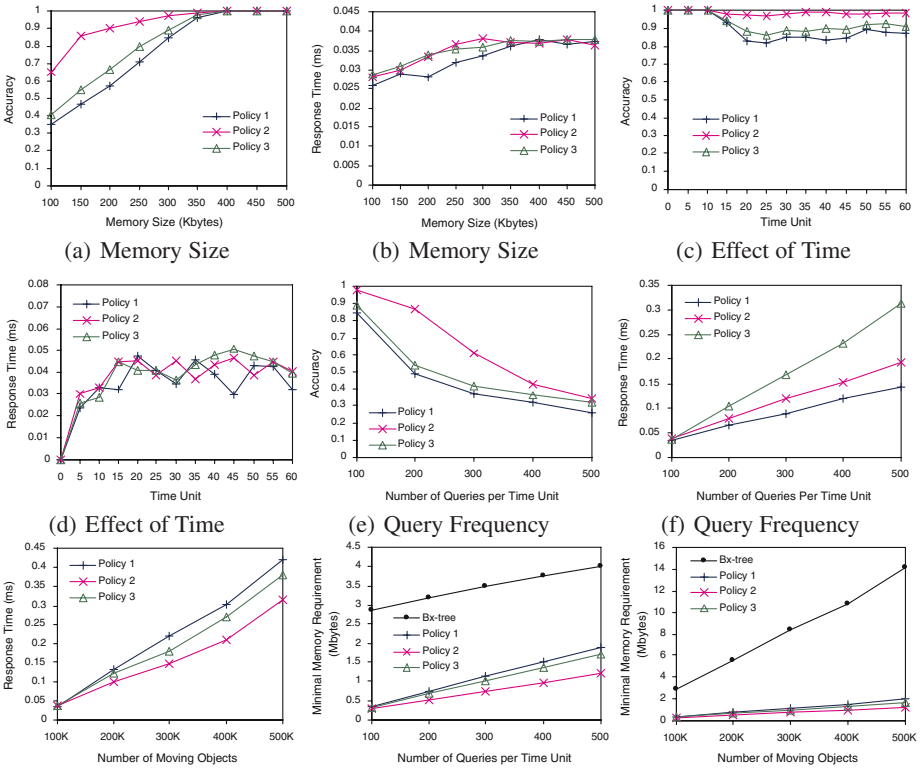


Fig. 5. Experimental Results

• **Effect of Time.** Next, we investigate performance degradation across time. The 100K moving objects are kept updated during 60 timestamps (two times of maximum update interval). As shown in Figure 5(c) and (d), the performance of all the policies decreases a little as time passes, which is due to the execution of the discarding policies, and the larger query range.

• **Effect of Number of Queries Per Time Unit.** In this set of experiments, we vary the number of queries at the same timestamp. We fix the memory size to 300K and test the accuracy and response time. From Figure 5(e) and (f), we observe that the performance degenerates with growing number of queries at the same time. The decrease of the accuracy is mainly caused by the increase of the result dataset. Due to the memory limitation, even potential answers at near future time may be discarded, which affects the accuracy and also increases the processing time.

• **Effect of Data Size.** To study the scalability of our algorithms, we examine the method with varying the number moving objects from 100K to 500K. For the response time (see Figure 5(i)), Policy 2 is the best since it requires the smallest memory so that the discarding process can be executed fastest.

• **Comparison with the B^x-tree.** To show the effectiveness of the proposed method, we compare it with the B^x-tree [4] which has much smaller space requirement compared with other existing index structures, e.g. the TPR*-tree [14]. We first explore the minimum memory required for each policy to achieve high accuracy (above 99%) by varying the numbers of queries per time unit. Not that, the B^x-tree stores all the objects for queries. Figure 5(h) shows the results. Not surprisingly, the minimum memory required for all policies increases with the number of queries. However, our algorithms can save up to 90% space compared with the B^x-tree. Among three policies, Policy 2 has the smallest space requirement, followed by Policy 3 and 1. This is consistent with the previous results in Figure 5(a). Those policies perform better when using the same size of memory, will need less space to reach high accuracy.

Figure 5(i) shows that our algorithms scale very well compared with the B^x-tree for large data sizes. By using our algorithm, less than 2M bytes memory is required for the 500K data, whereas the B^x-tree needs about 15M bytes space.

6 Conclusion

In this paper, we proposed a novel scheme which can handle infinite data streams in memory, and provide prompt response, by compromising with small errors. Our approach supports continuously moving queries over moving objects, both of which are represented by linear functions. Due to the constraints of the memory size and response time, we propose light data structures, and employ hashing techniques. Also, we derive several replacement policies to discard objects that are of no potential interest to the queries. Experimental studies were conducted and the results show that our proposed method is both memory and query efficient.

References

1. Y. Chen, F. Rao, X. Yu, D. Liu, and L. Zhang. Managing Location Stream Using Moving Object Database. *Proc. DEXA*, pp. 916–920, 2003.
2. A. Čivilis, C. S. Jensen, J. Nenortaitė, and S. Pakalnis. Efficient Tracking of Moving Objects with Precision Guarantees. *Proc. Mobiquitous*, pp. 164–173, 2004.
3. H. G. Elmongui, M. Ouzzani and W. G. aref. Challenges in Spatio-temporal Stream Query Optimization. *Proc. MobiDE*, pp. 27–34, 2006.
4. C. S. Jensen, D. Lin and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. *Proc. VLDB*, pp. 768–779, 2004.
5. D. V. Kalashnikov, S. Prabhakar, W. G. Aref, and S. E. Hambrusch. Efficient Evaluation of Continuous Range Queries on Moving Objects. *Proc. DEXA*, pp. 731–740, 2002.
6. Y. Li, J. Yang, and J. Han. Continuous K-Nearest Neighbor Search for Moving Objects. *Proc. SSDBM*, pp. 123–126, 2004.
7. M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. *Proc. SIGMOD*, pp. 623–634, 2004.
8. M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous Query Processing of Spatio-temporall Data Streams in PLACE. *Proc. STDBM*, pp. 57–64, 2004.
9. R. V. Nehme, and E. A. Rundensteiner. SCUBA: Scalable Cluster-Based Algorithm for Evaluating Continuous Spatio-temporal Queries on Moving Objects. *Proc. EDBT*, pp. 1001–1019, 2006.

10. J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An Efficient Index for Predicted Trajectories. *Proc. SIGMOD*, pp. 637–646, 2004.
11. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proc. SIGMOD*, pp. 331–342, 2000.
12. D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch. Main Memory Evaluation of Monitoring Queries Over Moving Objects. *Distributed and Parallel Databases*,pp. 117–135, 2004.
13. Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. *Proc. VLDB*,pp. 287–298, 2002.
14. Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *Proc. VLDB*, pp. 790–801, 2003.
15. K. L. Wu, S. K. Chen, and P. S. Yu. Indexing continual Range Queries with Covering Tiles for Fast Locating of Moving Objects. *Proc. ICDCSW*,pp. 470–475, 2004.
16. M. Yiu, Y. Tao, and N. Mamoulis. The B^{dual}-Tree: Indexing Moving Objects by Space-Filling Curves in the Dual Space. *To appear in VLDB Journal*, 2006.