SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases

Mohamed F. Mokbel Xiaopeng Xiong Walid G. Aref
Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398
{mokbel,xxiong,aref}@cs.purdue.edu

ABSTRACT

This paper introduces the *Scalable INcremental hash-based Algorithm* (SINA, for short); a new algorithm for evaluating a set of concurrent continuous spatio-temporal queries. SINA is designed with two goals in mind: (1) Scalability in terms of the number of concurrent continuous spatio-temporal queries, and (2) Incremental evaluation of continuous spatio-temporal queries. SINA achieves scalability by employing a shared execution paradigm where the execution of continuous spatio-temporal queries is abstracted as a spatial join between a set of moving objects and a set of moving queries. Incremental evaluation is achieved by computing only the updates of the previously reported answer. We introduce two types of updates, namely *positive* and *negative* updates. Positive or negative updates indicate that a certain object should be added to or removed from the previously reported answer, respectively. SINA manages the computation of positive and negative updates via three phases: the hashing phase, the invalidation phase, and the joining phase. The hashing phase employs an in-memory hash-based join algorithm that results in a set of positive updates. The invalidation phase is triggered every T seconds or when the memory is fully occupied to produce a set of negative updates. Finally, the joining phase is triggered by the end of the invalidation phase to produce a set of both positive and negative updates that result from joining in-memory data with in-disk data. Experimental results show that SINA is scalable and is more efficient than other index-based spatio-temporal algorithms.

1. INTRODUCTION

With the increasing number of computer applications that rely on large spatio-temporal data sets, it becomes essential to provide efficient query processing techniques for spatio-temporal databases. Examples of these applications include location-aware services, traffic monitoring, and enhanced 911 service. Unlike traditional databases, spatio-temporal databases are concerned with objects that continuously change their locations and/or shapes over time. As a consequence of adding the *temporal* dimension, spatio-temporal databases become highly dynamic environments.

Unlike traditional applications, spatio-temporal applications (e.g., location-aware services) have the following distinguishing characteristics: (1) A large number of mobile and stationary objects, and consequently a large number of mobile and stationary queries. (2) 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 of the query or the database objects [27]. (3) Any delay of the query response results in an obsolete answer. For example, 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. These distinguished characteristics call for special spatio-temporal query processing algorithms to achieve scalability and efficient evaluation of continuous spatio-temporal queries.

In this paper, we propose the *Scalable INcremental hash-based Algorithm* (SINA, for short) for continuously evaluating a dynamic set of continuous spatio-temporal queries. SINA exploits two main paradigms: *Shared execution* and *incremental evaluation*. By utilizing the *shared execution* paradigm, continuous spatio-temporal queries are grouped together and joined with the set of moving objects. By utilizing the *incremental evaluation* paradigm, SINA avoids continuous reevaluation of spatio-temporal queries. Instead, SINA updates the query results every T time units by computing and sending only updates of the previously reported answer. We distinguish between two types of query updates: *Positive updates* and *negative updates*. Positive updates indicate that a certain object needs to be added to the result set of a certain query. In contrast, negative updates indicate that a certain object is no longer in the answer set of a certain query. As a result of having the concept of positive and negative updates, we achieve two goals: (1) Fast query evaluation, since we compute only the update (change) of the answer not the whole answer. (2) In a typical spatio-temporal application (e.g., location-aware services and traffic monitoring), query results are sent to customers via satellite servers [11]. Thus, limiting the amount of data sent to the positive and negative updates only rather than the whole query answer saves in network bandwidth.

SINA introduces a general framework that deals with all mutability combinations of objects and queries. Thus, it is applicable to: Stationary queries on moving objects (e.g.,

.

"Continuously report the cars that are within 3 miles of my home"), moving queries on stationary objects (e.g., *"Continuously report all gas stations that within 3 miles of my location"*), and moving queries on moving objects (e.g., *"Continuously report all police cars that within 3 miles of my car location"*). For simplicity, we present SINA in the context of continuous spatio-temporal range queries. However, as will be discussed in Section 5, SINA is applicable to a broad class of continuous spatio-temporal queries (e.g., nearest-neighbor and aggregate queries). In general, the contributions of this paper are summarized as follows:

1. We utilize the *shared execution* paradigm as a means to achieve scalability for continuous spatio-temporal queries (Section 3).

2. We propose SINA; a new algorithm for incrementally evaluating a set of concurrently executing continuous spatio-temporal queries. Incremental evaluation is achieved through computing *positive* and *negative* updates of the previously reported answer (Section 4).

3. We prove the correctness of SINA by proving the following: (a) Completeness, i.e., all query results will be produced by SINA. (b) Uniqueness, i.e., SINA produces duplicate-free results. (c) Progressiveness, i.e., SINA reports only the updates of the previously reported answer (Section 6).

4. We provide experimental evidence that SINA outperforms other R-tree-based algorithms (e.g., Q-index [19] and Frequently Updated R-tree [16]) (Section 7).

The rest of the paper is organized as follows: Section 2 highlights related work for continuous spatio-temporal query processing. In Section 3, we introduce the concept of shared execution for a group of spatio-temporal queries. Section 4 proposes the *Scalable INcremental hash-based Algorithm* (SINA). The extensibility of SINA to a variety of continuous spatio-temporal queries and to handle clients that are disconnected from the server for short periods of times is discussed in Section 5. The correctness proof of SINA is given in Section 6. Section 7 provides an extensive list of experiments to study the performance of SINA. Finally, Section 8 concludes the paper.

2. RELATED WORK

Most of the recent research in spatio-temporal query processing (e.g., [15, 23, 25, 31, 32]) focus on continuously evaluating one spatio-temporal query at a time. Issues of scalability, incremental evaluation, mutability of both objects and queries, and client overhead are examples of challenges that either are overlooked wholly or partially by these approaches. Mainly, three different approaches are investigated: (1) The validity of the results [31, 32]. With each query answer, the server returns a valid time [32] or a valid region [31] 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 [23] (assuming computational and storage capabilities at the client side) or in the server side [15]. Previously cached results are used to prune the search for the new results of k-nearest-neighbor queries [23] and range queries [15]. (3) Precomputing the result [15, 25]. If the trajectory of the query movement is known apriori, then by using computational geometry for stationary objects [25] or velocity information for moving objects [15], we can identify which object will be nearest-neighbors [25] to or within a range [15] from the query trajectory. If the trajectory information is changed, then the query needs to be reevaluated.

There is lot of research in optimizing the execution of multiple queries in traditional databases (e.g., see [22]), continuous web queries (e.g., see [7]), and continuous streaming queries (e.g., see [6]). Optimization techniques for evaluating a set of continuous spatio-temporal queries are recently addressed for centralized [19] and distributed environments [5, 8]. Distributed environments assume that clients have computational and storage capabilities to share query processing with the server. The main idea of [5, 8] is to ship some part of the query processing down to the moving objects, while the server mainly acts as a mediator among moving objects. This assumption is not always realistic. In many cases, clients use cheap, low battery, and passive devices that do not have any computational or storage capabilities. While [5] is limited to stationary range queries, [8] can be applied for both moving and stationary queries. Our work is distinguished from these approaches where we do not assume any storage or computational capabilities at the clients. Instead, clients are required to submit the minimal possible information to the servers, mainly the identifier and location of the moving objects.

Up to the authors' knowledge, the only work that addresses the scalability issue with no client overhead is the Q-index [19]. The main idea of the Q-index is to build an R-tree-like [9] index structure on the queries instead of the objects. Then, at each time interval T, moving objects probe the Q-index to find the queries they belong to. 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.

In general, spatio-temporal queries can be evaluated using a spatio-temporal access method [2]. The TPR-tree [20] and its variants (e.g., the TPR*-tree [26]) are used to index objects with future (predictive) trajectories. However, there are no special mechanisms to support the continuous spatio-temporal queries in any of these access methods. Thus, to emulate the continuity, a client may need to issue the query multiple times at consecutive time intervals for reevaluation. In addition, these spatio-temporal access methods can answer only queries about moving objects but not for stationary objects.

Our proposed *Scalable INcremental hash-based Algorithm* (SINA) distinguishes itself from the above approaches, where we go beyond the idea of reevaluating continuous queries. Instead, we use incremental evaluation to compute only the updates of the previously reported result. In addition, unlike [5, 8], SINA does not assume any computational capabilities on the client side. Moreover, SINA is scalable to support a large number of concurrently outstanding continuous queries and can deal with many variations of continuous spatio-temporal queries. Table 1 gives a comparison of our proposed approach with previous approaches [15, 19, 23, 25] and with the usage of the TPR-tree [20].

	Property	SR [23]	DQ [15]	CNN [25]	Q-index [19]	TPR[20]	SINA
Execution Model	Incremental	×	√	×	×	×	√
	Shared execution	×	×	×	√	×	√
Query types	Moving queries on stationary objects	√	√	√	×	×	√
	Stationary queries on moving objects	×	×	×	√	√	√
	Moving queries on moving objects	×	√	×	×	√	√
Assumptions	No client overhead	×	√	√	√	√	√
	No velocity assumptions	√	×	×	√	×	√

Table 1: Comparison of different algorithms for continuous spatio-temporal queries.

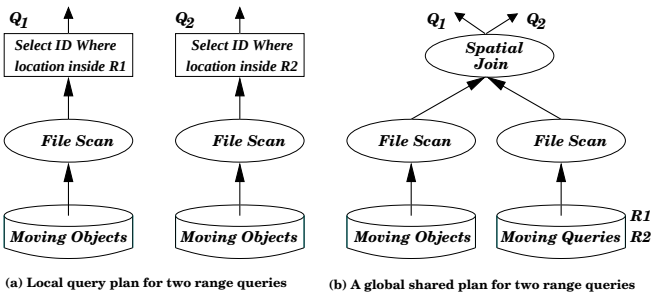


Figure 1: Shared execution of continuous queries.

3. SHARED EXECUTION OF CONTINUOUS SPATIO-TEMPORAL QUERIES

In this section, we exploit the *shared execution* paradigm as a means of achieving scalability for concurrently executing continuous spatio-temporal queries. The main idea is to group similar queries in a query table. Then, the evaluation of a set of continuous spatio-temporal queries is abstracted as a spatial join between the moving objects and the moving queries. Similar ideas of shared execution have been exploited in the NiagaraCQ [7] for web queries, PSoup [6], and [12] for streaming queries.

Figure 1a 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"*. Each query performs a file scan on the moving object table followed by a selection filter. With shared execution, we have the execution plan of Figure 1b. The table for moving queries contains the regions of the range queries. Then, a spatial join is performed between the table of objects (points) and the table of queries (regions). The output of the spatial join is split and is sent to the queries.

For stationary objects (e.g., gas stations), the spatial join can be performed using an R-tree index [9] on the object table. Similarly, if the queries are stationary, the Q-index [19] can be used for query indexing. However, if both objects and queries are highly dynamic, the R-tree and Q-index structures result in poor performance. To avoid this drawback, we can follow one of two approaches: (1) Utilize the techniques of frequently updating R-tree (e.g., see [14, 16]) to cope with the frequent updates of moving objects and moving queries. (2) Use a spatial join algorithm that does not assume the existence of any indexing structure. Our proposed *Scalable INcremental hash-based Algorithm* (SINA) utilizes the second approach. Experimental results, given in Section 7, compare SINA with the first approach and highlights the drawbacks and advantages of each approach.

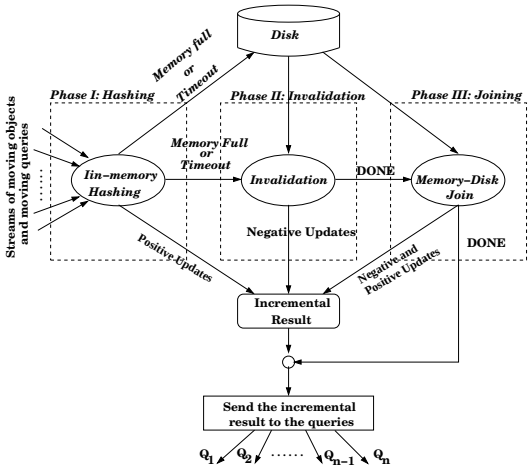


Figure 2: State diagram of SINA.

4. INCREMENTAL EVALUATION OF CONTINUOUS SPATIO-TEMPORAL QUERIES

The main idea of the *Scalable INcremental hash-based Algorithm* (SINA) is to maintain an in-memory table, termed *Updated_Answer*, that stores the positive and negative updates during the course of execution to be sent to the clients. Positive updates indicate that a certain object needs to be added to the query results. Similarly, negative updates indicate that a certain object needs to be removed from the previously reported answer. Entries in the *Updated_Answer* table have the form *(QID, Update_List(±,OID))* where QID is the query identifier, the *Update_List* is a list of $OIDs$ (object identifiers) and the type of the update (+ or −). To reduce the size of the *Updated_Answer* table, negative updates may cancel previous positive updates and vice versa. SINA sends the set of updates to the appropriate queries every T time units.

SINA has three phases: The *hashing*, *invalidation*, and *joining* phases. Figure 2 provides a state diagram of SINA. The hashing phase is continuously running where it receives incoming information from moving objects and moving queries. While tuples arrive, an in-memory hash-based join algorithm is applied between moving objects and moving queries. The result of the hashing phase is a set of positive updates added to the *Updated_Answer* table. The invalidation phase is triggered every T time units or when the memory is full to flush in-memory data into disk. The invalidation phase acts as a filter for the joining phase where the invalidation phase reports negative updates of some ob-

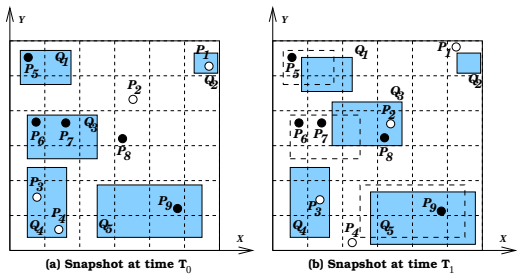


Figure 3: Example of range spatio-temporal queries.

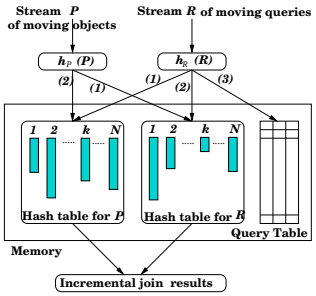


Figure 4: Phase I: Hashing.

jects to save their processing in the joining phase. The joining phase is triggered by the end of the invalidation phase to perform a join between in-memory moving objects and queries with in-disk stationary objects and queries. The joining phase results in reporting both positive and negative updates. Once the joining phase is completed, the positive and negative updates are sent to the users that issued the continuous queries.

Throughout this section, we use the example given in Figure 3 to illustrate the ideas and execution of SINA. Figure 3a gives a snapshot of the database at time T_0 with nine moving objects, p_1 to p_9, and five continuous range queries, Q_1 to Q_5. At time T_1 (Figure 3b), 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. The old query locations are plotted with dotted borders. Black objects are stationary, while white objects are moving. We use the term *"moving"* object/queries at time T_i to indicate the set of objects/queries that report a change of information from the last evaluation time T_{i-1}. Moving objects and queries are stored in memory for the evaluation time T_i. Similarly, we use the term *"stationary"* objects/queries to indicate the set of objects/queries that did not report any change of information from the last evaluation time T_{i-1}. Stationary objects and queries are stored in disk at the evaluation time T_i. Notice that stationary objects/queries at time T_i may become moving objects and queries at time T_{i+1} and vice versa.

4.1 Phase I: Hashing

Data Structure. The hashing phase maintains two in-memory hash tables, each with N buckets for sources P and R that correspond to moving objects (i.e., points) and moving queries (i.e., rectangles), respectively. In addition, for the moving queries, we keep an in-memory query table that keeps track of the corresponding buckets of the upper-left and lower-right corners of the query region. In the following, we use the symbols P_k and R_k to denote the kth bucket P and R, respectively.

Algorithm. Figures 4 and 5 provide an illustration and pseudo code of the hashing phase, respectively. Once a new moving object tuple t with hash value $k = h_P(t)$ is received (Step 2 in Figure 5), we probe the hash table R_k for moving queries that can join with t (i.e., contain t) (Step 2b in Figure 5). For the queries that satisfy the join condition (i.e., the containment of the point objects in the query region), we add positive updates to the *Updated_Answer* table (Step 2c in Figure 5). Then, we store t in the hash bucket P_k (Step 2d in Figure 5). Similarly, if a moving query tuple t is received, we probe all the hash buckets of P that intersect with t. For the objects that satisfy the join condition, we

Procedure HashingPhase(tuple t, source (P/ R)) **Begin**

1. *If there is not enough memory to accommodate t, start the InvalidationPhase()*, **return**
2. *If (source==P) //Moving object*
   (a) k = *the hash value* $h_P(t)$ *of tuple t.*
   (b) S_q = *Set of queries from joining t with queries in* R_k
   (c) *For each* $Q \in S_q$, *add* $(Q, +t)$ *to Updated_Answer*
   (d) *Store t in Bucket* P_k
   (e) **return**
3. S_k = *Set of buckets result from hash function* $h_R(t)$
4. *For each bucket* $k \in S_k$
   (a) S_o = *Set of objects from joining t with objects in* P_k
   (b) *For each* $O \in S_o$, *add* $(t, +O)$ *to Updated_Answer*
   (c) *Store a clipped part of t in Bucket* R_k
5. *Store t in the query table*

End.

Figure 5: Pseudo code of the Hashing phase

add positive updates to the *Updated_Answer* table (Step 4b in Figure 5). Then, the tuple t is clipped and is stored in all the R buckets that t overlaps. Finally, to keep track of the list of buckets that t intersects with, we store t in the in-memory query table with two bucket numbers; the upper-left and the lower-right (Step 5 in Figure 5).

Example. In the example of Figure 3, the hashing phase is concerned with objects and queries that report a change of location in the time interval $[T_0, T_1]$. Thus, the objects p_1, p_2, p_3, p_4 are joined with the queries Q_1, Q_3, Q_5. Only the positive update $(Q_3, +p_2)$ is reported.

Discussion. The hashing phase is designed to deal only with memory, thus, there is no I/O overhead. Joining the in-memory data with the in-disk objects and queries is to be performed at the joining phase. The fact that the hashing phase performs in-memory join within the hashing process enables sending early and fast results to the users. In many applications, it is desirable that the user have early and fast partial results, sometimes at the price of slightly increasing the total execution time. Similar ideas for in-memory hash-based join have been studied in the context of non-blocking join algorithms, e.g., the symmetric hash join [29], XJoin [28], and the hash-merge join [17].

4.2 Phase II: Invalidation

Data Structure. Figure 6 sketches the data structures used in the invalidation phase. The invalidation phase relies

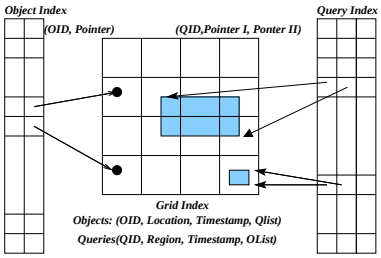


Figure 6: Phase II: Invalidation.

Procedure InvalidationPhase() **Begin**

- *For (k=0;k<MAX_GRID_CELL;k++)*
  1. *For each moving object $M_o \in P_k$, call Invalidate_Object(M_o,G_k)*
  2. *For each moving query $M_q \in R_k$*
     (a) *if $M_q \in G_k$, update the information of M_q in G_k*
     (b) *else, insert a new entry in G_k for M_q, with an OList initialized from the Updated_Answer*
- *call Invalidate_Queries()*

End.

Figure 7: Pseudo code of the Invalidation phase

on partitioning the two-dimensional space into $N \times N$ grid cells[1]. Objects and queries are stored in grid cells based on their locations. To handle skewed data distribution of objects and queries, we employ similar techniques as in [18] where we map grid cells into smaller size tiles in a round robin fashion. Tiles are directly mapped to disk-based pages. An object entry O has the form $(OID, loc, t, QList)$, where OID is the object identifier, loc is the recent location of the object, t is the timestamp of the recently reported location loc, and $QList$ is the list of the queries that O is satisfying. A query Q is clipped to all grid cells that Q overlaps with. For any grid cell C, a query entry Q has the form $(QID, region, t, OList)$, where QID is the query identifier, $region$ is the recent rectangular region of Q that intersects with C, t is the timestamp of the recently reported region, and $OList$ is the list of the objects in C that satisfy $Q.region$. In addition to the grid structure, we keep track of two auxiliary data structures; the object index and the query index. The object and query indexes are indexed on the OID and QID, respectively, and are used to provide the ability for searching the old locations of moving objects and queries given their identifiers.

Algorithm. The pseudo code of the invalidation phase is given in Figures 7, 8, and 9. The invalidation phase starts by flushing the non-empty buckets that contain moved objects (Step 1 in Figure 7) and moved queries (Step 2 in Figure 7) into the corresponding grid cells in disk. Figure 8 gives the pseudo code of invalidating a moving object M_o that is mapped into grid cell G_k. If there is an old entry of M_o in G_k, this means that M_o did not cross a cell boundary. Thus, we update only the information of M_o in G_k (Step 1 in Figure 8). If M_o is a new entry in G_k, we insert a new

[1]For simplicity, we present SINA in the context of a disk-based grid. However, the uniform grid can be substituted by more sophisticated structures e.g., the FUR-tree [16] or quad-tree-like structures [21].

Procedure Invalidate_Object(Object M_o, GridCell G_k) **Begin**

1. *If $M_o \in G_k$*
   (a) *Update the location and timestamp of M_o in G_k*
   (b) *S_q = Queries in Updated_Answer that contains M_o*
   (c) *For each query $Q \in S_q \cap M_o.QList$, add $(Q, -M_o)$ to Updated_Answer*
   (d) *$M_o.QList = M_o.QList \cup S_q$*
   (e) **return**
2. *Insert M_o as a new entry in G_k with timestamp and a QList initialized with from the Updated_Answer*
3. *G_{old} = Old cell M_o from the Object index table*
4. *If G_{old} = NULL,* **return**
5. *Retrieve O_{old}; the old entry of M_o from G_{old}*
6. *For each query $Q \in O_{old}.QList$*
   (a) *Add (Q,-M_o) to Updated_Answer table*
   (b) *Remove the entry M_o from Q.Olist*
7. *Delete the entry O_{old} from G_{old}*

End.

Figure 8: Pseudo code invalidating moving objects

entry for M_o in G_k with the current timestamp and a $QList$ that contains the moving queries from the *Updated_Answer* table that are satisfied by M_o (Step 2 in Figure 8). Then, we utilize the auxiliary structure object index using $M_o.OID$ to get the old entry O_{old} of M_o (Step 3 in Figure 8). For all queries in $O_{old}.QList$, we report negative updates to the *Updated_Answer* table and update the corresponding $OList$s (Step 6 in Figure 8). Finally, we delete the old entry of M_o (Step 7 in Figure 8).

The invalidation process of moving queries starts by flushing query parts in the corresponding disk-based cells (Step 2 in Figure 7). Similar to moving objects, we either update an old entry or insert a new one. Then, we compare the in-memory query table with the in-disk query index. For each moving query, we keep track with a set S_k that contains the cells that were part of the old region of the query, but are not in the new query region (Step 1 in Figure 9). Then, we send negative updates for each object that was part of the query answer in each grid cell of S_k (Step 1 in Figure 9). Finally, we delete the old entry of the moving query.

Example. For the example given in Figure 3, the invalidation phase is concerned only with moving objects and queries that change their locations in the time interval $[T_0, T_1]$. Moving objects p_1, p_2 do not report any updates where p_1 does not cross its cell boundaries and p_2 was not involved in any query answer at time T_0. Although p_3 is still in Q_4, however, the negative update $(Q_4, -p_3)$ is reported since object p_3 crosses its cell boundaries. To guarantee that only incremental results will be maintained, this negative tuple will be deleted in the joining phase. For object p_4, we report the negative update $(Q_4, -p_4)$. For moving queries Q_1, Q_5, we do not report any result, where they do not leave any of their old cells. Query Q_3 reports a negative update $(Q_3, -p_6)$ where Q_3 completely leaves its old cell that contains p_6. Notice that we do not report any negative update for p_7 where Q_3 still did not leave the cell that contains p_7.

Discussion. The invalidation phase uses the object index and the query index to retrieve the old information for mov-

Procedure Invalidate_Queries() **Begin**

- *For each query M_q in the in-memory query table*
  1. *S_k = Set of grid cells that was covered by the old value of M_q and not covered by the new value of M_q*
  2. *For each grid $k \in S_k$*
     (a) *Retrieve Q_{old}; the old entry of M_q in cell k*
     (b) *For each $O \in Q_{old}.OList$, add $(M_q, -O)$ to Updated_Answer and remove M_q from $O.QList$*
  3. *Delete the entry Q_{old} from k*

End.

Figure 9: Pseudo code invalidating moving queries

ing objects and moving queries that cross their cell boundaries, respectively. Another approach is to let the client send the old location information along with the new location information. In this case, there will be no need for maintaining the two auxiliary data structures. Although this approach would simplify SINA, and save I/O overhead, however, this approach lacks practicality. The main reason is that this approach assumes that the client has the ability to store its old location information, which is not guaranteed for all clients. The objective of SINA is to assume the minimal computation and storage requirement from clients.

Using auxiliary data structures to keep track of the old locations is utilized in the LUR-tree as a linked list [14] and in the frequently updated R-tree as a hash table [16]. However, the invalidation phase in SINA limits the access of the auxiliary data structures to only the objects that move out of their cells, rather than to all moved objects, which is the case in [14, 16].

The invalidation phase reports negative updates that correspond to moving objects that cross their cell boundaries and moving queries that leave some of their old cells. For moving objects and queries that move within their cell boundaries, we defer their invalidation process to the joining phase. Another approach for the invalidation phase is to report negative updates from all moving objects and queries regardless of their old locations. This approach would incur redundant I/O overhead. In the joining phase, the cells that contain in-cell moving objects or queries have to be fetched into memory to perform a join between objects and queries. Computing negative updates for the in-cell movement in the invalidation phase results in redundant operations between the two phases. Thus, the invalidation phase acts as a filter to avoid unnecessary joins in the joining phase.

4.3 Phase III: Joining

Data Structure. The joining phase does not require any additional data structure where it uses only the grid data structure that is utilized in the invalidation phase.

Algorithm. Figure 10 gives the pseudo code of the joining phase. For each grid cell, the joining phase performs two spatial join operations: (1) Joining in-memory objects with in-disk queries (Steps 1 and 2 in Figure 10), (2) Joining in-memory moving queries with in-disk objects (Steps 3 and 4 in Figure 10). For each moving object/query, we get the set of queries/objects from applying a spatial join algorithm, respectively (Steps 2a and 4a in Figure 10). Then, based on the answer set, we report positive and negative updates while updating the corresponding data structures.

Procedure JoiningPhase() **Begin**

- *For (k=0; $k <$ MAX_GRID_CELL; k++)*
  1. *Join moving objects in the in-memory bucket P_k with stationary queries in the in-disk grid cell G_k*
  2. *For each moving object $M_o \in P_k$*
     (a) *S_q = Set of queries that results from the join*
     (b) *For each query $Q \in (S_q - M_o.QList)$, add $(Q, +M_o)$ to Updated_Answer, update $Q.OList$*
     (c) *For each stationary $Q \in (M_o.QList - S_q)$, add $(Q, -M_o)$ to Updated_Answer, update $Q.OList$*
     (d) *$M_o.QList = S_q \cup M_o.QList$*
  3. *Join moving queries in the in-memory bucket R_k with stationary objects in the in-disk grid cell G_k*
  4. *For each moving query $M_q \in R_k$*
     (a) *S_o = Set of objects that results from the join*
     (b) *For each object $O \in (S_o - M_q.OList)$, add $(M_q, +O)$ to Updated_Answer, , update $O.QList$*
     (c) *For each stationary $O \in (M_q.OList - S_o)$, add $(M_q, -O)$ to Updated_Answer, update $O.QList$*
     (d) *$M_q.OList = S_o \cup M_q.OList$*
- *Send the Updated_Answer table to the users*
- *Empty all memory data structure*

End.

Figure 10: Pseudo code for the joining phase

After performing the spatial join for all grid cells, we send the *Updated_Answer* to the clients, and clear all memory data structures.

Example. For the example given in Figure 3, during this phase, moving object p_1 reports the negative update $(Q_2, -p_1)$. Object p_2 does not report any updates where there are no in-disk stationary queries to join with p_2's new cell (Q_3 is a moving query). The moving object p_3 is joined with the stationary query Q_4 that produces $(Q_4, +p_3)$ as a positive update. Notice that this positive update cancels the corresponding previously reported negative update in the invalidation phase. Thus, the size of the *Updated_Answer* table is minimized and only the incremental results are maintained. Object p_4 does not produce any results where there are no in-disk queries to join with. For moving queries, Q_1 reports the negative update $(Q_1, -p_5)$ where p_5 and Q_1 are not joined together in the upper-left corner cell. Query Q_3 reports the positive update $(Q_3, +p_8)$ as a result of the spatial join of one of the news cells covered by Q_3. Also, Q_3 reports $(Q_3, -p_7)$. Query Q_5 does not report any of the updates where object p_9 is still in the new region of Q_5.

Discussion. The joining phase only joins the cells that have new moving objects and/or queries. Cells that contain only stationary (i.e., not recently moving) objects and queries are not processed in the joining phase. In addition, cells that contain stationary or old information of moving objects and/or queries are filtered out and are processed at the invalidation phase (e.g., the cells that contain p_2, p_3, and p_4 at time T_0 in Figure 3a). Each iteration of the joining phase deals with only one grid cell. Thus, the I/O cost of each iteration is bounded by the number of disk pages of a grid cell. For the CPU time, we utilize a plane-sweep-based spatial join algorithm similar to the ones used in hash-based spatial join algorithms (e.g., [18]).

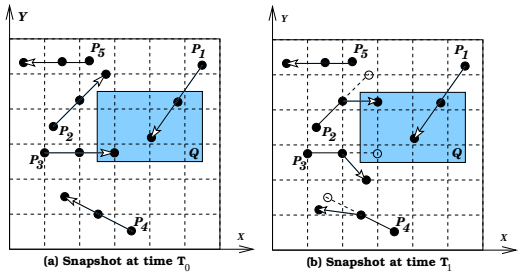


Figure 11: Querying the future

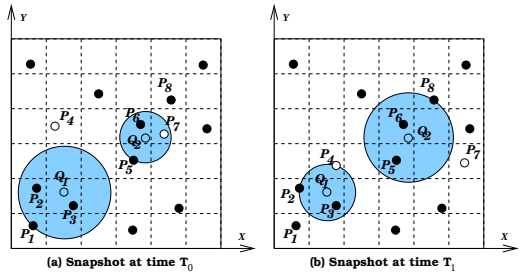


Figure 12: k-NN spatio-temporal queries

5. EXTENSIBILITY OF SINA

In this section, we explore the extensibility of SINA to support a broad class of continuous spatio-temporal queries (e.g., future, k-nearest-neighbor, and aggregate spatio-temporal queries) and to support clients that may be disconnected from the server for short periods of time (i.e., *out-of-sync* clients).

5.1 Querying the Future

Future queries [24], also termed as predictive queries [26, 30], are interested in predicting the locations of moving objects. An example of a future query is *"Alert me if a non-friendly airplane is going to cross a certain region in the next 30 minutes"*. Notice that the alert is set before the actual event happens, thus, the term future or predictive query. To support future queries, D-dimensional moving objects report their current location $\vec{x_0} = (x_1, x_2, \cdots, x_d)$ at time t_0 and a velocity vector $\vec{v} = (v_1, v_2, \cdots, v_d)$. The predicted location $\vec{x_t}$ of the moving object at any instance time $t > t_0$ is computed by $\vec{x_t} = \vec{x_0} + \vec{v}(t - t_0)$.

Figure 11a gives an example of querying the future. Five moving objects p_1 to p_5, have the ability to report their current location at time T_0 and a velocity vector that is used to predict their future locations at times T_1 and T_2. The range query Q is interested in objects that will intersect with its region at time $T_2 > T_0$. At time T_0 the rectangular query region is joined with the lines representation of the moving objects. The returned answer set of Q is (p_1, p_3). At T_1 (Figure 11b), only the objects p_2, p_3, and p_4 change their locations. Based on the new information, SINA reports only the positive update $(Q, +p_2)$ and negative update $(Q, -p_3)$ that indicate that p_2 is considered now as part of the answer set of Q while p_3 is no longer in the answer set of Q. Notice that, due to the incremental property of SINA, no tuples are produced for object p_1 where it does not change its information from the previously reported result at time T_0.

The extension of SINA to support future queries is straightforward. Moving objects are represented as lines instead of points. Thus, in the hashing phase, moving objects are clipped into several hash buckets (same as rectangular queries). In the invalidation and joining phases, moving objects will be treated as moving queries in the sense that they may span more than one grid cell. The shared execution paradigm can exactly fit for future queries. Also, moving queries do not need any special handling other than the ones used in the original description of SINA in Section 4.

5.2 k-Nearest-Neighbor Queries

Although continuous spatio-temporal k-nearest-neighbor queries (kNN) are widely addressed (e.g., see [2, 23, 25, 32]), scalability, incremental evaluation, mutability of both objects and queries, and client overhead are examples of challenges that either are overlooked wholly or partially by previous approaches. An example of a continuous kNN spatio-temporal query is *"Continuously, find the nearest three police cars relative to my current location.*

SINA can be utilized to continuously report the changes of a set of concurrent kNN queries. Figure 12a gives an example of two kNN queries where $k = 3$ issued at points Q_1 and Q_2. Assuming that both queries are issued at time T_0, we compute the first-time answer using any of the traditional algorithms of kNN queries (e.g., [13]). For Q_1, the answer would be $Q_1 = p_1, p_2, p_3$ while for Q_2, the answer would be $Q_2 = p_5, p_6, p_7$. In this case, we present Q_1 and Q_2 as circular range queries with radius equal to the distance of the kth neighbor. Later, at time T_1 (Figure 12b), object p_4 and p_7 are moved. Thus, SINA can be utilized to allow for a shared execution among the two queries and to compute the updates from the previously reported answer. Notice that the only change to the original SINA is that we utilize circular range queries rather than rectangular range queries. For Q_1, object p_4 intersect with the query region. This results in invalidating the furthest neighbor of Q_1, which is p_1. Thus, two update tuples are reported $(Q_1, -p_1)$ and $(Q_1, +p_4)$. For Q_2, the object p_7 was part of the answer at time T_0. However, after p_7 moves,the joining phase checks whether p_7 still inside the query region or not. If p_7 is outside the circular query region, we compute another nearest-neighbor, which is p_8. Thus, two update tuples are reported, $(Q_2, -p_7)$ and $(Q_2, +p_8)$. Notice that the query regions of Q_1 and Q_2 are changed from T_0 to T_1. However, this does not affect the execution of SINA, where we always store the previous region of the query.

5.3 Aggregate Queries

Continuous aggregate spatio-temporal queries are those queries that always report statistical and/or summary information. An example of aggregate queries is: *"Continuously, verify that the number of police cars in a certain area is above a certain threshold"* where the aggregate function is COUNT. Another example is: *Continuously, report the moving object with maximum speed in a certain region*, where the aggregate function is MAX.

Continuous spatio-temporal aggregate queries are recently addressed in [10] where dense areas are discovered online (i.e., areas with the number of moving objects above a certain threshold) (aggregate function COUNT). The areas to be discovered are limited to pre-defined grid cells. Thus,

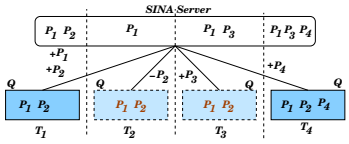


Figure 13: Example of Out-of-Sync queries

if a dense area is not aligned to a grid cell, it will not be discovered. The work in [10] can be modelled as a special instance of SINA in the following way: For a $N \times N$ gird, we can consider having N^2 spatio-temporal disjoint aggregate range queries, where each query represents a grid cell. Moreover, SINA can extend [10] to have the ability to discover pre-defined dense areas of arbitrary regions. Thus, important areas (e.g., areas around airport or in downtown) can be discovered even if it is not aligned to grid cells. All pre-defined dense areas are treated as range queries. Then, the shared execution paradigm with the incremental evaluation of SINA continuously reports the density of such areas. Positive and negative updates report only the increase and decrease of the density from the previously reported result.

5.4 Out-of-Sync Clients

Mobile objects tend to be disconnected and reconnected several times from the server for some reasons beyond their control, i.e., being out of battery, losing communication signals, being in a congested network, etc. This *out-of-sync* behavior may lead to erroneous query results in any incremental approach. Figure 13 gives an example of erroneous query result. The answer of query Q that is stored at both the client and server at time T_1 is (p_1, p_2) . At time T_2, the client is disconnected from the server. However, the server does not recognize that Q is disconnected. Thus, the server keeps computing the answer of Q, and sends the negative update $(Q, -p_2)$. Since the client is disconnected, the client could not receive this negative update. Notice the inconsistency of the stored result at the server side (p_2) and the client side (p_1, p_2). Similarly, at time T_3, the client is still disconnected. The client is connected again at time T_4. The server computes the incremental result from T_3 and sends only the positive update $(Q, +p_4)$. At this time, the client is able to update its result to be (p_1, p_2, p_4). However, this is a wrong answer, where the correct answer is kept at the server (p_1, p_3, p_4). SINA can easily be extended to resolve the *out-of-sync* problem by adding the following *catch-up* phase.

Catch-up Phase. A naive solution for the catch-up phase is once the client wakes up, it empties its previous result and sends a *wakeup* message to the server. The server replies by the query answer stored at the server side. For example, in Figure 13, at time T_4, SINA will send the whole answer (p_1, p_3, p_4). This approach is simple to implement and process in the server side. However, it may result in significant delay due to the network cost in sending the whole answer. Consider a moving query with hundreds of objects in its result that gets disconnected for a short period of time. Although, the query has missed a couple of points during its disconnected time, the server would send the complete answer to the query.

To save the network bandwidth, SINA maintains a repository of committed query answers. An answer is considered committed if it is guaranteed that the client has received it. Once the client wakes up from the disconnected mode, it sends a *wakeup* message to the server. SINA compares the latest answer for the query with the committed answer, and sends the difference of the answer in the form of positive and negative updates. For example, in Figure 13, SINA stores the committed answer of Q at time T_1 as (p_1, p_2). Then, at time T_4, SINA compares the current answer with the committed one, and send the updates $(Q, -p_2, +p_3, +p_4)$. In a real life example, the size of the incremental answer is much less than the complete answer. Thus, SINA saves in network bandwidth.

Once SINA receives any information from a moving query, SINA considers its latest answer as a committed one. However, stationary queries are required to send explicit *commit* message to SINA to enable committing the latest result. *Commit* messages can be sent at the convenient times of the clients. Handling the *commit* message is easier in the case of moving queries. This is acceptable at SINA since the *out-of-sync* problem is a property of moving queries rather than stationary queries.

6. CORRECTNESS OF SINA

In this section, we provide a proof of correctness of the *Scalable INcremental hash-based Algorithm* (SINA). The correctness proof is divided into three parts: First, we prove that SINA is complete, i.e., all result tuples are produced. Second, we prove that SINA is a duplicate-free algorithm, i.e., output tuples are produced exactly once. Third, we prove that SINA is progressive, i.e., only new results will be sent to the user.

THEOREM 1. *For any two sets of moving objects P and moving queries R, SINA produces all output results (p, r) of $P \bowtie R$, where the join condition p inside r is satisfied at any time instance t.*

PROOF. Assume that $\exists (p, r) : p \in P, r \in R$, and at some time instance t, p was located inside r. However, the tuple (p, r) is not reported by SINA. Since (p, r) satisfies the join condition, then there exists a hash bucket h such that $h = h_P(p)$ and $h \in h_R(r)$. Assume that the latest information sent from p and r were in time intervals $[T_i, T_{i+1}]$ and $[T_j, T_{j+1}]$, respectively. Then, there are exactly two possible cases:

Case 1: $i = j$. In this case, both p and r reports their recent information at the same time interval $[T_i, T_{i+1}]$. Thus, we guarantee that p and r were resident in the memory at the same time. If p arrives before r, then p will be stored in bucket P_h without joining with r. Later when r arrives, it will probe the bucket P_h, and join with p. The same proof is applicable when r arrives before p. Thus, the tuple (p, r) cannot be missed in case of $i = j$.

Case 2: $i \neq j$. Assume $i < j$. This indicates that p arrives before r. Then, in the invalidation phase, p is flushed into disk before r arrives. Once r arrives, it is stored in an in-memory hash bucket h that corresponds to the disk-based cell of object p. Since the joining phase join all in-memory hash buckets with their corresponding in-disk grid cells, we guarantee that r will be joined with p. The same proof is applicable when $i > j$ where the in-memory object p will be joined with the in-disk query r. Thus, the tuple (p, r) cannot be missed in case of $i \neq j$.

From Cases 1 and 2, we conclude that the assumption that (p, r) is not reported by SINA is not possible. Thus, SINA produces all output results. □

THEOREM 2. *At any evaluation time T_i, SINA produces the output result that corresponds to all information change in $[T_{i-1}, T_i]$ exactly once.*

PROOF. Assume that $\exists(p, r) : p \in P, r \in R$, and (p, r) satisfies the join condition. Assume that SINA reports the tuple (p, r) twice. We denote such two instances as $(p, r)_1$ and $(p, r)_2$. Since, we are interested only on tuples that satisfy the join condition (i.e., positive updates), then, we skip the invalidation phase where it produces only negative updates. Thus, we identify the following three cases:

Case 1: $(p, r)_1$ and $(p, r)_2$ are both produced in the hashing phase. Assume that p arrives after r. Once p arrives, it probes the hash bucket of r and outputs the result $(p, r)_1$. Then, during the hashing phase, only newly incoming tuples are used to probe the hash buckets of p and r. Thus, (p, r) cannot be produced again in the hashing phase.

Case 2: $(p, r)_1$ and $(p, r)_2$ are both produced in the joining phase. The joining phase produces positive updates at two outlets: in-memory moving objects with in-disk (not recently moving) queries and moving queries with in-disk objects. If $(p, r)_1$ is produced in the former outlet, then p is a moving object that reports its recent information in $[T_{i-1}, T_i]$. Thus, $(p, r)_2$ cannot be produced in the second outlet where it is concerned only with in-disk objects. The same proof is applicable when $(p, r)_1$ is produced in the second outlet. Thus, the tuple (p, r) cannot be produced again in the joining phase.

Case 3: One of the tuples, say $(p, r)_1$, is produced in the hashing phase, while the other one is produced in the joining phase. Since $(p, r)_1$ is reported in the hashing phase, then we guarantee that both p and r were in memory (moving) in the same time interval $[T_{i-1}, T_i]$. Thus, p is a moving object and r is a moving query. In the joining phase, (p, r) cannot be produced again in the first outlet where r is not stationary. Similarly, (p, r) cannot be produced again in the second outlet where p is not stationary. Thus, the tuple (p, r) cannot be produced again in the joining phase.

From the above three cases, we conclude that the assumption that the tuple (p, r) is reported twice at the evaluation time T_i is not valid. □

THEOREM 3. *For any two sets of moving objects P and moving queries R, at any evaluation time T_i, SINA produces ONLY the changes of the previously reported result at time T_{i-1}.*

PROOF. Assume that $\exists p_1, p_2, p_3 \in P, r \in R$, and only $(p_1, r), (p_2, r)$ satisfy the join condition at time T_{i-1}. Then, at time T_i, p_1 is still inside r, p_2 is moved out of r while p_3 is moved inside r. In the following we prove that only the tuples $(r, -p_2)$ and $(r, +p_3)$ are produced at time T_i. Mainly, we identify the following three cases:

Case 1: r is a moving query, p_1, p_2, and p_3 are moving objects. This case is processed only in the hashing and invalidation phases. Based on Theorem 2, the hashing phase produces only the updates $(r, +p_1)$ and $(r, +p_3)$. In the invalidation phase, $(r, +p_1)$ is deleted either in Step 1c in Figure 8 (if p_1 moves within its cell boundary) or in Step 6a in Figure 8 (if p_1 moves out of its cell) by adding the counterpart tuple $(r, -p_1)$. The tuple $(r, -p2)$ is produced only in the invalidation phase either in Step 1c or Step 6a of Figure 8.

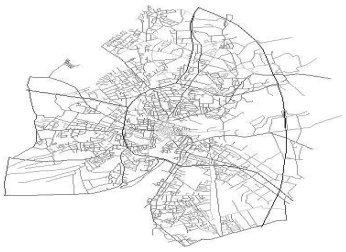


Figure 14: Road network map of Oldenburg City

Case 2: r is a moving query, p_1, p_2, and p_3 are stationary objects. This case is processed only in the invalidation and joining phases. Since p_1 is still in the answer set of r, then p_1 is inside some grid cell c that intersects with both the old and new regions of r. Thus, p_1 will be processed only in the joining phase, particulary at Step 4 in Figure 10. However, since p_1 is an old answer, no action will be taken. For object p_2, assume that C_{i-1}, C_i are the set of grid cells that are covered by r in T_{i-1}, T_i, respectively. c_2 is the gird cell of p_2. Since p_2 is not in the answer set of r at time T_i, then $c_2 \notin C_i - C_{i-1}$. If $p_2 \in C_{i-1} - C_i$, then the tuple $(r, -p_2)$ will be produced in the invalidation phase (Step 2b in Figure 9). However, if $p_2 \in C_{i-1} \cap C_i$, then the tuple $(r, -p_2)$ will be produced in the joining phase (Step 4c in Figure 10). For object p_3, since p_3 is not an old answer, then it will not be processed in the invalidation phase. Thus, the tuple $(r, +p_3)$ will be reported in the joining phase (Step 4b in Figure 10).

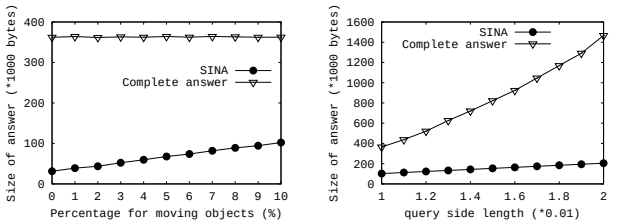
Case 3: r is a stationary query, p_1, p_2, and p_3 are moving objects. The proof is very similar to Case 2 by reversing the roles of queries and objects.

We do not include the case of stationary queries on stationary objects where it is not precessed. Also, we assume that either all $p_i's$ are moving or stationary. However, the proof is still valid for any combination of moving and stationary $p_i's$. Thus, from the above three cases, we conclude that: At time T_i, SINA only produces the change of the result from the previously reported answer at time T_{i-1}. □

7. EXPERIMENTAL RESULTS

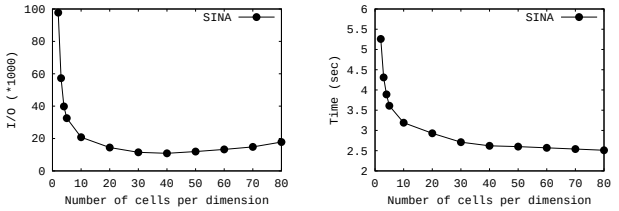
In this section, we compare the performance of SINA with the following: (1) Having an R-tree-based index on the object table. To cope with the moving objects, we implement the frequently updated R-tree [16] (FUR-tree, for short). The FUR-tree modifies the original R-tree to efficiently handling moving objects. (2) Having a Q-index [19] on the query table. Since Q-index is designed for static queries, we modify the original Q-index to employ the techniques of the FUR-tree to handle moving queries. Thus, the Q-index can handle moving queries as efficient as the FUR-tree handles moving objects. (3) Having both the FUR-tree on moving objects and the modified Q-index on the query table. Then, we employ an R-tree based spatial join algorithm [4] (RSJ, for short) to join objects and queries.

We use the *Network-based Generator of Moving Objects* [3] to generate a set of moving objects and moving queries. The input to the generator is the road map of Oldenburg (a city



(a) Moving objects (%) (b) Query size

Figure 15: The answer size



(a) I/O (b) CPU Time

Figure 16: The impact of grid size N

in Germany) given in Figure 14. The output of the generator is a set of moving points that moves on the road network of the given city. Moving objects can be cars, cyclists, pedestrians, etc. We choose some points randomly and consider them as the centers of square queries. Unless mentioned otherwise, we generate 100K moving objects and 100K moving queries. Each moving object or query reports its new information (if changed) every 100 seconds. The space is represented as the unit square, query sizes are assumed to be square regions of side length 0.01. SINA is adopted to refresh query results every $T = 10$ seconds. The percentage of object and queries that report a change of information within T seconds is 10% of the moving objects and queries, respectively.

All the experiments in this section are conducted on Intel Pentium IV CPU 1.4GHz with 256MB RAM running Linux 2.4.4. SINA is implemented using GNU C++. The page size is 2KB. We implement FUR-tree, Q-index, and RSJ using the original implementation of R*-tree [1]. Our performance measures are the I/O overhead and CPU time incurred. For the I/O, we consider that the first two levels of any R-tree-based structures are in memory. The CPU time is computed as the time used to perform the spatial join in the memory (i.e., once the page is retrieved from disk). For SINA, the CPU time also includes the time that the hashing phase consumes for the in-memory join.

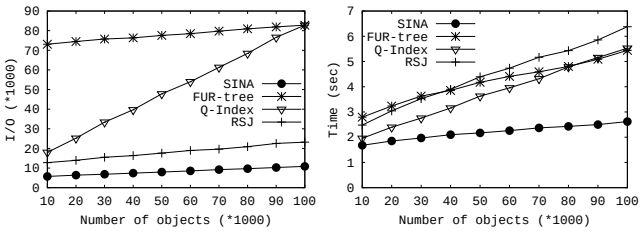
7.1 Properties of SINA

7.1.1 Size of the Result

Figure 15 compares between the size of the answer returned by SINA and the size of the complete answer returned by any non-incremental algorithm. In Figure 15a, the percentage of moving objects varies from 0% to 10%. The size of the complete answer is constant and is orders of magnitude of the size of the incremental answer returned by SINA. A complete answer is not affected by recently moved objects. However, for SINA, the size of the answer is increasing slightly, where it is affected by the number of objects being evaluated at every T seconds. In Figure 15b, the query side length varies from 0.01 to 0.02. The size of the complete answer is increased dramatically to up to seven times that of the incremental result returned by SINA. The saving in the size of the answer directly affect the communication cost from the server to the clients.

7.1.2 Number of Grid Cells

Figures 16a and 16b give the effect of increasing the grid size N on the I/O and CPU time incurred by SINA, respectively. With small number of grid cells (i.e, less than 10), each cell contains a large number of disk pages. Thus a spatial join within each cell results in excessive I/O and CPU time. On the other hand, with a large number of grid cells (i.e., more than 60), each cell contains a small number of moving objects and queries. Although this results in lower CPU time, where the spatial join is performed among few tuples. However, disk pages are under utilized. Thus, additional I/O overhead will be incurred. Based on this experiment, we set the number of grid cells N along one dimension to be 40.

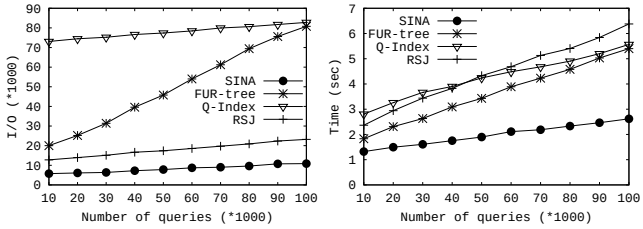


(a) I/O (b) CPU Time

Figure 17: Scalability with number of objects

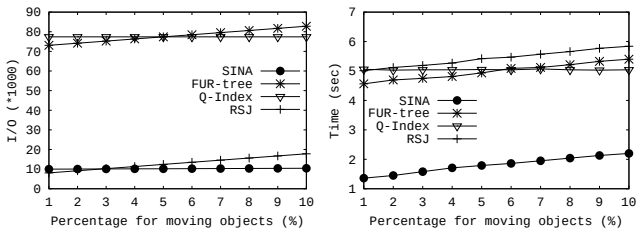
7.2 Number of Objects/Queries

In this section, we compare the scalability of SINA with the FUR-tree, Q-index, and RSJ algorithms. Figures 17a and 17b give the effect of increasing the number of moving objects from 10K to 100K on I/O and CPU time, respectively. In Figure 17a, SINA outperforms all other algorithms. RSJ has double the I/O's of SINA due to the R-tree update cost. Notice that the performance of the R-trees is degraded with the increase in the number of moving objects and moving queries. The performance of Q-index is dramatically degraded with the increase of the number of moving objects as moving objects are not indexed. The FUR-tree has the worst performance for all cases where there is no index of the 100K queries. However, the performance is slightly affected by the increase of the moving objects. The slight increase is due to the maintenance of the increasing size of the moving objects. When the number of moving objects is increased up to 100K, both the FUR-tree and the Q-index have similar performance which is eight times worse than that of the performance of SINA. The main reason is that



(a) I/O (b) CPU Time

Figure 18: Scalability with number of queries



(a) I/O (b) CPU Time

Figure 19: Percentage of moving objects

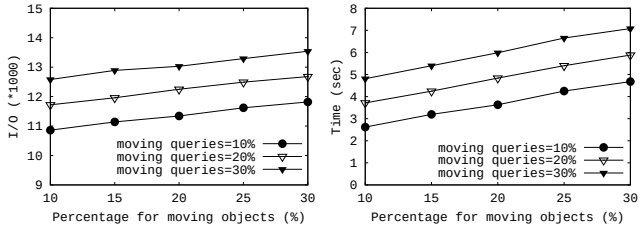
both FUR-tree and Q-index utilize only one index structure. Thus, the non-indexed objects and queries worsen the performance of FUR-tree and Q-index, respectively.

In Figure 17b, SINA has the lowest CPU time. The relative performance of SINA over other R-tree-based algorithms increases with the increase of the number of moving objects. The main reason is that the update cost of SINA is much lower than updating R-tree structures. As the number of moving objects increases, the quality of the bounding rectangles in the R-tree structure is degraded. Thus, searching and querying an R-tree incurs higher CPU time. The RSJ algorithm gives lower performance in CPU time than FUR-tree and Q-index since RSJ needs to update in two R-trees. The performance of RSJ ranges from 1.5 to 3 times worse than the performance of SINA.

Figure 18 gives similar experiment to Figure 17 with exchanging the roles of objects and queries. Since both SINA and RSJ treat objects and queries similarly, their performance is similar to the one in Figure 17. However, the FUR-tree and Q-index exchange their performance as they deal with objects and queries differently.

7.3 Percentage of Moving Objects/Queries

Figure 19 investigates the effect of increasing the percentage of the number of moving objects and queries on the performance of SINA and R-tree-based algorithms. The percentage of moving objects varies from 1% to 10%. The percentage of moving queries is set to 5%. For the I/O overhead (Figure 19a), RSJ has similar performance as SINA for up to 5% of moving objects. Then, RSJ incurs up to double the number of I/O's over that of SINA for 10% of moving objects. Both the FUR-tree and the Q-index have similar performance which is almost eight times of magnitude worse than that of SINA. When the percentage of moving objects is lower than 5% (i.e., lower than the percentage of moving queries), the FUR-tree has better performance. When the percentage of the number of moving objects and moving queries are equal (i.e., 5%) both FUR-tree and Q-index have similar performance. Basically, the performance of FUR-tree and Q-index are degraded with the increase of the percentage of moving objects and moving queries, respectively.



(a) I/O (b) CPU Time

Figure 20: Scalability of SINA with update rates

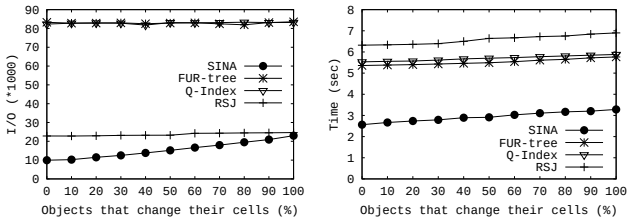
For the CPU time (Figure 19b), SINA outperforms all R-tree based algorithms. This is mainly due to the high update cost of the R-tree. The RSJ algorithm has the highest CPU time, where it updates in two R-trees. In addition, SINA computes incremental results while R-tree-based algorithms are non-incremental.

Similar performance is achieved when fixing the number of moving objects to 5% while varying the number of moving queries from 0% to 10%. The only difference is that we replace the roles of objects and queries. Thus, the performance of the FUR-tree and Q-index is exchanged while SINA and SRJ maintain their performance.

In Figure 19, we limit the number of moving queries to 5% and the number of moving objects to 10%. Having more dynamic environment degrades the performance of all R-tree based algorithms. In the following experiment, we explore the scalability of SINA in terms of handling highly dynamic environments. In Figure 20, the percentage of moving objects varies from 10% to 30%. We plot three lines for SINA that correspond to the percentage of moving queries as 10%, 20%, and 30%. We do not include any performance results of any of the R-tree-based algorithms where their performance is dramatically degraded in highly dynamic environments. Figures 20a, and 20b gives the I/O and CPU time incurred by SINA, respectively. The trend of SINA is similar with all percentages of moving queries. Also, the performance of SINA increases linearly with the increase of moving objects. Thus, SINA is more suitable for highly dynamic environments.

7.4 Locality of movement

This section investigates the effect of locality of movement on SINA and R-tree-based algorithms. By locality of movement, we mean that objects and queries are moving within a certain distance. As an extreme example, if all objects are moving within small distance, then at each evaluation time T of SINA, all objects and queries are moving within



(a) I/O (b) CPU Time

Figure 21: Effect of movement locality

their cells. Thus, SINA achieves its best performance. On the other side, SINA has its worst performance if 100% of the objects change their cells. By tuning the moving distance of moving objects, we can keep track of the number of moving objects that cross their cell boundaries. Figures 21a and 21b give the effect of the movement locality on the I/O and CPU time, respectively. For I/O, even the worst case of SINA is still better than R-tree-based algorithms (similar to RSJ and four times better than the FUR-tree and the Q-index). The performance of R-tree-based algorithms is almost not affected even all objects change their cells. The main reason is that changing the cell in the grid structure does not necessarily mean changing the R-tree node. For the CPU time, SINA outperforms all other algorithms by two orders of magnitude. In addition, the performance of SINA has only slight increase with the number of objects that change their cells.

8. CONCLUSION

This paper introduces the *Scalable INcremental hash-based Algorithm* (SINA, for short); a new algorithm for evaluating a set of concurrent continuous spatio-temporal range queries. SINA employs the *shared execution* and *incremental evaluation* paradigms to achieve scalability and efficient processing of continuous spatio-temporal queries. SINA has three phases: Hashing phase, invalidation phase, and joining phase. The hashing phase employs an in-memory hash based join algorithm that results in a set of positive updates. The invalidation phase is triggered every T time units or when the memory is full to produce a set of negative updates. Then, the joining phase is triggered to produce a set of both positive and negative updates that result from joining in-memory data with in-disk data. We discussed the extensibility of SINA to support a wide variety of spatio-temporal queries and *out-of-sync* clients. The correctness of SINA is proved in terms of completeness, uniqueness, and progressiveness. Comprehensive experiments show that the performance of SINA is orders of magnitude better than other R-tree based algorithms where the experiments demonstrate that SINA is: (1) Scalable to a large number of moving objects and/or moving queries, (2) Stable in highly dynamic environments. Finally, SINA saves in the network bandwidth by minimizing the data sent to clients.

9. REFERENCES

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, 1990.

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

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

[4] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. In P. Buneman and S. Jajodia, editors, *SIGMOD*, 1993.

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

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

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

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

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

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

[13] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2), 1999.

[14] 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.

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

[16] 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.

[17] M. F. Mokbel, M. Lu, and W. G. Aref. Hash-merge Join: A Non-blocking Join algorithm for Producing Fast and Early Join Results. In *ICDE*, 2004.

[18] J. M. Patel and D. J. DeWitt. Partition Based Spatial-Merge Join. In *SIGMOD*, 1996.

[19] 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.

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

[21] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2), 1984.

[22] T. K. Sellis. Multiple-Query Optimization. *TODS*, 13(1), 1988.

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

[24] 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.

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

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

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

[28] T. Urhan and M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2), 2000.

[29] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *PDIS 1991*, 1991.

[30] O. Wolfson and H. Yin. Accuracy and Resource Concumption in Tracking and Location Prediction. In *SSTD*, 2003.

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

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