2005

SOLE: Scalable On-Line Execution of Continuous Queries on Spatio-temporal Data Streams

Mohamed F. Mokbel

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Report Number:
05-016

SOLE: SCALABLE ON-LINE EXECUTION OF CONTINUOUS QUERIES ON SPATIO-TEMPORAL DATA STREAMS

Mohamed F. Mokbel
Walid G. Aref

CSD TR #05-016
July 2005

vldb manuscript No.
(will be inserted by the editor)

Mohamed F. Mokbel · Walid G. Aref

SOLE: Scalable On-Line Execution of Continuous Queries on Spatio-temporal Data Streams

the date of receipt and acceptance should be inserted later

Abstract This paper presents the *Scalable On-Line Execution* algorithm (SOLE, for short) for continuous and on-line evaluation of concurrent continuous spatio-temporal queries over data streams. Incoming spatio-temporal data streams are processed in-memory against a set of outstanding continuous queries. The SOLE algorithm utilizes the scarce memory resource efficiently by keeping track of only the *significant* objects. In-memory stored objects are expired (i.e., dropped) from memory once they become *insignificant*. SOLE is a scalable algorithm where all the continuous outstanding queries share the same buffer pool. In addition, SOLE is presented as a spatio-temporal join between two input streams, a stream of spatio-temporal objects and a stream of spatio-temporal queries. To cope with intervals of high arrival rates of objects and/or queries, SOLE utilizes a *self-tuning* approach based on *load-shedding* where some of the stored objects are dropped from memory. SOLE is implemented as a pipelined query operator that can be combined with traditional query operators in a query execution plan to support a wide variety of continuous queries. Performance experiments based on a real implementation of SOLE inside a prototype of a data stream management system show the scalability and efficiency of SOLE in highly dynamic environments.

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

Mohamed F. Mokbel
Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN, 55455 E-mail: mokbel@cs.umn.edu

Walid G. Aref
Department of Computer Science, Purdue University, West Lafayette, IN 47907 E-mail: aref@cs.purdue.edu

1 Introduction

The rapid increase of spatio-temporal applications calls for new query processing techniques to deal with the continuous arrival of spatio-temporal data streams. Examples of spatio-temporal applications include location-aware services [28], traffic monitoring [34], and enhanced 911 services [19]. Recent research efforts for continuous spatio-temporal query processing (e.g., see [20–25, 33, 29, 36, 39, 41, 42, 44]) suffer from one or more of the following drawbacks: (1) They rely mainly on the ability of storing and indexing spatio-temporal data. Such indexing schemes fail in practice to cope with high arrival rates of spatio-temporal data streams where only in-memory algorithms for continuous queries can be realized. (2) Most of the proposed techniques are based on high level implementations on top of the database engine. Such high level implementation does not scale up with the large number of moving objects in highly dynamic environments. (3) Most of the algorithms are tailored to support only one query type. From a system point of view, it is more attractive to have one unified framework that can support various query types.

On the other side, research efforts in data stream management systems (e.g., see [2, 6, 10, 11, 32]) focus mainly on processing continuous queries over data streams. However, the spatial and temporal properties of both data streams and continuous queries are overlooked. Continuous query processing in spatio-temporal streams has the following distinguishing characteristics: (1) Queries as well as data have the ability to continuously change their locations. Due to this mobility, any delay in processing spatio-temporal queries may result in an obsolete answer. (2) An object may be added to or removed from the answer set of a spatio-temporal query. Consider moving vehicles that move in and out of a certain query region. (3) The commonly used model of *sliding-window* queries [3, 4, 15] does not support common spatio-temporal queries that are interested on the current state of the database rather than on the recent

historical state. The current state of a database is a combination of recently received data and old data that has not been updated recently.

In this paper, we propose the *Scalable On-Line Execution* algorithm (SOLE, for short) for continuous and on-line evaluation of concurrent continuous spatio-temporal queries over spatio-temporal data streams. SOLE combines the recent advances of both spatio-temporal continuous query processors and data stream management systems. On-line execution is achieved in SOLE by allowing only in-memory processing of incoming spatio-temporal data streams. The scarce memory resource is efficiently utilized by keeping track of only those objects that are considered *significant*. Scalability in SOLE is achieved by using a shared buffer pool that is accessible by all outstanding queries. Furthermore, SOLE is presented as a spatio-temporal join between two input streams; a stream of spatio-temporal objects and a stream of spatio-temporal queries. To cope with intervals of very high arrival rates of objects and/or queries, SOLE adopts a *self-tuning* approach based on *load-shedding*. The main idea is to dynamically adopt the notion of *significant* objects based on the current load. Thus, in-memory stored objects that become *insignificant* with respect to the new notion may be dropped from memory. In addition, newly incoming objects are admitted to the system only if they are considered *significant*. The main goal of *self-tuning* in SOLE is to support larger numbers of continuous queries, yet with an approximate answer.

Two alternative approaches exist for implementing spatio-temporal algorithms in database systems: using *table functions* or encapsulating the algorithm into a *physical pipelined operator*. In the first approach, which is employed by existing spatio-temporal algorithms, algorithms are implemented using SQL table functions [37]. Since there is no straightforward method of pushing query predicates into table functions [38], the performance of this table function is severely limited and the approach does not give enough flexibility in optimizing the issued queries. The second approach, which we adopt in SOLE, is to define a query operator that can be part of the query execution plan. The SOLE operator can be combined with traditional operators (e.g., join, aggregates, and distinct) to support a wide variety of spatio-temporal queries. In addition, with the SOLE operator, the query optimizer can support multiple candidate execution plans.

The rest of this paper is organized as follows: Section 2 highlights related work. The basic concepts of SOLE are discussed in Section 3. The SOLE algorithm is presented in Section 5. Approximate query processing in SOLE via *load shedding* and *self tuning* is presented in Section 6. Experimental results that are based on a real implementation of SOLE inside a data stream management system are presented in Section 7. Finally, Section 8 concludes the paper.

2 Related Work

Up to the authors' knowledge, SOLE provides the first attempt to furnish query processors in data stream management systems with the required operators and algorithms to support a scalable execution of concurrent continuous spatio-temporal queries over spatio-temporal data streams. Since SOLE bridges the areas of spatio-temporal databases and data stream management systems, in this section we discuss the related work in each area separately.

2.1 Spatio-temporal Databases

Existing algorithms for continuous spatio-temporal query processing focus mainly on materializing incoming spatio-temporal data in disk-based indexing structures (e.g., hash tables [9,40], grid files [14,29,35], the B-tree [22], the R-tree [23,25], and the TPR-tree [39, 42]). Scalable execution of concurrent spatio-temporal queries is addressed recently for centralized [14,36] and distributed environments [8,14]. However, the underlying data structure is either a disk-based gird structure [14,29] or a disk-based R-tree [8,36]. None of these techniques deal with the issue of spatio-temporal data streams. Issues of high arrival rates, infinite nature of data, and spatio-temporal streams are overlooked by these approaches. With the notion of data streams, only in-memory algorithms and data structures can be realized.

The most related work to SOLE in the context of spatio-temporal databases is the SINA framework [29]. SOLE has common functionalities with SINA where both of them utilize a shared grid structure to produce incremental results in the form of *positive* and *negative* updates. However, SOLE distinguishes itself from SINA and other spatio-temporal query processors in the following aspects: (1) SOLE is an in-memory algorithm where all data structures are memory based. (2) SOLE is equipped with *load shedding* techniques to cope with intervals of high arrival rates of moving objects and/or queries. (3) As a result of the streaming environment, SOLE deals with new challenging issues, e.g., uncertainty in query areas, scarce memory resources, and approximate query processing. (4) SOLE is encapsulated into a physical non-blocking pipelined query operator where the result of SOLE is produced one tuple at a time. Previous spatio-temporal query processors (e.g., SINA) can be implemented only as a table function where the result is produced periodically in batches.

2.2 Data Stream Management Systems

Existing prototypes for data stream management systems [1,10,12,32] aim to efficiently support continuous

queries over data streams. However, the spatial and temporal properties of data streams and/or continuous queries are overlooked by these prototypes. With limited memory resources, existing stream query processors adopt the concept of *sliding* windows to limit the number of tuples stored in-memory to only the recent tuples [3,4,15]. Such model is not appropriate for many spatio-temporal applications where the focus is on the current status of the database rather than on the recent past. The only work for continuous queries over spatio-temporal streams is the GPAC algorithm [27]. However, GPAC is concerned only with the execution of a **single** outstanding continuous query. In a typical data stream environment, there is a huge number of outstanding continuous queries in which GPAC cannot afford.

Scalable execution of continuous queries in traditional data streams aim to either detect common subexpressions [11,12,26] or share resources at the operator level [3,13,16]. SOLE exploits both paradigms where evaluating multiple spatio-temporal queries is performed as a spatio-temporal join between an object stream and a query stream while a shared memory resource (buffer pool) is maintained to support all continuous queries. *Load shedding* in data stream management systems is addressed recently in [5,43]. The main idea to add a special operator to the query plan to regulate the load by discarding unimportant incoming tuples. Load shedding techniques in SOLE are distinguished from other approaches where in addition to discarding some of the incoming tuples, SOLE voluntary drops some of the tuples stored in-memory.

The most related work to SOLE in the context of data stream management systems is the NiagaraCQ framework [12]. SOLE has common functionalities with NiagaraCQ where both of them utilize a shared operator to join a set of objects with a set of queries. However, SOLE distinguishes itself from NiagaraCQ and other data stream management systems in the following: (1) As a result of the spatio-temporal environment, SOLE has to deal with new challenging issues, e.g., moving queries, uncertainty in query areas, *positive* and *negative* updates to the query result. (2) In a highly overloaded system, SOLE provides approximate results by employing *load shedding* techniques. (3) In addition to sharing the query operator as in NiagaraCQ, SOLE share memory resources at the operator level.

3 Basic Concepts in SOLE

In this section, we discuss the basic concepts of SOLE that include: The input/output model, supporting various queries, SOLE pipelined operator, and the SQL syntax.

3.1 Input/Output Model

Input. The input to SOLE is two streams: (1) A stream of spatio-temporal data that is sent from continuously moving objects with the format (OID, Loc, T), where OID is the object identifier, and Loc is the current location of the moving object at time T. Moving objects are required to send updates of their locations periodically. Failure to do so results in considering the moving object as disconnected. (2) A stream of continuous queries. Queries can be sent either from moving objects or from external entities (e.g., a traffic administrator). In general, a query Q is represented as $(QID, Region)$, where QID is the query identifier, and $Region$ is the spatial area covered by Q.

Output. SOLE employs an incremental evaluation paradigm similar to the one used in SINA [29]. The main idea is to avoid continuous reevaluation of continuous spatio-temporal queries. Instead, SOLE updates the query result by computing and sending only updates of the previously reported answer. SOLE distinguish between two types of query updates: *Positive updates* and *negative updates*. A *positive* update indicates that a certain object needs to be added to the result set of a certain query. In contrast, a *negative* update indicates that a certain object is no longer in the answer set of a certain query. Thus, the output of SOLE is a stream of tuples with the format $(QID, \pm, OID)$, where QID is the query identifier that would receive this output tuple, $\pm$ indicates whether this output is a *positive* or *negative* updates. A *positive/negative* update indicates the addition/removal of object OID to/from query QID.

3.2 SOLE as a Pipelined Operator

SOLE is encapsulated into a physical pipelined operator that can interact with traditional query operators in a large pipelined query plan. Having the SOLE operator either in the bottom or in the middle of the query pipeline requires that all the above operators be equipped with special mechanisms to handle *negative* tuples. Fortunately, recent data stream management systems (e.g., Borealis [1], NILE [18], STREAM [32]) have the ability to process such negative tuples.

Basically, *negative* tuples are processed in traditional operators as follows: *Selection* and *Join* operators handle *negative* tuples in the same way as *positive* tuples. The only difference is that the output will be in the form of a *negative* tuple. *Aggregates* update their aggregate functions by considering the received *negative* tuple. The *Distinct* operator reports a *negative* tuple at the output only if the corresponding *positive* tuple is in the recently reported result. For detailed algorithms about handling the *negative* tuples in various traditional query operators, the reader is referred to [17].

3.3 Supporting Various Query Types

SOLE is a unified framework that deals with range queries as well as k-nearest-neighbor (kNN) queries. In addition SOLE supports both stationary and moving queries with the same framework.

Moving Queries. Each moving query is bounded to a *focal* object. For example, if a moving object M submits a query Q that asks about objects within a certain range of M, then M is considered the *focal* object of Q. A moving query Q is represented as $(QID, FocalID, Region)$, where QID is the query identifier, $FocalID$ is the object identifier that submits Q, and $Region$ is the spatial area of Q.

kNN Queries. A kNN query is represented as a circular range query. The only difference is that the size of the query range may grow or shrink based on the movement of the query and objects of interest. Initially, a kNN query is submitted to SOLE with the format $(QID, center, k)$ or $(QID, FocalID, k)$ for stationary and moving queries, respectively. Thus, the center of the query circular region is either stated explicitly as in stationary queries or implicitly as the current location of the object $FocalID$ in case of moving queries. Once the kNN query is registered in SOLE, the first incoming k objects are considered as the initial query answer. The radius of the circular region is determined by the distance from the query center to the current kth farthest neighbor. Once the kNN query determines its initial circular region, the query execution continues as a regular range query, yet with a variable size. Whenever a newly coming object P lies inside the circular query region, P removes the kth farthest neighbor from the answer set (with a *negative* update) and adds itself to the answer set (with a *positive* update). The query circular region is *shrunk* to reflect the new kth neighbor. Similarly, if an object P, that is one of the k neighbors, updates its location to be outside the circular region, we expand the query circular region to reflect the fact that P is considered the farthest kth neighbor. Notice that in case of expanding the query region, we do not output any updates.

3.4 SQL Syntax

Since SOLE is implemented as a query operator, we use the following SQL syntax that invoke the processing of SOLE.

```
SELECT select_clause
FROM from_clause
WHERE where_clause
INSIDE in_clause
kNN knn_clause
```

The *in_clause* may have one of two forms:

- Static range query (x_1, y_1, x_2, y_2), where (x_1, y_1) and (x_2, y_2) represent the top left and bottom right corners of the rectangular range query.
- Moving rectangular range query $('M', ID, xdist, ydist)$, where $'M'$ is a flag indicates that the query is moving, ID is the identifier of the query *focal* point, $xdist$ is the length of the query rectangle, and $ydist$ is the width of the query rectangle.

Similarly, the *knn_clause* may have one of two forms:

- Static kNN query (k, x, y), where k is the number of the neighbors to be maintained, and (x, y) is the center of the query point.
- Moving kNN query $('M', k, ID)$, where $'M'$ is a flag indicates that the query is moving, k is the number of neighbors to be maintained, and ID is the identifier of the query *focal* point.

4 Single Execution of Continuous Queries in SOLE

To clarify many of the ideas used in SOLE, in this section, we present the SOLE in the context of single query execution [27]. In the next section, we show how SOLE can be generalized to the case of evaluating multiple concurrent continuous spatio-temporal queries. Assuming that for a query Q, the query answer is stored in $Q.Answer$, then, whenever SOLE receives a data input of object P, SOLE distinguishes among four cases:

- **Case I:** $P \in Q.Answer$ and P satisfies Q (e.g., Q_1 in Figure 1a). As SOLE processes only the updates of the previously reported result, P will neither be processed nor will be sent to the user.
- **Case II:** $P \in Q.Answer$ and P does not satisfy Q (Figure 1b). In this case, SOLE reports a *negative* update P^- to the user.
- **Case III:** $P \notin Q.Answer$ and P satisfies Q (Figure 1c). In this case, SOLE reports a *positive* update to the user.
- **Case IV:** $P \notin Q.Answer$ and P does not satisfy Q (e.g., Q_2 in Figure 1a). In this case, P has no effect on Q. Thus, P will neither be processed nor will be sent to the user.

On the other side, whenever SOLE receives an update from a moving query, it classifies in-memory stored objects into four categories C_1 to C_4 where: (1) $C_1 \subset Q.Answer$ and C_1 satisfies the new $Q.Region$ (e.g., white objects in Figure 1d). SOLE does not process any of the objects in C_1. (2) $C_2 \subset Q.Answer$, C_2 does not satisfy the query region (e.g., gray objects in Figure 1d). For each data object in C_2, SOLE produces a *negative* update. (3) $C_3 \not\subset Q.Answer$ and C_3 satisfies the new $Q.Region$ (e.g., black objects in Figure 1d). For each data object in C_3, SOLE produces a *positive* update. (4) $C_4 \not\subset Q.Answer$ and C_4 does not satisfy $Q.Region$. SOLE does not process objects in C_4.

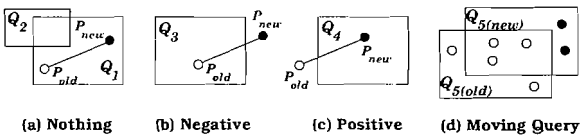


Fig. 1 Positive/Negative updates in SOLE.

4.1 Uncertainty in Spatio-temporal Queries

A key feature of SOLE is to utilize the scarce memory resource efficiently by keeping track of only those objects that satisfy at least one outstanding query. However, a straightforward application of this feature may result in *uncertainty* areas.

The *uncertainty* area of a query Q is defined as follows:

Definition 1 The uncertainty area of query Q is the spatial area of Q that may contain potential moving objects that satisfy Q, with Q not being aware of the contents of this area.

Figure 2 gives three consecutive snapshots of seven objects P_1 to P_7, a moving range queries Q_1, and a k-nearest-neighbor query ($k = 2$) Q_2. Two types of uncertainties are distinguished:

1. **Moving query Q_1.** At time T_0 (Figure 2a), P_1 is outside the area of Q_1. Thus, P_1 is not physically stored in the database. Recall that only objects that satisfy the query region are stored in the database. At time T_1 (Figure 2b), Q_1 is moved. The shaded area in Q_1 represents its *uncertainty* area. Although P_1 is inside the new query region, P_1 is not reported in the query answer where it is not actually stored. At T_2 (Figure 2c), P_1 moves out of the query region. Thus, P_1 is never reported at the query result, although it was inside the query region in the time interval $[T_1, T_2]$.
2. **Stationary query Q_2.** At time T_0, the answer of Q_2 is (P_5, P_6). The query circular region is centered at Q_2 with its radius being the distance from Q_2 to P_5. Since P_7 is outside the query spatial region, P_7 is not stored in the database. At T_1, P_5 is moved far from Q_2. Since Q_2 is aware only of P_5 and P_6, we extend the region of Q_2 to include the new location of P_5. Thus, an *uncertainty* area is produced. Notice that Q_2 is unaware of P_7 since P_7 is not stored in the database. At T_2, P_7 moves out of the new query region. Thus, P_7 never appears as an answer of Q_2, although it should have been part of the answer in the time interval $[T_1, T_2]$.

4.2 Avoiding Uncertainty in SOLE

SOLE avoids uncertainty areas in spatio-temporal queries using a *caching* technique. The main idea is to predict

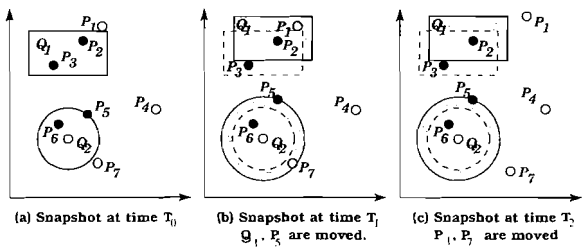


Fig. 2 Uncertainty in spatio-temporal queries.

the uncertainty area of a continuous query Q and *cache* in-memory all moving objects that lie in Q's uncertainty area. Whenever an uncertainty area is produced, we probe the in-memory *cache* and produce the result immediately. A *conservative* approach for *caching* is to expand the query region in all directions with the maximum possible distance that a moving object can travel between any two consecutive updates. Such *conservative* approach completely avoids uncertainty areas where it is guaranteed that all objects in the uncertainty area are stored in the *cache*.

Figure 3 gives an example of using *caching* to avoid uncertainty in moving queries. The shaded area represents the query region. The cached area is represented as a dashed rectangle. Moving objects that belong to the query answer or to the query's cache area are plotted as white or gray circles, respectively. At time T_0 (Figure 3a), two objects satisfy the query answer (P_1, P_2), three objects are in the cache area (P_3, P_4, P_5), and two objects outside the cache area (P_6, P_7). Only objects that either in the query or the cache area are stored in-memory. At T_1 (Figure 3b), all objects change their locations. However, we only report P_2^- and P_3^+. The cache area is updated to contain (P_2, P_4, P_6). Changes in the cache area do NOT result in any updates. At T_2 (Figure 3c), the query Q moves within its cache area. Two updates are sent to the user: P_3^- andP_4^+. The cache area is adjusted to contain P_3 and P_6 only. Notice that without employing the cache area, we would miss P_4^+.

The *conservative* caching approach requires only the knowledge of the maximum object speed, which is typically available in moving object applications (e.g., moving cars in road network have limited speeds). This is in contrast to all validity region approaches (e.g., the safe region [36], the valid region [46], and the No-Action region [45]) that require the knowledge of the locations of other objects. This information is not available in our case since SOLE is aware only of objects that satisfy the query predicate. Thus, validity region approaches are not applicable in the case of spatio-temporal streams.

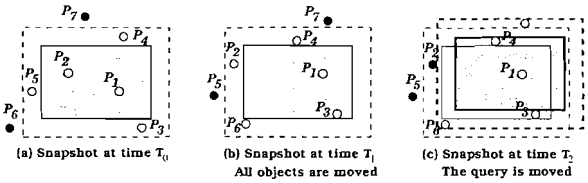


Fig. 3 Avoiding uncertainty in SOLE.

5 SOLE: Scalable On-Line Execution of Continuous Queries

In a typical spatio-temporal application (e.g., location-aware servers), there are large numbers of concurrent spatio-temporal continuous queries. Dealing with each query as a separate entity (e.g., as discussed in Section 4) would easily consume the system resources and degrade the system performance. In this section, we present the scalability of SOLE in terms of handling large numbers of concurrent continuous queries of mixed types (e.g., range and kNN queries). Without loss of generality, all the discussion in the rest of this paper is presented in the context of stationary and moving range queries. The applicability to k-nearest-neighbor queries is straightforward as described in Section 3.

5.1 Overview of Sharing in SOLE

Figure 4a gives the pipelined execution of N queries (Q_1 to Q_N) of various types with no sharing, i.e., each query is considered a separate entity. The input data stream goes through each spatio-temporal query operator separately. With each operator, we keep track of a separate buffer that contains all the objects that are needed by this query (e.g., objects that are inside the query region or its cache area). With a separate buffer for each single query, the memory can be exhausted with a small number of continuous queries.

Figure 4b gives the pipelined execution of the same N queries as in Figure 4a, yet with the shared SOLE operator. The problem of evaluating concurrent continuous queries is reduced to a spatio-temporal join between two streams: a stream of moving objects and a stream of continuous spatio-temporal queries. The *shared* spatio-temporal join operator has a shared buffer pool that is accessible by all continuous queries. The output of the *shared* SOLE operator has the form $(Q_i, \pm P_j)$ which indicates an addition or removal of object P_j to/from query Q_i. The shared SOLE operator is followed by a *split* operator that distributes the output of SOLE either to the users or to the various query operators. The *split* operator is similar to the one used in NiagaraCQ [12] and it is out of the focus of this paper. Our focus is in realizing: (1) The shared memory buffer, and (2) The shared SOLE spatio-temporal join operator.

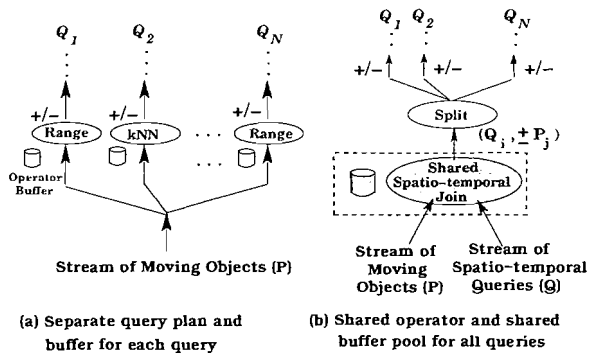


Fig. 4 Overview of shared execution in SOLE.

5.2 Shared Memory Buffer

SOLE maintains a simple grid structure as an in-memory shared buffer pool among all continuous queries and objects. The shared buffer pool is logically divided into two parts: a query buffer that stores all outstanding continuous queries and an object buffer that is concerned with moving objects. In addition to the grid structure, SOLE employs a hash table h to index moving objects based on their identifers.

To optimize the scarce memory resource, SOLE employs two main techniques: (1) Rather than redundantly storing a moving object P multiple times with each query Q_i that needs P, SOLE stores P at most once along with a reference counter that indicates the number of continuous queries that need P. (2) Rather than storing all moving objects, SOLE keeps track with only the *significant* objects. *Insignificant* objects are ignored (i.e., dropped) from memory. *Significant* objects are defined as follows:

Definition 2 A moving object P is considered **significant** if P satisfies any of the following two conditions: (1) There is at least one outstanding continuous query Q that **shows interest** in object P (i.e., P has a non-zero reference counter), (2) P is the focal object of at least one outstanding continuous query.

We define when a query Q **shows interest** in an object P as follows:

Definition 3 A continuous query Q is interested in object P if P either lies in Q's spatial area or in Q's cache area.

Having the previous definition of *significant* objects, SOLE continuously maintains the following assertion:

Assertion 1 *Only significant objects are stored in the shared memory buffer*

To always satisfy this assertion, SOLE continuously keeps track of the following: (1) A newly incoming data object P is stored in memory only if P is *significant*, (2) At any time, if an object P that is already stored

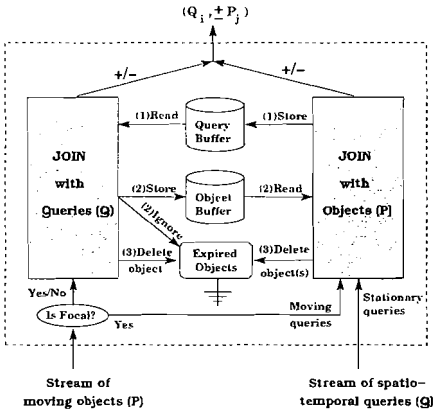


Fig. 5 Shared join operator in SOLE.

in the shared buffer becomes *insignificant*, we drop P immediately from the shared buffer.

Significant moving objects are hashed to grid cells based on their spatial locations. An entry of a *significant* moving object P in a grid cell C has the form $(PID, Location, RefCount, FocalList)$. PID and $Location$ are the object identifer and location, respectively. $RefCount$ indicates the number of queries that are interested in P. $FocalList$ is the list of *active* moving queries that have P as their *focal* object. Unlike data objects that are stored in only one grid cell, continuous queries are stored in all grid cells that overlap either the query spatial area or the query cache area. A query entry in a grid cell contains only the query identifier (QID). The spatial region for each query is stored separately in a global lookup table.

5.3 Shared Spatio-temporal Join Operator

Overview. Figure 5 puts a magnifying glass over the shared spatial join operator in Figure 4b. For any incoming data object, say P, the shared spatial join operator consults its query buffer to check if any query is affected by P (either in a positive or a negative way). Based on the result, we decide either to store P in the object buffer or to ignore P and delete P's old location (if any) from the object buffer. On the other hand, for any incoming continuous query, say Q, first we store Q or update Q's old location (if any) in the query buffer. Then, we consult the object buffer to check if any of the objects needs to be added to or removed from Q's answer. Based on this operation, some in-memory stored objects may become *insignificant*, hence, are deleted immediately from the object buffer. Stationary queries are submitted directly to the shared spatial join operator, while moving queries are generated from the movement of their focal objects.

Algorithm. Based on the data stored in the shared buffer, SOLE distinguishes among four types of data inputs: (1) A new data object P that is not stored in memory, (2) Update of the location of object P, (3) A new stationary query Q, (4) An update of the region of a moving query Q. Figures 6, 7, 9, and 10 give the pseudo code of SOLE upon receiving each input type. The details of the algorithms are described below. SOLE makes use of the following notations: $\hat{Q}$ indicates the extended query region that covers the cache area so that $Q \subset \hat{Q}$. C_Q, $\hat{C_Q}$ are the set of grid cells that are covered by Q and $\hat{Q}$, respectively. C_P represents a single grid cell that covers the object P.

Procedure IncomingNewObject(`Object` P, `GridCell` C_P)
Begin

1. *For each Query* $Q_i \in C_P$ *AND* $P \in \hat{Q}_i$
   (a) $P.RefCount{+}{+}$
   (b) `if` *(*$P \in Q_i$*)* `then` *output* $(Q_i, +P)$.
2. `if` *(*$P.RefCount$*)* `then` *store* P *in* C_P *and in hash table* h.

End.

Fig. 6 Pseudo code for receiving a new value of P.

Procedure UpdateObj(`Object` P_{old},P, `GridCell` $C_{P_{old}}$,C_P)
Begin

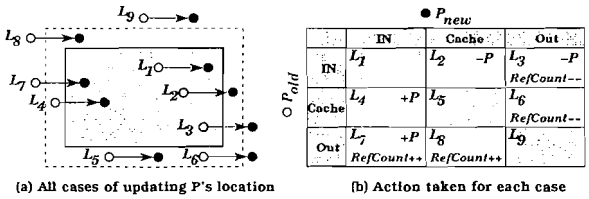
1. *For each query* $Q_i \in P.FocalList$, `UpdateQuery`(Q_i)
2. Let L be the line $(P_{old}.P)$
3. *For each query* $Q_i \in (C_{P_{old}} \cup C_P)$
   (a) `if` Q_i *intersects* L, `then`
      - `if` $P \in Q_i$ `then` *Output* $(Q_i, +P)$; `if` $P_{old} \notin \hat{Q}_i$; $P.RefCount{+}{+}$
      - `else` *Output* $(Q_i, -P)$; `if` $P \notin \hat{Q}_i$ `then` $P.RefCount{-}{-}$
   (b) `else if` $\hat{Q}_i$ *intersects* L
      - `if` $P \in \hat{Q}_i$ `then` $P.RefCount{+}{+}$; `else` $P.RefCount{-}{-}$
4. `if` *(!*$P.RefCount$*)* `then` *delete* P_{old} *and ignore* P; `return`.
5. `if` $(C_{P_{old}} \neq C_P)$ `then` *move* P_{old} *from* $C_{P_{old}}$ *to* C_P.
6. *Update the location of* P_{old} *to that of* P *in* C_P.

End.

Fig. 7 Pseudo code for updating P's location.

Input Type I: A new object P. Figure 6 gives the pseudo code of SOLE upon receiving a new object P in the grid cell C_P (i.e., P is not stored in memory). P is tested against all the queries that are stored in C_P (Step 1 in Figure 6). For each query $Q_i \in C_P$, only three cases can take place: (1) P lies in $\hat{Q}_i$ but not in Q. In this case, we need only to increase the reference counter of P to indicate that there is one more query interested in P (Step 1a in Figure 6). Notice that no output is produced in this case since P does not satisfy Q_i. (2) P satisfies Q_i. In this case, in addition to increasing the reference counter, we output a *positive* update that indicates the addition of P to the answer set of Q_i (Step 1b in Figure 6). In the above two cases, P is stored in the shared buffer as it is considered *significant*. (3) P neither satisfies Q_i nor lies in $\hat{Q}_i$. Thus, P is simply ignored as it is *insignificant*.

Input Type II: An update of P. Figure 7 gives the pseudo code of SOLE upon receiving an update of ob-



(a) All cases of updating P's location (b) Action taken for each case

Fig. 8 All cases of updating P's location.

Procedure StationaryQuery(Query Q) **Begin**

- *For each grid cell* $c_j \in \hat{C}_Q$
  1. *Register* Q *in* c_j
  2. *For each object* $P_i \in c_j$ *AND* $P_i \in \hat{Q}$
     - $P.RefCount++$; if $P \in Q$ then *output* $(Q, +P)$

End.

Fig. 9 Pseudo code for receiving a new query Q.

ject P's location. The old location of P is retrieved from the hash table h. First, we evaluate all moving queries (if any) that have P as their *focal* object (Step 1 in Figure 7). Then, we check all the queries that belong to either C_P or $C_{P_{old}}$ (Step 3 in Figure 7) against the line L that connects P and P_{old}. Figure 8a gives nine different cases for the intersection of L with Q where P_{old} and P are plotted as white and black circles, respectively. Both P_{old} and P can be in one of the three states, *in*, *cache*, or *out* that indicates that P satisfies Q, in the cache area of Q, or does not satisfy Q, respectively. The action taken for each case is given in Figure 8b. Basically, if there is no change of state from P_{old} to P (e.g., L_1, L_5, and L_9), no action will be taken. If P_{old} was in Q, however, P is not, (e.g., L_2 and L_3) we output the negative update $(Q, -P)$. The reference counter is decreased only when P_{old} is of interest to Q while P is not (e.g., L_3 and L_6). Notice that in the case of L_2, we do not need to decrease the reference counter where although P does not satisfy Q, P is still of interest to Q as P lies in $\hat{Q}_i$. Also, in the case of L_6, we do not need to output a *negative* update, however we decrease the reference counter. In this case, since P and P_{old} are not in the answer set of Q, there is no need to update the answer. Similarly, with a symmetric behavior, we output a *positive* update in the cases of L_4 and L_7 and we increment the reference counter in the cases of L_7 and L_8. After testing all cases, we check whether object P becomes *insignificant*. If this is the case, we immediately drop P from memory (Step 4 in Figure 7). If P is still *significant*, we update P's location and cell (if needed) in the grid structure (Steps 5 and 6 in Figure 7).

Input Type III: A new query Q. Figure 9 gives the pseudo code of SOLE upon receiving a continuous stationary query Q. Basically, we register Q in all the grid cells that are covered by $\hat{Q}$. In addition, we test Q against all data objects that are stored in these cells. We increase the reference counter of only those objects that

Procedure UpdateQuery(Query Q_{old}, Q) **Begin**

- *For each object* $P_i \in (\hat{C}_{Q_{old}} \cap \hat{C}_Q)$
  1. if $P_i \in Q_{old}$ then
     - if $P_i \notin Q$ then *(Output* $(Q, -P_i)$; if $P_i \notin \hat{Q}$ then $(P_i.RefCount--$; if $(!P_i.RefCount)$ then delete$(P_i)))$
  2. else if $P_i \in Q$ then *(Output* $(Q, +P_i)$; if $P_i \notin \hat{Q}_{old}$ then $P_i.RefCount++)$
  3. else if $P_i \in \hat{Q}_{old}$ *AND* $P_i \notin \hat{Q}$ then $(P_i.RefCount--$; if $(!P_i.RefCount)$ then delete$(P_i))$
  4. else if $P_i \in \hat{Q}$ *AND* $P_i \notin Q\hat{o}ld$ then $P_i.RefCount++$.
- *Register* Q *in* $\hat{C}_Q - \hat{C}_{Q_{old}}$; *unregister* Q *from* $\hat{C}_{Q_{old}} - \hat{C}_Q$

End.

Fig. 10 Pseudo code for updating a query.

lie in $\hat{Q}$. In addition, objects that satisfy Q results in producing *positive* updates.

Input Type IV: An update of Q's region. Figure 10 gives the pseudo code of SOLE upon receiving an update of a moving query region. All stored objects in all cells that are covered by the old and new regions of Q are tested against Q. Figure 11a divides the space covered by the old and new regions of Q into seven regions (R_1-R_7). The action taken for any point that lies in any of these regions is given in Figure 11b. Similar to Figure 8b, a region R_i could have any of the three states *in*, *cache*, or *out* based on whether R_i is inside Q, is in the cache area of Q, or is outside Q. Basically, no action is taken for objects in any region R_i that maintains its state for both Q and Q_{old} (e.g., R_4). If a region R_i is inside Q_{old}, but is not in Q, (e.g., R_2 and R_3), we output a *negative* update for each object in R_i. We decrement the reference counter of these objects only if they lie in the region that is out of the new cache area (e.g., R_2) (Step 1 in Figure 10). Also, the reference counter is decremented for all objects in the region that are in the old cache area but are out of the new cache area (e.g., R_1) (Step 3 in Figure 10). Similarly, the reference counter is increased for regions R_6 and R_7 while a *positive* output is sent for the points in regions R_5 and R_6. Notice that whenever we decrement the reference counter for any moving object P, we check whether P becomes *insignificant*. If this is the case, we immediately drop P from memory (e.g., Steps 1 and 3 in Figure 11). Finally, Q is registered in all the new cells that are covered by the new region and not the old region. Similarly, Q is unregistered from all cells that are covered by the old region and not the new region.

6 Approximate Query Processing in SOLE

Even with the scalability features of SOLE, the memory resource may be exhausted at intervals of unexpected massive numbers of queries and moving objects (e.g.,

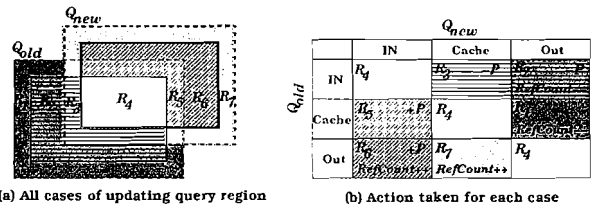


Fig. 11 All cases of updating Q's region.

during rush hours). To cope with such intervals, SOLE is equipped with a *self-tuning* approach that tunes the memory load to support a large number of concurrent queries, yet with an approximate answer. The main idea is to tune the definition of *significant* objects based on the current workload. By adapting the definition of *significant* objects, the memory load will be *shed* in two ways: (1) In-memory stored objects will be revisited for the new meaning of *significant* objects. If an *insignificant* object is found, it will be *shed* from memory. (2) Some of the newly input data will be *shed* at the input level.

Figure 12 gives the architecture of *self-tuning* in SOLE. Once the shared join operator incurs high resource consumption, e.g., the memory becomes almost full, the join operator triggers the execution of the *load shedding* procedure. The *load shedding* procedure may consult some statistics that are collected during the course of execution to decide on a new meaning of *significant* objects. While the shared join operator is running with the new definition of *significant* objects, it may send updates of the current memory load to the *load shedding* procedure. The load shedding procedure replies back by continuously adopting the notion of *significant* objects based on the continuously changing memory load. Finally, once the memory load returns to a stable state, the shared join operator retains the original meaning of *significant* objects and stops the execution of the *load shedding* procedure. Solid lines in Figure 12 indicate the mandatory steps that should be taken by any *load shedding* technique. Dashed lines indicate a set of operations that may or may not be employed based on the underlying *load shedding* technique. In the rest of this section, we propose two *load shedding* techniques, namely *query load shedding* and *object load shedding*.

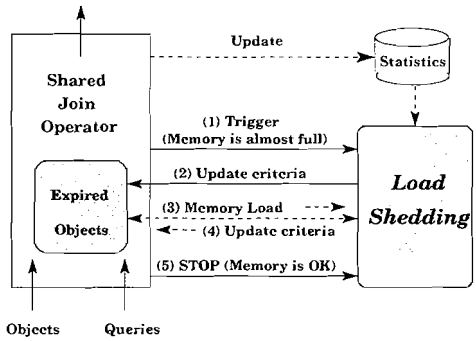


Fig. 12 Architecture of *self tuning* in SOLE.

6.1 Query Load Shedding

The main idea of *query load shedding* is to *negotiate* the query region with the user. Whenever a query, say Q, is submitted to SOLE, Q specifies the minimum accuracy that is acceptable by Q. Initially, the submitted query Q is evaluated with complete accuracy. However, when the system is overloaded, Q's accuracy is degraded to its minimum permissible accuracy. Reducing the accuracy is achieved by shrinking Q's cache area from all directions to have a smaller cache area. After we are done with all the cache area, if the system is still overloaded, and we have not reached to the minimum permissable accuracy yet, we start to reduce Q's area itself. Thus, the notion of *significant* objects is adopted to be those tuples that lie in the *reduced* query area of at least one continuous outstanding query. By reducing the query sizes of all outstanding queries, objects that are outside of the reduced area and are not of interest to any other query are **immediately** dropped from memory and the corresponding *negative* updates are sent. During the course of execution, we gradually increase the query size to cope with the memory load. Finally, when the system reaches a stable state, we retain the original query sizes.

Query load shedding has two main advantages: (1) It is intuitive and simple to implement where there is no need to maintain any kind of statistical information, and (2) *Insignificant* objects are immediately dropped from memory. On the other side, there are two main disadvantages: (1) The query load shedding process is expensive, where it scans all stored objects and queries. This exhaustive behavior results in pause time intervals where the system cannot produce output nor process data inputs. (2) Although the query accuracy is guaranteed (assuming uniform data distribution), there is no guarantee of the amount of reduced memory. Assume the case that the reduced area from a query Q_i lies completely inside another query Q_j. Thus, even though Q_i is reduced, we cannot drop tuples from the reduced area where they are still needed by Q_j. Thus, the accuracy of Q_i is reduced, yet the amount of memory is not.

6.2 Object Load Shedding

The main idea of *object load shedding* is to drop objects that have less effect on the average query accuracy. Thus, the definition of *significant* objects is adopted to be those objects that are of interest to at least k queries (i.e., objects with reference counter greater than or equal k). Notice that the original definition of *significant* objects implicitly assumes that $k = 1$. A key point in *object* load shedding is that we do not perform an exhaustive scan to drop *insignificant* objects. Instead, *insignificant*

objects are **lazily** dropped whenever they get accessed later during the course of execution. Such *lazy* behavior completely avoids the pause time intervals in *query* load shedding. In contrast to *query* load shedding, in *object* load shedding, we guarantee the reduced memory load.

During the course of execution, we monitor the memory load and decrease/increase k accordingly. Once the system stabilizes and returns to its original state, we set $k = 1$ to retain the original execution of SOLE. Determining the threshold value k is achieved by maintaining a statistical table S that keeps track of the number of objects that satisfy a certain number of queries. Assuming that we will never drop an object that has a reference counter greater than N, then S can be represented as an array of N numbers where the jth entry in S corresponds to the number of moving objects that are of interest to j queries. Whenever the system is overloaded, we go through S to get the minimum k that achieves the required reduced load.

6.3 Load Shedding with Locking

Degenerate cases may affect severely the behavior of load shedding. Consider the case of a query Q that has only one object P as its answer while P is not of interest to any other query. By applying *object* load shedding, P will be dropped where it is of interest to only one query Q. Thus, the accuracy of Q is dropped to zero. To alleviate such problem, we use a *locking* technique. Basically, each query Q has a threshold n where if Q has less than n objects in its answer set, all the n objects are *locked*. *Locked* objects do not participate in the statistical table S. Once an object is *locked*, the corresponding entry in S is updated. Whenever we *lazily* drop objects from memory, we make sure that we do not drop any *locked* object. The concept of *locking* can also be generalized to accommodate locking of important objects and/or queries.

7 Experimental Results

In this section, we study the performance of various aspects of SOLE that includes: the size of the cache area, the benefit of encapsulating SOLE in a pipeline operator, the grid size of the shared memory buffer, the scalability of SOLE, and approximate query processing via load shedding techniques. All the experiments in this section are based on a real implementation of SOLE algorithms and operators inside our prototype database engine for spatio-temporal streams, PLACE [30,31]. We run PLACE on Intel Pentium IV CPU 2.4GHz with 512MB RAM running Windows XP. Without loss of generality, all the presented experiments are conducted on stationary and moving continuous spatio-temporal queries. Similar results are achieved when employing continuous k-nearest-neighbor queries.

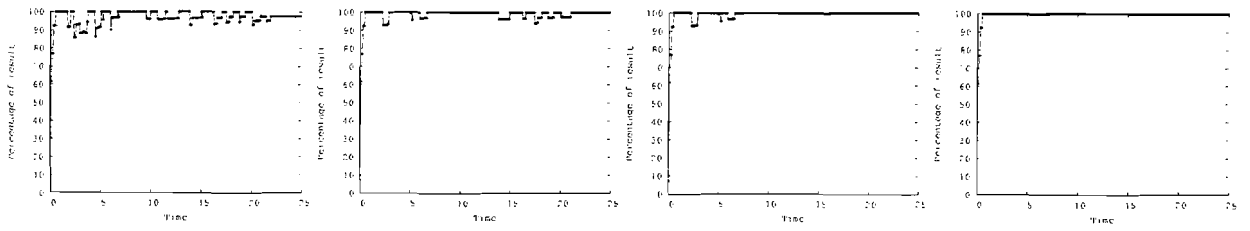
Fig. 13 Greater Lafayette, Indiana, USA.

We use the *Network-based Generator of Moving Objects* [7] to generate a set of moving objects and moving queries in the form of spatio-temporal streams. The input to the generator is the road map of the Greater Lafayatte (a city in the state of Indiana, USA) given in Figure 13. The output of the generator is a set of moving points that move on the road network of the given city. Moving objects can be cars, cyclists, pedestrians, etc. Any moving object can be a *focal* of a moving query. Unless mentioned otherwise, we generate 110K moving objects as follows: Initially, we generate 10K moving objects from the generator, then we run the generator for 1000 time units. At each time unit, we generate new 100 moving objects. Moving objects are required to report their locations every time unit T. Failure to do so results in disconnecting the moving object from the server.

The rest of this section is organized as follows. Section 7.1 studies the effect of the cache size and the gain of having SOLE as a pipelined operator in terms of single query execution. In Section 7.3, we study the scalability of SOLE. Finally, Section 7.6 studies the performance of load shedding techniques.

7.1 Single Execution: Size of the Cache Area

Figures 14a-d give the performance of the first 25 seconds of executing a moving query of size 0.5% of the space with no cache, 25% cache, 50% cache, and *conservative* cache (i.e., 100% cache), respectively. Our performance measure is the query accuracy that is represented as the percentage of the number of produced tuples to the actual number that should have been produced if all moving objects are materialized into secondary storage. Without caching (Figure 14a), the query accuracy suffers from continuous fluctuations where sometimes the accuracy drops to 85%. With only 25% cache the query accuracy is greatly enhanced (Figure 14b). The accuracy is almost stable with minor fluctuations that degrade the accuracy to only 95%. A *conservative* caching would result in having a single line that always have 100% accuracy.



(a) No Cache (b) 25% Cache (c) 50% Cache (d) Conservative Cache

Fig. 14 Cache area in SOLE.

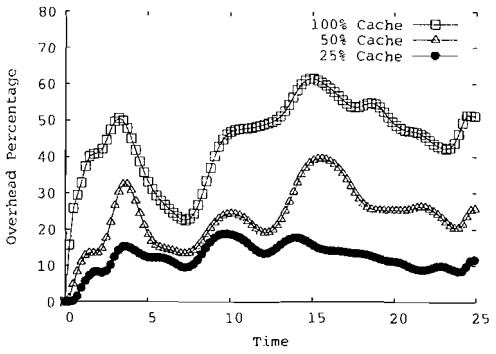


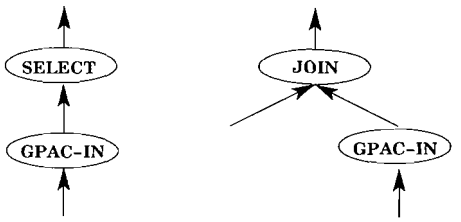
Fig. 15 Cache area in SOLE.

Figure 15 gives the memory overhead when using a 25%, 50%, or 100% (*conservative*) cache sizes. The overhead is computed as a percentage from the original query memory requirements. Thus a 0% cache does not incur any overhead. On average a 25% cache results in only 10% overhead over the original query, while the 50% and 100% caches result in 25% and 50% overhead, respectively. As a compromise between the cache overhead and the query accuracy, we use a 25% cache in SOLE in all the following experiments.

7.2 Single Execution: Pipelined Query Operators

Consider the query Q: *"Continuously report all trucks that are within MyArea"*. *MyArea* can be either a stationary or moving range query. A high level implementation of this query is to have only a selection operator that selects only the *"trucks"*. Then, a high level algorithm implementation would take the selection output and incrementally produce the query result. However, an encapsulation of SOLE into a physical pipelined query operator allows for more flexible plans. Figure 16a gives a query evaluation plan when pushing the SOLE operator before the *selection* operator. The following is the SQL presentation of the query.

```
SELECT M.ObjectID
FROM MovingObjects M
WHERE M.type = "truck"
INSIDE MyArea
```



(a) SELECTION **(b) JOIN**

Fig. 16 Pipelined SOLE operators.

Figure 17 compares the high level implementation of the above query with pipelined INSIDE operator for both stationary and moving queries. The selectivity of the queries varies from 2% to 64%. The selectivity of the selection operator is 5%. Our measure of comparison is the number of tuples that go through the query evaluation pipeline. When SOLE is implemented at the application level, its performance is not affected by the query selectivity. However, when INSIDE is pushed before the *selection*, it acts as a filter for the query evaluation pipeline, thus, limiting the tuples through the pipeline to only the progressive updates. With INSIDE selectivity less than 32%, pushing INSIDE before the selection greatly affects the performance. However, with selectivity more than 32%, it would be better to have the INSIDE operator above the *selection* operator.

Consider a more complex query plan that contains a *join* operator. The query Q: *"Continuously report moving objects that belong to my favorite set of objects and that lie within MyArea"*. A high level implementation of SOLE would probe a streaming database engine to join all moving objects with my favorite set of objects. Then, the output of the join is sent to the SOLE algorithm for further processing. However, with the INSIDE operator, we can have a query evaluation plan as that of Figure 16b where the INSIDE operator is pushed below the *Join* op-

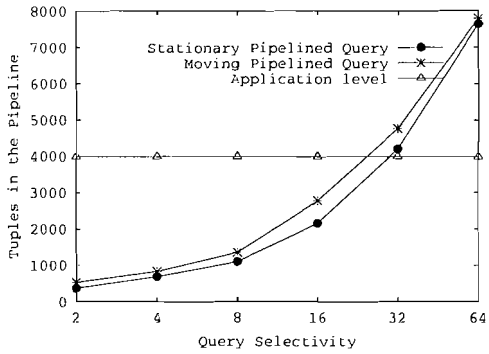


Fig. 17 Pipelined operators with SELECT.

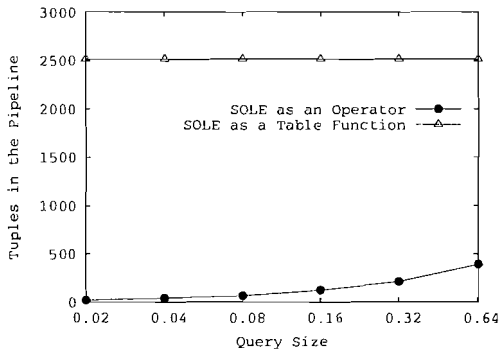
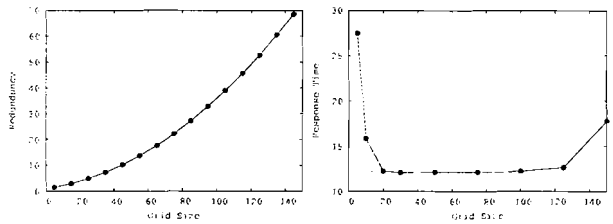


Fig. 18 Pipelined operators with Join.

erator. The SQL representation of the above query is as follows:

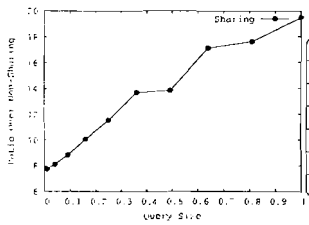
```
SELECT M.ObjectID
FROM MovingObjects M, MyFavoriteCars F
WHERE M.ObjectID = F.ObjectID
INSIDE MyArea
```

Figure 18 compares the high level implementation of the above query with the pipelined INSIDE operator for both stationary and moving queries. The selectivity of the queries varies from 2% to 64%. As in Figure 17, the selectivity of SOLE does not affect the performance if it is implemented in the application level. Unlike the case of *selection* operators, SOLE provides a dramatic increase in the performance (around an order of magnitude) when implemented as a pipelined operator. The main reason in this dramatic gain in performance is the high overhead incurred when evaluating the *join* operation. Thus, the INSIDE operator filters out the input tuples and limit the input to the join operator to only the incremental *positive* and *negative* updates.



(a) Redundancy (b) Response Time

Fig. 19 Grid Size.



%	Single	Sharing	Ratio
0.01	*8184*	*63446*	*7.75*
0.09	*934*	*8250*	*8.83*
0.25	*349*	*4016*	*11.51*
0.49	*186*	*2577*	*13.85*
0.81	*118*	*2082*	*17.64*
1	*103*	*2007*	*19.49*

(a) Ratio (b) Table of values

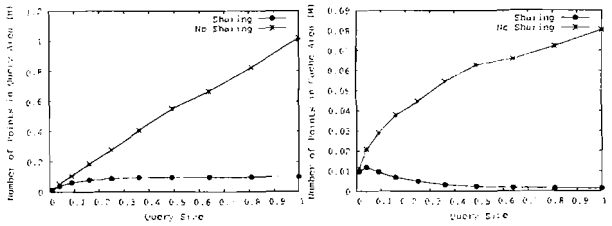
Fig. 20 Maximum Number of Supported Queries.

7.3 Scalable Execution: Grid Size

Figure 19 studies the trade-offs for the number of grid cells in the shared memory buffer of SOLE for 50K moving queries of various sizes. Increasing the number of cells in each dimension increases the redundancy that results from replicating the query entry in all overlapping grid cells. On the other hand, increasing the grid size results in a better response time. The response time is defined as the time interval from the arrival of an object, say P, to either the time that P appears at the output of SOLE or the time that SOLE decides to discard P. When the grid size increases over 100, the response time performance degrades. Having a grid of 100 cells in each dimension results in a total of 10K small-sized grid cells, thus, with each movement of a moving query Q, we need to register/unregister Q in a large number of grid cells. As a compromise between redundancy and response time, SOLE uses a grid of size 30 in each dimension.

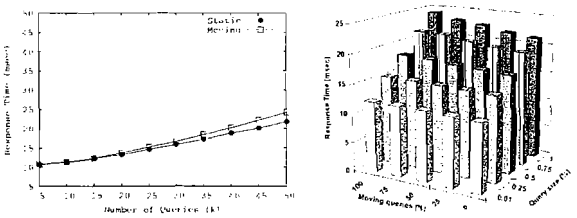
7.4 Scalable Execution: SOLE Vs. Non-Shared Execution

Figure 20 compares the performance of the SOLE shared operator as opposed to dealing with each query as a separate entity (i.e., with no sharing). Figure 20a gives the ratio of the number of supported queries via sharing over the non-sharing case for various query sizes. Some of the



(a) Query area (b) Cache area

Fig. 21 Data size in the query and cache areas.



(a) Number of queries (b) Query size & Percent of moving queries

Fig. 22 Response time in SOLE.

actual values are depicted in the table in Figure 20b. For small query sizes (e.g., 0.01%) with sharing. SOLE supports more than 60K queries, which is almost 8 times better than the case of non-sharing. The performance of sharing increases with the query size where it becomes 20 times better than non-sharing in case of query size 1% of the space. The main reason of the increasing performance with the size increase is that sharing benefits from the overlapped areas of continuous queries. Objects that lie in any overlapped area are stored only once in the sharing case rather than multiple times in the non-sharing case. With small query sizes, overlapping of query areas is much less than the case of large query sizes.

Figures 21a and 21b give the memory requirements for storing objects in the query region and the query cache area, respectively, for 1K queries over 100K moving objects. In Figure 21a, for large query sizes (e.g., 1% of the space), a non-shared execution would need a memory of size 1M objects, while in SOLE, we need, at most, a memory of size 100K objects. The main reason is that with non-sharing, objects that are needed by multiple queries are redundantly stored in each query buffer, while with sharing, each object is stored at most once in the shared memory buffer. Thus, in terms of the query area, SOLE has a ten times performance advantage over the non-shared case. Figure 21b gives the memory requirement for storing objects in the cache area. The behavior of the non-sharing case is expected where the memory requirements increase with the increase in the query size. Surprisingly, the caching overhead in the case of sharing decreases with the increase in the query size. The main reason is that with the size increase, the caching area of a certain query is likely to be part of the actual area of another query. Thus, objects that are inside this caching area are not considered an overhead, where they are part of the actual answer of some other query.

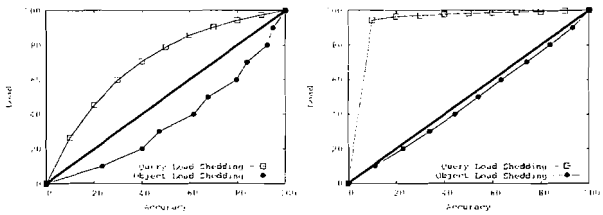
7.5 Scalable Execution: Response Time

Figure 22a gives the effect of the number of concurrent continuous queries on the performance of SOLE. The number of queries varies from 5K to 50K. Our performance measure is the average response time. The response time is defined as the time interval from the arrival of object P to either the time that P appears at the output of SOLE or the time that SOLE decides to discard P. We run the experiment twice; once with only stationary queries, and the second time with only moving queries. The increase in response time with the number of queries is acceptable since as we increase the number of queries 10 times (from 5K to 50K), we get only twice the increase in response time in the case of stationary queries (from 11 to 22 msec). The performance of moving queries has only a slight increase over stationary queries (2 msec in case of 50K queries).

Figure 22b gives the effect of varying both the query size and the percentage of moving queries on the response time of the SOLE operator. The number of outstanding queries is fixed to 30K. The response time increases with the increase in both the query size and the percentage of moving queries. However, the SOLE operator is less sensitive to the percentage of moving queries than to the query size. Increasing the percentage of moving queries results in a slight increase in response time. This performance indicates that SOLE can efficiently deal with moving queries in the same performance as with stationary queries. On the other hand, increasing the query size from 0.01% to 1% only doubles the response time (from around 12 msec to around 24 msec) for various moving percentages.

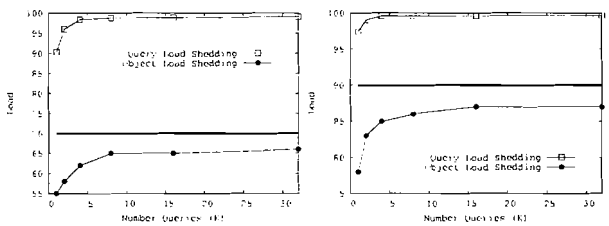
7.6 Load Shedding: Accuracy in Query Answer

Figures 23a and 23b compare the performance of *query* and *object* load shedding techniques for processing 1K and 25K queries with various sizes, respectively. Our performance measure is the reduced load to achieve a ceratin query accuracy. When the system is overloaded, we vary the required accuracy from 0% to 100%. In degenerate cases, setting the accuracy to 100% requires keeping the whole memory load (100% load) while setting the accuracy to 0% requires deleting all memory load. The bold diagonal line in Figure 23 represents the required accuracy. It is "expected" that if we ask for m% accuracy, we



(a) 1K Queries (b) 25K Queries

Fig. 23 Load Vs. Accuracy.



(a) 70% Accuracy (b) 90% Accuracy

Fig. 24 Reduced load for a certain accuracy.

will need to keep only $m\%$ of the memory load. Thus, reducing the memory load to be lower than the diagonal line is considered a gain over the "expected" behavior. The *object* load shedding always maintains better performance than that of the *query* load shedding. For example, in the case of 1K queries, to achieve an average accuracy of 90%, we need to keep track of only 85% of the memory load in the case of *object* load shedding while 97% of the memory is needed in the case of *query* load shedding. The performance of both load shedding techniques is worse with the increase in the number of queries to 25K. However, the *object* load shedding still keeps a good performance where it is almost equal to the "expected" performance. The performance of *query* load shedding is dramatically degraded where we need more than 90% of the memory load to achieve only 20% accuracy.

Figures 24a and 24b compare the performance of *query* and *object* load shedding to achieve an accuracy of 70% and 90%, while varying the number of queries from 2K to 32K. The *object* load shedding greatly outperforms the *query* load shedding and results in a better performance than the "expected" reduced load for all query sizes. The main reason behind the bad performance of *query* load shedding is that in the case of a large number of queries, there are high overlapping areas. Thus, the reduced area of a certain query is highly likely to overlap other queries. So, even though we reduce the query area, we cannot drop any of the tuples that lie in the reduced area. Such tuples are still of interest to other outstanding queries.

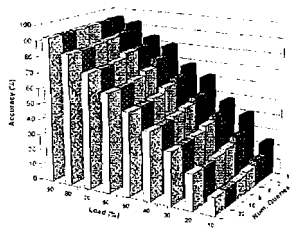


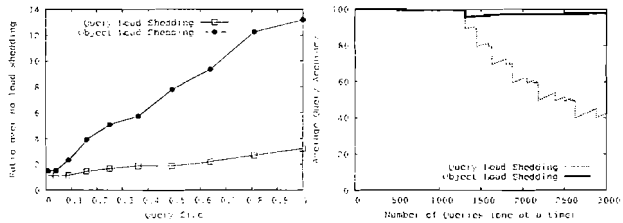
Fig. 25 Performance of Object Load Shedding.

Figure 25 focuses on the performance of *object* load shedding. The required reduced load varies from 10% to 90% while the number of queries varies from 1K to 32K. This experiment shows that *object* load shedding is scalable and is stable when increasing the number of queries. For example, when reducing the memory load to 90%, we consistently get an accuracy around 94% regardless of the number of queries. Such consistent behavior appears in various reduced loads.

7.7 Load Shedding: Scalability with Load Shedding

Figure 26a gives the ratio of the number of supported queries with *query* and *object* load shedding techniques over the sharing case with no load shedding. All queries are supported with a minimum accuracy of 90%. Depending on the query size, *query* load shedding can support up to 3 times more queries than the case with no load shedding. This indicates a ratio of up to 60 times better than the non-sharing cases (refer to the table in Figure 20b). On the other hand, *object* load shedding has much better scalable performance than that of *query* load shedding. With *object* load shedding SOLE can have up to 13 times more queries than the case of no load shedding, which indicates up to 260 times than the case of no sharing.

Figure 26b gives the performance of the *query* and *object* load shedding techniques in terms of maintaining the average query accuracy with the arrival of continuous queries. The horizontal access advances with time to represent the arrival of each continuous query. With tight memory resources, the memory is consumed completely with the arrival of about 1200 queries. At this point, the process of *load shedding* is triggered. The required memory consumption level is set to 90%. Since *query* load shedding immediately drops tuples from memory, the query accuracy is dropped sharply to 90%. In contrast, in *object* load shedding, the accuracy degrades slowly. With the arrival of more queries, *query* load shedding tries to slowly enhance its performance. However, the memory consumption is faster than the recovery of *query* load shedding. Thus, soon, we will need to drop some more tuples from memory that will result in less accuracy.



(a) Ratio (b) Query Arrival

Fig. 26 Scalability with Load Shedding.

The behavior continues with two contradicting actions: (1) *Query* load shedding tends to enhance the accuracy by retaining the original query size, and (2) The arrival of more queries consumes memory resources. Since the second action is faster than the first one, the performance has a zigzag behavior that leads to reducing the query accuracy. On the other hand, *object* load shedding does not suffer from this drawback. Instead, due to the smartness of choosing victim objects, *object* load shedding always maintains sufficient accuracy with minimum memory load.

8 Conclusion

We presented the *Scalable On-Line Execution* algorithm (SOLE, for short) for continuous and on-line evaluation of concurrent continuous spatio-temporal queries over spatio-temporal data streams. SOLE is an in-memory algorithm that utilizes the scarce memory resources efficiently by keeping track of only those objects that are considered *significant*. SOLE is a unified framework for stationary and moving queries that is encapsulated into a physical pipelined query operator. To cope with intervals of high arrival rates of objects and/or queries, SOLE utilizes *load shedding* techniques that aim to support more continuous queries, yet with an approximate answer. Two load shedding techniques were proposed, namely, *query* load shedding and *object* load shedding. Experimental results based on a real implementation of SOLE inside a prototype data stream management system show that SOLE can support up to 20 times more continuous queries than the case of dealing with each query separately. With *object* load shedding, SOLE can support up to 260 times more queries than the case of no sharing.

References

1. Abadi, D., Ahmad, Y., Balakrishnan, H., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Janotti, J., Lindner, W., Madden, S., Rasin, A., Stonebraker, M., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: Proceedings of the International Conference on Innovative Data Systems Research, CIDR (2005)
2. Abadi, D.J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Aurora: A New Model and Architecture for Data Stream Management. VLDB Journal **12**(2) (2003)
3. Arasu, A., Widom, J.: Resource Sharing in Continuous Sliding-Window Aggregates. In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2004)
4. Ayad, A., Naughton, J.F.: Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD (2004)
5. Babcock, B., Datar, M., Motwani, R.: Load Shedding for Aggregation Queries over Data Streams. In: Proceedings of the International Conference on Data Engineering, ICDE (2004)
6. Babu, S., Widom, J.: Continuous Queries over Data Streams. SIGMOD Record **30**(3) (2001)
7. Brinkhoff, T.: A Framework for Generating Network-Based Moving Objects. GeoInformatica **6**(2) (2002)
8. Cai, Y., Hua, K.A., Cao, G.: Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. In: Proceedings of the International Conference on Mobile Data Management, MDM (2004)
9. Chakka, V.P., Everspaugh, A., Patel, J.M.: Indexing Large Trajectory Data Sets with SETI. In: Proceedings of the International Conference on Innovative Data Systems Research, CIDR (2003)
10. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.A.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: Proceedings of the International Conference on Innovative Data Systems Research, CIDR (2003)
11. Chandrasekaran, S., Franklin, M.J.: PSoup: a system for streaming queries over streaming data. VLDB Journal **12**(2), 140–156 (2003)
12. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD (2000)
13. Dobra, A., Garofalakis, M.N., Gehrke, J., Rastogi, R.: Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In: Proceedings of the International Conference on Extending Database Technology, EDBT (2004)
14. Gedik, B., Liu, L.: MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In: Proceedings of the International Conference on Extending Database Technology, EDBT (2004)
15. Golab, L., Ozsu, M.T.: Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2003)
16. Hammad, M.A., Franklin, M.J., Aref, W.G., Elmagarmid, A.K.: Scheduling for shared window joins over data streams. In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2003)
17. Hammad, M.A., Ghanem, T.M., Aref, W.G., Elmagarmid, A.K., Mokbel, M.F.: Efficient pipelined execution of sliding-window queries over data streams. Tech. Rep. TR CSD-03-035, Purdue University Department of Computer Sciences (2003)
18. Hammad, M.A., Mokbel, M.F., Ali, M.H., Aref, W.G., Catlin, A.C., Elmagarmid, A.K., Eltabakh, M., Elfeky,

M.G., Ghanem, T.M., Gwadera, R., Ilyas, I.F., Marzouk, M., Xiong, X.: Nile: A Query Processing Engine for Data Streams (Demo). In: Proceedings of the International Conference on Data Engineering, ICDE (2004)
19. http://www.fcc.gov/911/enhanced/:
20. Hu, H., Xu, J., Lee, D.L.: A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD. Baltimore, MD (2005)
21. Iwerks, G.S., Samet, H., Smith, K.: Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2003)
22. Jensen, C.S., Lin, D., Ooi, B.C.: Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2004)
23. Kwon, D., Lee, S., Lee, S.: Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In: Proceedings of the International Conference on Mobile Data Management, MDM (2002)
24. Lazaridis, I., Porkaew, K., Mehrotra, S.: Dynamic Queries over Mobile Objects. In: Proceedings of the International Conference on Extending Database Technology, EDBT (2002)
25. Lee, M.L., Hsu, W., Jensen, C.S., Teo, K.L.: Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2003)
26. Madden, S., Shah, M., Hellerstein, J.M., Raman, V.: Continuously adaptive continuous queries over streams. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD (2002)
27. Mokbel, M.F., Aref, W.G.: GPAC: Generic and Progressive Processing of Mobile Queries over Mobile Data. In: Proceedings of the International Conference on Mobile Data Management, MDM (2005)
28. Mokbel, M.F., Aref, W.G., Hambrusch, S.E., Prabhakar, S.: Towards Scalable Location-aware Services: Requirements and Research Issues. In: Proceedings of the ACM Symposium on Advances in Geographic Information Systems, ACM GIS (2003)
29. Mokbel, M.F., Xiong, X., Aref, W.G.: SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD (2004)
30. Mokbel, M.F., Xiong, X., Aref, W.G., Hambrusch, S., Prabhakar, S., Hammad, M.: PLACE: A Query Processor for Handling Real-time Spatio-temporal Data Streams (Demo). In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2004)
31. Mokbel, M.F., Xiong, X., Hammad, M.A., Aref, W.G.: Continuous Query Processing of Spatio-temporal Data Streams in PLACE. In: Proceedings of the second workshop on Spatio-Temporal Database Management, STDBM (2004)
32. Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G.S., Olston, C., Rosenstein, J., Varma, R.: Query Processing, Approximation, and Resource Management in a Data Stream Management System. In: Proceedings of the International Conference on Innovative Data Systems Research, CIDR (2003)
33. Mouratidis, K., Papadias, D., Hadjieleftheriou, M.: Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD. Baltimore, MD (2005)
34. Nadeem, T., Dashtinezhad, S., Liao, C., Iftode, L.: TrafficView: A Scalable Traffic Monitoring System. In: Proceedings of the International Conference on Mobile Data Management, MDM, pp. 13–26. Berkeley, CA (2004)
35. Patel, J.M., Chen, Y., Chakka, V.P.: STRIPES: An Efficient Index for Predicted Trajectories. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD (2004)
36. Prabhakar, S., Xia, Y., Kalashnikov, D.V., Aref, W.G., Hambrusch, S.E.: Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. IEEE Trans. on Computers **51**(10) (2002)
37. Reinwald, B., Pirahesh, H.: SQL Open Heterogeneous Data Access. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD (1998)
38. Reinwald, B., Pirahesh, H., Krishnamoorthy, G., Lapis, G., Tran, B.T., Vora, S.: Heterogeneous query processing through sql table functions. In: Proceedings of the International Conference on Data Engineering, ICDE (1999)
39. Saltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A.: Indexing the Positions of Continuously Moving Objects. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD (2000)
40. Song, Z., Roussopoulos, N.: Hashing Moving Objects. In: Proceedings of the International Conference on Mobile Data Management, MDM (2001)
41. Tao, Y., Papadias, D., Shen, Q.: Continuous Nearest Neighbor Search. In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2002)
42. Tao, Y., Papadias, D., Sun, J.: The TPR*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2003)
43. Tatbul, N., Cetintemel, U., Zdonik, S.B., Cherniack, M., Stonebraker, M.: Load Shedding in a Data Stream Manager. In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2003)
44. Xiong, X., Mokbel, M.F., Aref, W.G.: SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In: Proceedings of the International Conference on Data Engineering, ICDE (2005)
45. Xiong, X., Mokbel, M.F., Aref, W.G., Hambrusch, S., Prabhakar, S.: Scalable Spatio-temporal Continuous Query Processing for Location-aware Services. In: Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM (2004)
46. Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.: Location-based Spatial Queries. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD (2003)