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 *load-shedding* approach 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 wide spread of location-detection devices (e.g., GPS devices, handheld devices, and cellular phones) results in new environments where massive spatio-temporal data are continuously *streamed* out from mobile users. The high arrival rates of spatio-temporal *data streams* along with its massive data sizes make it infeasible for traditional spatio-temporal data management techniques to store, query, or index incoming spatio-temporal data. Unfortunately, most of the exiting techniques for spatio-temporal databases (e.g., see [27–29, 31, 33–35, 39, 43, 46, 48, 51, 52, 57]) rely mainly on the basic assumption that all incoming spatio-temporal data can be stored on disk. Thus, *continuous query* processing techniques (e.g, [27, 39, 52, 57]) aim to utilize the disk storage to produce incremental results of continuous queries. While this assumption is valid for certain data sizes and data arrival rates, it may not be feasible for high arrival rates and massive data sizes. When considering data streaming environment, only *in-memory* solutions are feasible.

On the other side, recent research efforts in data stream management systems (e.g., see [2, 7, 13, 14, 42]) focus mainly on processing continuous queries over traditional data streams. However, the *spatial* and *temporal* properties of both data streams and continuous queries are overlooked. Continuous query processing in *spatio-temporal streams* is distinguished from traditional data streams in the following: (1) Queries as well as data have the ability to continuously change their locations. Thus, spatio-temporal data streams are considered as a series of data updates rather than the append-only model of traditional data streams. (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 [4, 5, 23] does not support common spatio-temporal queries that are interested on the current state of the database rather than on the recent historical state.

In this paper, we aim to combine the recent advances in both the traditional spatio-temporal query processors and data stream query processors in order to provide an efficient query processing for *spatio-temporal streams*. Towards this goal, 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. On-line execution is achieved in SOLE by allowing only *in-memory* processing of each single data input as it is received by the system. Such on-line execution model is distinguished from most of the existing spatio-temporal continuous query processors (e.g., [39,46,57]) that buffer a set of updates together and process them once every T time units.

As in traditional data streaming application, the memory is the most scarce resource. Thus, memory in SOLE is efficiently utilized by keeping track of only those objects that are considered *significant*. A moving object is considered *significant* if it satisfies at least one active continuous query. As a result of keeping only those *significant* objects, continuous queries may encounter some regions of *uncertainty* in which certain moving objects may not be reported in the result. SOLE avoids such query *uncertainty* regions using a conservative *caching* approach in which the query area is extended to cover any possible uncertainty area. Scalability in SOLE is achieved by using a shared buffer pool that is accessible by all active 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 *load-shedding* approach that dynamically adopts the notion of *significant* objects based on the current workload. The main goal of *load-shedding* in SOLE is to support larger numbers of continuous queries, yet with an approximate answer.

The online nature of SOLE makes it possible to encapsulate its functionalities inside pipelinable query operators that can be combined with traditional query operators (e.g., join, aggregates, and distinct) in a query pipeline. Combining traditional query operators with SOLE operators enables the support for a wide variety of complex continuous spatio-temporal queries. In addition, having SOLE as query operators enables the involvement of the query optimizer to support multiple candidate execution plans for continuous spatio-temporal queries. Such design of SOLE results in orders of magnitude of performance than traditional spatio-temporal query processing techniques that can be implemented only on-top of existing database engines. The SOLE operator is implemented inside the PLACE server [38,40]; a prototype data stream management system for supporting spatio-temporal applications. In general, the contributions of this paper can be summarized as follows:

1. We propose SOLE as the the first attempt to combine spatio-temporal continuous query processing techniques with data stream management systems to support continuous queries over spatio-temporal data streams.
2. We show that due to the nature of data streaming environments, continuous spatio-temporal queries may encounter uncertainty areas. We show also that SOLE can overcome such uncertainty areas using a conservative caching technique.
3. We provide a scalable framework for SOLE that modifies the commonly used shared execution paradigm to support data streaming environments, uncertainty areas, and online execution of continuous spatio-temporal queries.
4. We provide load shedding schemes within SOLE that can be triggered at instances of high system workload. Load shedding techniques aim to support larger numbers of continuous queries with an approximate answer.
5. We encapsulate the functionalities of SOLE into pipelined query operators by utilizing the online nature of SOLE. The SOLE operators are implemented inside the PLACE prototype for spatio-temporal data stream management systems.
6. We provide experimental evidence, based on the real implementation of SOLE, that various aspects of SOLE (e.g., query operators, uncertainty management, scalability, and load shedding) can efficiently support large numbers of continuous queries over spatio-temporal data streams.

The rest of this paper is organized as follows: Section 2 highlights related work to SOLE in the context of spatio-temporal databases and data stream management systems. The basic concepts of SOLE are discussed in Section 3. The SOLE algorithms for single and multiple continuous spatio-temporal queries are presented in Sections 4 and 5, respectively. Section 6 discusses the load shedding techniques in SOLE. 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 index structures (e.g., hash tables [12,49], grid files [21,39,44], the B-tree [29], the R-tree [33,35], and the TPR-tree [48,52]). Thus, it is implicitly assumed that *all* incoming data can be stored. Scalable execution of continuous spatio-temporal queries is addressed recently for centralized [21, 39,46,57] and distributed environments [9,21]. However, the underlying data structure is either a disk-based grid structure [21,39] or a disk-based R-tree [9,46]. None of these techniques deal with the issue of spatio-temporal data streams where only in-memory solutions are allowed. Memory-based data structures have been proposed in [31,32,59] to deal with reasonable size of data that can fit in memory, but it is not scalable to large data sizes or streaming environments.

The most related work to SOLE in the context of spatio-temporal databases is the SINA framework [39]. SOLE has common functionalities with SINA where both of them utilize a shared grid structure as a basis for shared execution and incremental evaluation paradigms. 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 the employed data structures are built in memory while SINA is a *disk-based* query processing technique that mainly relies on the disk storage to perform its operations. (2) Due to the size limitations of memory, not all objects are really stored in SOLE. On the other side, in SINA, all data objects are physically stored. (3) As some data objects are not stored, SOLE suffers from having uncertainty areas in its queries where part of the query area may not be aware by the existence of some moving objects. Such scenario cannot happen in SINA as it is proven to be correct based on the knowledge of all stored objects. (4) SOLE is an *online* algorithm where it produces the incremental result with the change of any location of the query and/or objects. This online feature is in contrast to SINA where SINA buffers all the updates for the last T time units and processes them as a bulk. Such *online* behavior of SOLE makes it suitable to be encapsulated into a pipelined operator. On the other side, the *bulk* behavior of SINA hinders its applicability to be implemented inside real systems. (5) SOLE is equipped with *load shedding* techniques to cope with intervals of high arrival rates of moving objects and/or queries. The main idea is to drop some data objects from memory to allow for supporting more queries with an approximate answer. Such load shedding cannot be supported in SINA as it is mainly a disk-based algorithm and does not suffer from limited storage space.

2.2 Data Stream Management Systems

Existing prototypes for data stream management systems [1,10,13,15,26,30,42] 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 [4,5,23]. 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 [37] which is designed to deal only with the execution of a **single** continuous query.

Scalable execution of continuous queries in traditional data streams aims to either detect common subexpressions [14,15,36] or share resources at the operator level [4,20,24]. SOLE evaluates multiple spatio-temporal continuous queries 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* and adaptive memory management in data stream management systems are addressed recently in [6,11,18,19,47,53]. The main idea is to either add a special operator to the query plan to regulate the load by discarding unimportant incoming tuples or dynamically adjust the window size and time granularity at runtime. However, none of these approaches can be directly applicable to SOLE as they are not designed to deal with the spatial and temporal properties of data streams. In addition, none of these approaches deals with the special features of SOLE, e.g., uncertainty areas, concurrent spatio-temporal queries, and significant objects. Our proposed load shedding techniques are not competitive to any of the previous approaches. Instead, they are specifically designed to be applied within the SOLE framework in which previously proposed techniques cannot be applied.

The most related work to SOLE in the context of data stream management systems is the NiagaraCQ framework [15]. 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, incremental evaluation 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 including the input/output model, supporting various queries, SOLE pipelined operator, and the SQL syntax.

3.1 Input/Output Model

Input. The inputs to SOLE are two streams: (1) A stream of spatio-temporal data that is sent from continuously moving objects with the format $(OID, Loc, time)$, where OID is the object identifier, and Loc is the current location of the moving object at time $time$. For simplicity, we consider moving objects as moving *points* in the two-dimensional space. Such scenario depicts the moving of pedestrians, vehicles, or ships in the space. Extensions of SOLE to deal with moving regions with various extents and shapes can be done by replacing the Loc attribute to be a $Polygon$ attribute with size and shape. In the rest of this paper, we focus on the simple and common case of having moving points. Moving objects are required to send updates of their locations periodically. Failure to do so results in considering the moving object as disconnected. For example, if a moving P did not send any location update in the last t time units, SOLE would delete P from its memory and appropriate actions will be taken. (2) A stream of continuous queries. Queries can be sent either from moving objects or from external entities (e.g., a traffic administrator). Although, continuous queries may be be received with different formats, their internal representation at SOLE is unified. In general, a query Q is internally represented as $(QID, Region)$, where QID is the query identifier, and $Region$ is the spatial area covered by Q. The query region is determined based on the query type. For example, in range queries, the query region is the area that the query wants to monitor. The rest of this section gives details on how to set the query region.

Output. SOLE employs an incremental evaluation paradigm similar to the one used in SINA [39]. 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. This is in contrast to previous continuous query approaches (e.g., [21,34,46,50,51,60,61]) that abstract the continuous queries to a set of snapshot queries that are continuously reevaluated with the change of data inputs or queries. SOLE distinguishes 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* update. A *positive*/*negative* update indicates the addition/removal of object OID to/from query QID. For example, if a new object P becomes part of the query answer of Q, we send the *positive* update $(Q, +P)$. On the other side, if an object P that was in the query answer of Q changes its status to be out of the answer of Q, we send the *negative* update $(Q, -P)$ to the query. For more details about the concepts of *positive* and *negative* updates, the reader is referred to [38,39].

3.2 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 within 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* of Q. A moving query Q is submitted to SOLE 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. Internally in SOLE, the moving query is represented as $(QID, Region)$ where $Region$ is a moving area that changes its location according to the movement of the $FocalID$ moving object.

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. Internally in SOLE, the kNN query is represented as $(QID, Region)$ where the $Region$ is a circle with center c and radius r. The center c is either stated explicitly as $center$ 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. Then, the radius r is determined as the distance from the query center c to the 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 region 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.

3.3 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 [26], STREAM [42]) 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 [25].

3.4 SQL Syntax

Since SOLE is implemented as a query operator, we use the following SQL to 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 Execution of Single Continuous Queries in SOLE

To clarify the new ideas used in SOLE, in this section, we present SOLE in the context of single query execution [37]. In the next section, we show how SOLE can be generalized to the case of evaluating multiple concurrent continuous spatio-temporal queries.

4.1 Predicate-based Spatio-temporal Queries

Traditional stream query processing techniques (e.g., see [2,13,42]) employ the so-called *sliding-window queries* to accommodate the massive amount of streaming data. The main idea is to limit the execution of continuous queries to only the *recently* received data tuples rather than the whole received tuples. In sliding window queries, incoming streaming data follow a first-in-first-expire model in which whenever a tuple becomes old enough, it is expired (i.e., deleted) from memory leaving its space to a more *recent* tuple. As a result, traditional sliding-window queries can support only (recent) historical queries. Such model is not suitable for spatio-temporal queries where most of the spatio-temporal queries in mobile environments are concerned with the *current state* of data rather than the recent history.

To suit the needs of mobile environments, SOLE employs a new kind of window queries, termed, *predicate-based* window queries [22]. In *predicate-based* window queries, an incoming data tuple is stored in memory only if it satisfies the query predicate. Once an object becomes out of the predicate, it is expired (i.e., deleted) from memory. Thus, data tuples are expired out-of-order. To support *predicate-based* window queries in SOLE, for each query Q, we store the tuples that satisfy Q's predicate in a data structure termed $Q.Answer$. Then, if any object P with location P_{old} sends a new location update P_{new}, SOLE distinguishes among four cases:

- **Case I:** $P \in Q.Answer$ and P satisfies Q (e.g., Q_1 in Figure 1a). As SOLE reports only the updates of the previously reported result, P will not 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 not 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 the following four non-overlapped sets C_1 to C_4

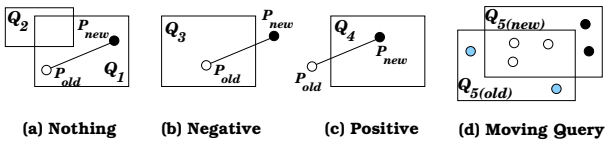


Fig. 1 Positive/Negative updates in SOLE.

where: (1) C_1 is represented by the white objects in Figure 1d where $C_1 \subset Q.Answer$ and every moving object in C_1 satisfies the new $Q.Region$. SOLE does not report any of the objects in C_1 as none of them affects the previously reported query result. (2) C_2 is represented by the gray objects in Figure 1d where $C_2 \subset Q.Answer$ and none of the objects in C_2 satisfies the query region. For each data object in C_2, SOLE produces a *negative* update. (3) C_3 is represented by the black objects in Figure 1d where $C_3 \not\subset Q.Answer$ and every moving object in C_3 satisfies the new $Q.Region$. For each data object in C_3, SOLE produces a *positive* update. (4) $C_4 \not\subset Q.Answer$ and none of the objects in C_4 satisfies $Q.Region$ (not shown in Figure 1d). SOLE does not produce any output for objects in C_4.

4.2 Memory Optimizations

In data streaming environments, storing data objects in disk is not a feasible solution while the memory storage is limited and is considered as the most scarce resource. To efficiently utilize the memory resource, SOLE stores only those data objects that are of interest to the outstanding continuous queries. Considering only a single outstanding continuous query Q, a moving object P will be stored in memory only if it satisfies Q. Similarly, if an object P which is stored in-memory becomes out of interest of Q, P is immediately dropped from memory.

As a general rule, the memory is only occupied by those objects that contribute to the query answer. For example, in Figure 1b, once P_{old} stepped out form the query region Q_3, it is discarded from memory while in Figure 1c, P_{new} will be stored in memory as it satisfies Q_4. Similarly, in Figure 1d, all gray objects will be dropped from memory as they become out of interest of Q_5. In this case, the query region is considered as the *predicate* in the *predicate-based* window query model where SOLE operates only on those data objects that satisfy the query predicate.

4.3 Uncertainty in SOLE

Since there are many data objects that are not physically stored in SOLE, i.e., those objects that are not of interest to the outstanding query, some *uncertainty* areas may take place. 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.

The query uncertainty is a new concept for spatio-temporal data streams. Traditional spatio-temporal query processing techniques (e.g., SINA [39]) do not suffer from any uncertainty as all location data updates are materialized in the disk storage. Thus, traditional spatio-temporal processors provide accurate results which is different from the case of SOLE where data are not materialized anywhere. In general, SOLE distinguishes among the following three types of uncertainty: *uncertainty in new queries*, *uncertainty in stationary queries*, and *uncertainty in moving queries*. Figure 2 gives an example of these uncertainty types as it represents a three consecutive snapshots of a database with ten moving objects P_1 to P_{10} and four queries Q_1 to Q_4.

1. **Uncertainty in new queries.** Initially, there are no active queries in the system. Thus, continuously arrived data streams are neither processed nor stored. Once a query Q is submitted to the system, we cannot provide a fast answer to Q, simply because there is nothing currently stored in the database. In this case, all the area covered by Q is considered an *uncertainty* area. Later on, moving objects update their locations and the answer of Q is progressively built. As an example, consider the moving object P_4 in Figure 2. P_4 arrives to the server at T_0. Since no query shows interest in P_4 at time T_0, P_4 is ignored and not stored in SOLE. Then, at time T_1, a new range query Q_3 is issued. At this time, all the region of Q_3 is considered uncertainty. Since P_4 is not stored in the system, it would not be reported in the query answer. At time T_2, object P_4 sends another location update to the server, yet, the new location update is outside Q_3, thus, it will not be included in the answer. Thus, due to the Q_3 uncertainty area, P_4 will not be reported in the query answer though it was in the answer from [T_1,T_2].
2. **Uncertainty in moving queries.** Uncertainty in moving queries comes from the fact that those queries tend to cover new spatial areas as they move. New areas may have moving objects that was dropped earlier. For example, consider the range query Q_1 in Figure 2. 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, i.e., the new area covered by Q_1. 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

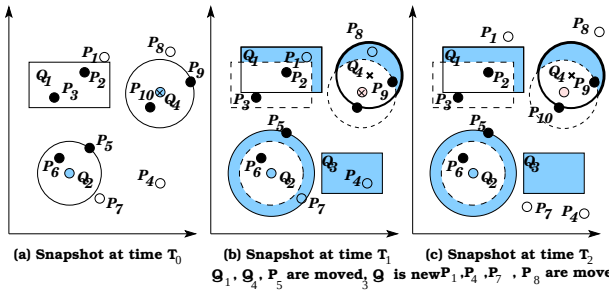


Fig. 2 Uncertainty in spatio-temporal queries.

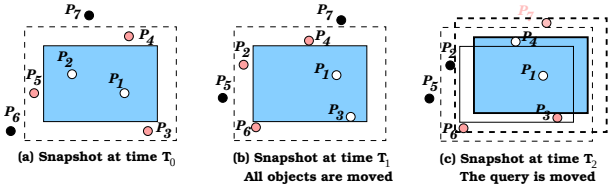


Fig. 3 Avoiding uncertainty in SOLE (range queries).

interval $[T_1, T_2]$. Another example for uncertainty in moving $k-$nearest-neighbor (k=2) queries is Q_4 in Figure 2. At time T_0 , P_8 is outside the area of Q_4. Thus, P_8 is not physically stored in the database. At time T_1, Q_4 is moved. The shaded area in Q_4 represents its *uncertainty* area. Although P_8 is inside the new query region, P_8 is not reported in the query answer where it is not actually stored. At T_2, P_8 moves out of the query region. Thus, P_8 is never reported at the query result, although it was inside the query region in the time interval $[T_1, T_2]$.

3. **Uncertainty in stationary queries.** Uncertainty in stationary queries comes from the fact that those queries may change their shapes over time. In this case, new spatial areas that are covered by the new shapes are considered as uncertainty areas. For example, consider the stationary k-nearest-neighbor query ($k = 2$) Q_2 in Figure 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 time 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 $[T_1, T_2]$.

In general, the uncertainty area in SOLE comes from the fact that moving objects are not actually stored in the database unless they are needed by existing queries. Such definition of uncertainty is an orthogonal definition from the location uncertainty in moving objects that has been used extensively in the literatures (e.g., [3,16,17,45, 54–56]). Location uncertainty refers to the lower resolution and inaccuracy of location-detection devices where the system is not aware of the exact location of moving objects. Instead, the system has a vague knowledge about the possible locations of moving objects. In contrast, in SOLE, the uncertainty is related to the query not to the object as new spatial areas are covered by existing or new queries.

4.4 Avoiding Uncertainty in SOLE

SOLE does not handle the *uncertainty* areas that result from the newly submitted continuous queries. New continuous queries suffer from uncertainty areas for the first few seconds where the query answer is built progressively. Continuous queries are issued to run for hours and days. Thus, having a *warming up* period for a few seconds does not affect neither the accuracy nor the efficiency of the query result. Section 7 provides more elaboration on the effect of uncertainty areas of new queries. On the other side, *uncertainty* areas that result from stationary or moving queries are crucial and are handled efficiently by SOLE.

SOLE avoids uncertainty areas in moving and stationary spatio-temporal queries using a *caching* technique. The main idea is to predict 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, SOLE probes the in-memory *cache* and produces 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*. The underlying assumption with the conservative cache approach is that all moving objects are required to report their location updates every t time units. Failure to do so would result in disconnecting the moving object. 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 [46], the valid region [60], and the No-Action region [58]) 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. In the rest of this section, we give two examples of using the conservative caching approach to avoid any uncertainty area in both moving and stationary queries.

Example 1. Moving Queries. Figure 3 gives an example of using *caching* to avoid uncertainty in moving queries. The shaded area represents the query region.

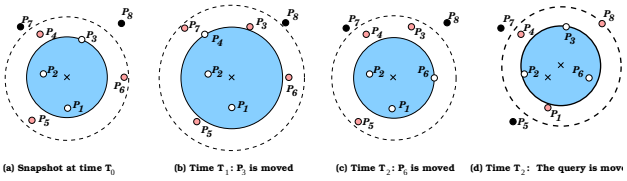


Fig. 4 Avoiding uncertainty in SOLE (kNN queries).

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^- and P_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^+.

Example 2. Stationary Queries. Figure 4 gives an example of continuous k-nearest-neighbor query ($k = 3$). A snapshot of the database at time T_0 is given in Figure 4a with P_1, P_2, and P_3 represent the query answer. P_4, P_5, and P_6 are stored in the cache list, while P_7 and P_8 are not stored in the database (since they are outside the cache region). At time T_1 (Figure 4b), object P_3 moves out of the query region but not outside the cached area. Since P_3 is still inside the *cache* area, we probe the *cache* list to find P_4 that is nearer to the focal point than P_3. Thus, we send a *negative* update P_3^- and a *positive* update P_4^+ to indicate that the current answer contains P_1, P_2, and P_4. At time T_2 (Figure 4c), P_6 moves from the cache area into the query area. Thus, P_6 is nearer to the query focal point than the kth previous answer (P_4). Thus, we send the *negative* update P_4^- and the *positive* update P_6^+ to indicate that the current answer contains P_1, P_2, and P_6. At time T_3 (Figure 4d), the query is moved along with its cache area. The query movement results in two updates: the *negative* update P_1^- and the *positive* update P_3^+.

4.5 Analysis of the Caching Approach

In this section, we study various parameters that affect the performance of the caching technique in terms of both the cache overhead and the query accuracy. Without loss of generality, we assume that the query original area is a square area with a side length x. Also, we assume that moving objects are distributed uniformly in the space.

Cache overhead. Assume that the *caching* technique would increase each side length of the square area by a distance d. Then, the overhead percentage of using the *caching* technique can be measured by the percentage of the increase in the total area from the original query to the extended query (i.e., the query area plus the cache area). Thus, the cache overhead can be formulated as:

$$cache\ overhead = 100 \times \frac{(x+d)^2 - x^2}{x^2}$$

Assuming that the original square side length x can be represented as a factor of the non-zero increase in the side length d, i.e., $x = md$, where m is termed as the *expansion factor* of the original query. Then, the cache overhead can be represented as

$$cache\ overhead = 100 \times \frac{(md+d)^2 - m^2d^2}{m^2d^2} \quad (1)$$

$$= 100 \times \frac{2md^2 + d^2}{m^2d^2} = 100 \times \frac{2m+1}{m^2} \quad (2)$$

This means that the larger the expansion factor m, the lower the cache overhead. For example, if m is so large, (i.e., order of tens), the cache overhead percentage will be boiled down to be $\frac{200}{m}$. Having m as 50 will result in only 4% overhead.

To get a better estimation of the value of the expansion factor m and the effect of various paraments, we consider that moving objects have a maximum velocity of v miles per hour. Furthermore, moving objects are assumed to report their locations to the server every t seconds, otherwise, moving objects will be considered as disconnected. Thus, the maximum possible distance d_{max} that a certain moving object can travel between any two consecutive updates is $d_{max} = \frac{t \times v}{3600}$. Then, for a *conservative* caching approach, we set $d = 2d_{max}$ to indicate the increase of each query region side by the maximum possible distance. However, for a non-conservative approach, we only set $d = 2 \times c \times d_{max}$ where $c, 0 \leq c \leq 1$, is a factor that indicates the percentage of caching we would like to have. Having $c = 1$ indicates the conservative caching approach while having $c = 0$ indicates that no caching is used. In terms of the velocity v and the time interval t, the distance d can be represented as:

$$d = \frac{t \times v \times c}{1800}$$

Since, $x = md$, the expansion factor m can be represented as:

$$m = \frac{1800 \times x}{t \times v \times c} \quad (3)$$

As given in Equation 2, the higher the value of m, the lower the cache overhead. Then, Equation 3 determines the factors that affect the cache overhead. For example, the higher the value of x, the original query side length, the lower is the cache overhead percentage. The main

idea is that the larger the original query, the lower the effect of extending its region. Similarly, the lower the value of c, the higher the value of m, and hence the lower is the cache overhead. Recall that $0 \leq c \leq 1$, the lowest value of c would result in a very large value of m. In contrast increasing the value of v and/or t reduces m and hence increases the cache overhead. This indicates that if moving objects are moving with very high velocity, then it is expected that moving objects would travel relatively long distances between two consecutive updates. Then, the cache area needs to have a large area to accommodate such distance. Similarly, if the time interval t between any two updates is relatively large, then the distance between two consecutive updates would call for a large cache area and hence a large percentage of the cache overhead.

Query accuracy. The *conservative* cashing approach guarantees to have 100% query accuracy as all uncertainty areas are covered, i.e., $c = 1$. Thus, the query accuracy Q_A is measured as the ratio of the extended query area with respect to the area covered by the *conservative* approach:

$$Q_A = 100 * \frac{(x + 2cd_{max})^2}{(x + 2d_{max})^2}$$

Given that $d_{max} = \frac{t \times v}{3600}$, then:

$$Q_A = 100 * \left(\frac{x + \frac{tvc}{1800}}{x + \frac{tv}{1800}} \right)^2 = 100 * \left(\frac{1800x + tvc}{1800x + tv} \right)^2 \quad (4)$$

Example. In a practical scenario, consider a square range query with side length $x = 3$ miles that monitors the traffic in a downtown area. If objects are moving with speed $v = 30$ miles/hour while updating their locations every $t = 30$ seconds, then the maximum traveled distance for each object is $d_{max} = 1/4$ mile. Using a *conservative* caching approach, i.e., $c = 1$, then the increase in the side length is $d = 1/2$ mile. Thus, the expansion factor $m = 6$, and the percentage of the increase in the query area is only around 35% (from Equation 2). On the other hand, because $c = 1$, then the query accuracy is 100% (from Equation 4). However, if we use a non-conservative caching with $c = 0.5$, then, $d = 1/4$, $m = 12$, and the cache overhead will be only 17% (Equation 2), while the query accuracy will be dropped to 86% (Equation 4). Similarly, if $c = 0.25$, the cache overhead will be only 8.5% while the query accuracy is 80%. Finally, in the extreme case, i.e., when $c = 0$, there is no cache overhead at all. In this case, as computed from Equation 4, the query accuracy drops to 73%.

5 SOLE: Scalable On-Line Execution of Continuous Queries

In a typical spatio-temporal application (e.g., location-based 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). Similar to the SINA framework [39], SOLE employs both *shared execution* and *incremental evaluation* paradigms as a means to achieve scalability. However, SOLE employs these paradigms in a completely different environments that include, data streaming, in-memory only algorithms and data structures, online execution where the query answer is immediately updated with any change in the input. 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, Figure 2, and Figure 4. Basically, an kNN query is treated as range queries, yet with only a variable region size.

5.1 Overview of Sharing in SOLE

Figure 5a 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 5b gives the pipelined execution of the same N queries as in Figure 5a, 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 [15] 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.

5.2 Shared Memory Buffer

SOLE maintains a simple grid structure that divides the space into equal non-overlapped rectangular cells as

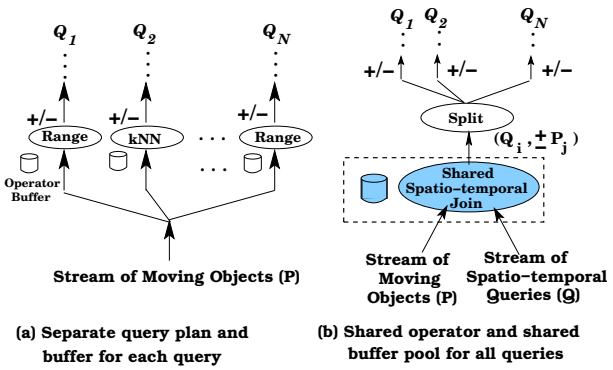


Fig. 5 Overview of shared execution in SOLE.

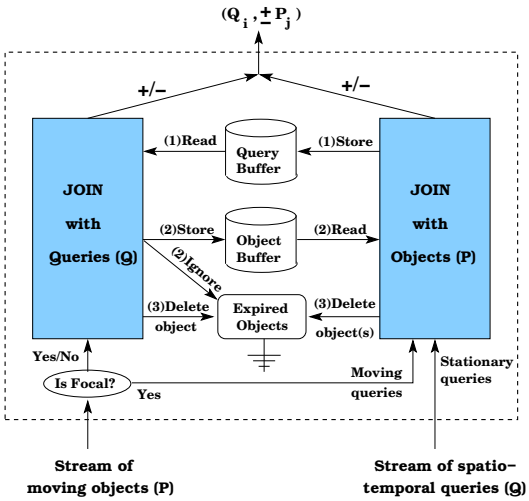


Fig. 6 Shared join operator in SOLE.

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 active 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 active continuous query.

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

Definition 3 A 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) If an object P that is already stored 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 in the format $(QID, Region)$.

5.3 Shared Spatio-temporal Join Operator

Overview. Figure 6 puts a magnifying glass over the shared spatio-temporal join operator in Figure 5b. For any incoming data object, say P, the shared spatio-temporal 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 spatio-temporal 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 mov-

Algorithm 1 Pseudo code for receiving a new object P

```
1: Function NewObject(Object P, GridCell C_P)
2: for each Query Q_i ∈ C_P AND P ∈ Q̂_i do
3:    P.RefCount++
4:    if (P ∈ Q_i) then
5:       output (Q_i, +P).
6:    end if
7: end for
8: if P.RefCount > 0 then
9:    store P in C_P and in hash table h.
10: end if
```

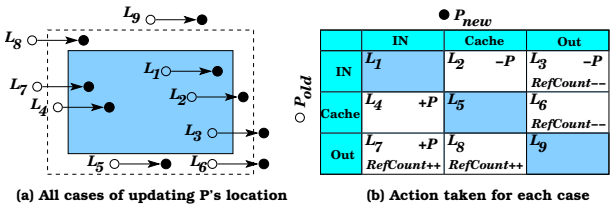


Fig. 7 All cases of updating P's location.

ing query Q. Algorithms 1, 2, 3, and 4 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.

Input Type I: A new object P. Algorithm 1 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 (Lines 2 to 7 in Algorithm 1). 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 (Lines 3 in Algorithm 1). 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 (Lines 4 to 6 in Algorithm 1). 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**.** Algorithm 2 gives the pseudo code of SOLE upon receiving an update of object 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 (Lines 2 to 4 in Algorithm 2). Then, we check all the queries that belong to either C_P or $C_{P_{old}}$ (Lines 6 to 26 in Algorithm 2) against the line L that connects P and P_{old}. Figure 7a 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

Algorithm 2 Pseudo code for updating P's location.

```
1: Function UpdateObject(Object P_old,P, GridCell
   C_Pold,C_P)
2: for each query Q_i ∈ P.FocalList do
3:    UpdateQuery(Q_i)
4: end for
5: Let L be the line (P_old, P)
6: for each query Q_i ∈ (C_Pold ∪ C_P) do
7:    if Q_i intersects L then
8:       if P ∈ Q_i then
9:          Output (Q_i, +P)
10:         if P_old ∉ Q̂_i then
11:            P.RefCount++
12:         end if
13:      else
14:         Output (Q_i, −P)
15:         if P ∉ Q̂_i then
16:            P.RefCount−−
17:         end if
18:      else if Q̂_i intersects L then
19:         if P ∈ Q̂_i then
20:            P.RefCount++
21:         else
22:            P.RefCount−−
23:         end if
24:      end if
25:   end if
26: end for
27: if P.RefCount = 0 then
28:    delete P_old and ignore P
29:    return.
30: end if
31: if C_Pold ≠ C_P then
32:    move P_old from C_Pold to C_P.
33: end if
34: Update the location of P_old to that of P in C_P.
```

states, *in*, *cache*, or *out* that indicates that P satisfies Q, in the cache area of Q, or does not satisfy Q, respectively. The actions taken for each case is given in Figure 7b. 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 (Lines 27 to 30 in Algorithm 2). If P is still *significant*, we update P's location and cell (if needed) in the grid structure (Lines 31 to 34 in Algorithm 2).

Algorithm 3 Pseudo code for receiving a new query Q.

```
1: Function NewQuery(Query Q)
2: for each grid cell c_j ∈ Ĉ_Q do
3:    Register Q in c_j
4:    for each object P_i ∈ c_j AND P_i ∈ Q̂ do
5:       P.RefCount++
6:       if P ∈ Q then
7:          output (Q, +P)
8:       end if
9:    end for
10: end for
```

Algorithm 4 Pseudo code for updating Q's location.

```
1: Function UpdateQuery(Query Q_old, Q)
2: for each object P_i ∈ (Ĉ_{Q_old} ∩ Ĉ_Q) do
3:    if P_i ∈ Q_old then
4:       if P_i ∉ Q then
5:          Output (Q, −P_i)
6:          if P_i ∉ Q̂ then
7:             P_i.RefCount − −
8:             if P_i.RefCount = 0 then
9:                delete P_i
10:            end if
11:         end if
12:      end if
13:   else if P_i ∈ Q then
14:      Output (Q, +P_i)
15:      if P_i ∉ Q̂_old then
16:         P_i.RefCount++
17:      end if
18:   else if P_i ∈ Q̂_old AND P_i ∉ Q̂ then
19:      P_i.RefCount − −,
20:      if P_i.RefCount = 0 then
21:         delete P_i
22:      end if
23:   else if P_i ∈ Q̂ AND P_i ∉ Q̂old then
24:      P_i.RefCount++
25:   end if
26: end for
27: Register Q in Ĉ_Q − Ĉ_{Q_old},
28: unregister Q from Ĉ_{Q_old} − Ĉ_Q
```

Input Type III: A new query Q. Algorithm 3 gives the pseudo code of SOLE upon receiving a continuous 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 lie in $\hat{Q}$. In addition, objects that satisfy Q results in producing *positive* updates.

Input Type IV: An update of Q's region. Algorithm 4 gives the pseudo code of SOLE upon receiving an update of a moving query region. The update can be either coming from the user directly or from a change of location of the *focal* query object. Also, the query update can be either an update in location or an update in the query area size. All stored objects in all cells that are covered by the old and new regions of Q are tested against Q. Figure 8a divides the space covered by the old and new regions of Q into seven regions (R_1-R_7). The actions taken for any point that lies in any of these regions

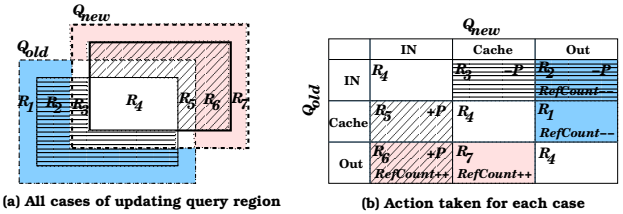


Fig. 8 All cases of updating Q's region.

are given in Figure 8b. Similar to Figure 7b, 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) (Lines 3 to 12 in Algorithm 4). 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) (Lines 13 to 17 in Algorithm 4). 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 (Lines 18 to 25 in Algorithm 4). 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 Load Shedding 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., during rush hours). To cope with such unexpected intervals, SOLE employs a *load-shedding* approach that tunes the memory load to support a large number of concurrent queries, yet with an approximate answer. The main idea is to change the definition of *significant* objects (Definition 2) 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 already existing object becomes insignificant according to the new definition, it is dropped from memory. (2) Newly input data will be tested for significance according to the new definition. If an object does not meet the new definition of significant objects, it will be ignored.

The rest of this section is organized as follows. Section 6.1 gives a high level architecture of the integration

of the *load shedding* module within the SOLE framework. The accuracy of load shedding is discussed in Section 6.2. Sections 6.3 and 6.4 propose two new methods for realizing load shedding inside SOLE, namely, *query load shedding* and *object load shedding*. Finally, Section 6.5 discusses maintaining the query accuracy while performing the load shedding.

6.1 Architecture of Load Shedding

Figure 9 gives the architecture of *load-shedding* 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 9 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.

6.2 Accuracy of Load Shedding

Load shedding aims to drop some of the in-memory tuples which may be needed by some outstanding queries. As a result, load shedding produces approximate query results. To make sure that the approximate query results are acceptable, whenever a query, say Q, is submitted to SOLE, Q specifies its minimum acceptable accuracy. Initially, every query Q is evaluated with complete accuracy. However, when the system is overloaded, Q's accuracy is degraded to its minimum permissible accuracy. Assuming a uniform distribution of moving objects overall the space, the accuracy of the query answer of SOLE is defined as $Acc_Q = \frac{100 \times N_{Current}}{N_{Actual}}$ where $N_{Current}$ is the number of stored moving objects within the query range and cache areas while N_{Actual} is the number of objects that should be in both the query range and cache area if load shedding was not employed. Notice that in the case of no load shedding, the query accuracy is 100%. Our definition of the query accuracy is independent from the query type as it relies mainly on the query area. For example, the accuracy of nearest neighbor queries is computed based on the area it covers not on the required number of neighbors.

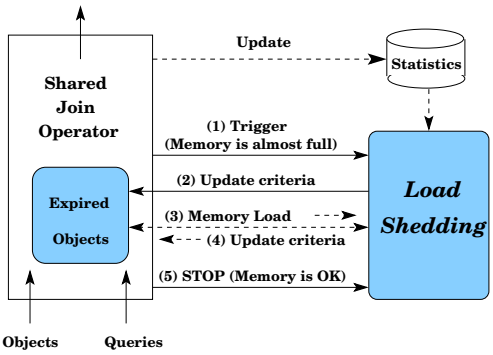


Fig. 9 Architecture of load shedding in SOLE.

6.3 Query Load Shedding

The main idea of the query load shedding is to shrink the query area. For example, if it is required to reduce the memory load to only 75%, then we aim to shrink the query area for each single query to its 75%, given that this will be within the permissible query accuracy. Query load shedding is performed in two stages. In the first stage, the query cache area is shrunk. All moving objects that become out of the new query area are eliminated if they are not needed by other queries. If the first stage did not result in the desired load shedding, the second stage starts by shrinking the query main area till the minimum permissible accuracy for each query is met or the memory load becomes acceptable. With the query load shedding, all the algorithms given in Section 5 are still valid. The only difference is that the notion of *significant* objects is adopted to be those tuples that lie in the *reduced* query area of at least one continuous query. By reducing the query sizes of all continuous queries, objects that are outside 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.

Figures 10a and 10b give an example of query load shedding. The complete snapshot of the database without load shedding is given in Figure 10a with seven queries Q_1 to Q_7 and 15 moving objects. Figure 10b gives the snapshot of the database after applying the query load shedding. Each query area (including the cache area) is reduced to 90%. This results in dropping a total of four objects (plotted as white circles in Figure 10b) from Q_1, Q_4, Q_5, and Q_6. Given an assumption of a uniform data distribution over the whole space and the query region, reducing the query area by 10% would result in a 90% query accuracy.

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 or

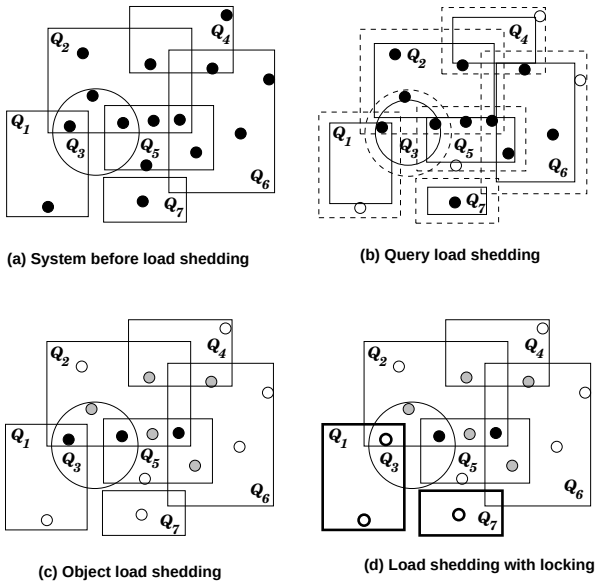


Fig. 10 Example of load shedding in SOLE.

additional data structures, and (2) *Insignificant* objects are immediately dropped from memory. On the other hand, there are two main disadvantages for the query load shedding: (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) Reducing the query accuracy to $x\%$ does not guarantee reducing the memory load to its $x\%$. Since the main objective is to reduce the memory workload, this may end up in trying to have more stages in reducing the query area. 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. As an example, consider the case of Q_4 in Figure 10b, reducing the query area results in ignoring the three moving objects inside Q_4. However, only one of these three objects is really dropped from the memory. The other two object did not get dropped form memory as they are still needed by Q_2 and Q_6.

6.4 Object Load Shedding

The *object load shedding* aims to avoid the drawbacks of the *query load shedding* by smartly choosing the objects to drop so that a dropped object has less effect on the query accuracy. The main idea is to drop those data objects that are of interest of only low number of queries. To realize such idea, the definition of *significant* objects is changed to be those objects that are of interest to at least k queries (i.e., objects with a reference counter greater than or equal k). Then, the object load shedding drops data objects with a reference counter lower than k. Notice that the original definition of *significant* objects implicitly assumes that $k = 1$.

Data structure. A main challenge in the object load shedding is to decide upon the value k. Thus, we maintain a simple one-dimensional statistical array S where $S[i]$ is the number of moving objects with a reference counter i. If a new object is received with a non-zero reference counter, k, in addition to storing this object in the hash table (Line 9 in Algorithm 1), we increase the entry $S[k]$ by one. Similarly, when an object with a reference counter k is dropped from memory, we decrease the array entry $S[k]$ by one. Similarly, if due to the execution of any of the algorithms 2, 3, or 4, a certain object changes its reference counter from k_{old} to k_{new}, we decrease $S[k_{old}]$ by one while increasing $S[k_{new}]$ by one.

Algorithm. Once the memory is overloaded with N data objects and the system decides to drop the memory load to be only $x\%$ of the current load, we consult the statistical array S for an appropriate value of k. The main idea is to initialize a counter C by zero, then, we scan S starting from $S[1]$ while accumulating its values into $C = C + S[k]$. We stop only when the ratio of C/N is less than the desired ratio x. At this point, we set the *new* notion of significant objects to be those objects that are of interest to less than k queries. To accommodate such change in SOLE, we modify the condition of line 8 in Algorithm 1 to be $P.RefCount > k$. Also, we modify the condition of line 27 in Algorithm 2 to be $P.RefCount < k$. Finally, we modify the condition of line 20 in Algorithm 2 to be $P.RefCount < k$.

Example. Figure 10c gives an example of object load shedding with seven outstanding queries Q_1 to Q_7 and 15 data objects. Objects that are plotted in white, gray, and black represent those objects with reference counter one, two, and three, respectively. Thus, $S[1] = 7$, $S[2] = 5$, and $S[3] = 3$ to indicate that the number of objects that has reference counters one, two, and three are seven, five, and three, respectively. To reduce the memory load o 80%, we will need to set k to two. In this case, all white circles are *candidate* to be dropped. However, they are not dropped immediately. Instead, they are dropped only when they get accessed till the memory is reduced to the desired load. Once the memory load becomes 80%, we set the value of k to one again and stop dropping memory objects.

Advantages. 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 addition, in contrast to the *query* load shedding, in the *object* load shedding, we guarantee the reduced memory load as we have the ability to choose the objects that we want to drop.

6.5 Maintaining the Query Accuracy in Load Shedding

Each query submitted to SOLE would have a minimum permissable accuracy. A straight forward application of either the query load shedding or the object load shedding does not guarantee the minimum permissable accuracy. For example, in Figure 10b, the query load shedding shrinks Q_1 slightly in which one object is dropped among only two objects that are of interest to Q_1. Thus, the accuracy of Q_1 is dropped to 50% which could be lower than the minimum required accuracy. For the object load shedding, dropping one object from Q_7 in Figure 10c results in dropping its accuracy to zero as this object is the only one that satisfies Q_7.

To avoid such accuracy violation, each query has the ability to *lock* itself once it discovers that removing any object from its answer will degrade its accuracy below the required level. Figure 10d gives an example of object load shedding with *locking* where each query has a minimum accuracy requirement 60%. In this case, both Q_1 and Q_7 (plotted as bold rectangles) lock themselves where removing an object from either Q_1 or Q_7 will degrade its accuracy to be 50% or 0%, respectively. Thus, all moving objects in Q_1 and Q_7 are locked and are not subject to dropping. To facilitate the execution of object load shedding technique, *locked* objects do not contribute in the computation of the statistical table S. Thus, once an object with a reference counter k is locked, the corresponding entry $S[k]$ is decreased by one.

To make sure of the current query accuracy, we need to take care of two types of dropped memory objects: (1) Objects that are dropped from memory, and (2) Objects that are dropped from the input and before being stored in memory. For each dropped object, we reduce the accuracy of all the queries affected by the dropped objects. To prevent the case that a dropped object is reported twice, and hence, mistakenly reduce the query accuracy, we keep track of a *shadow* table. The *shadow* table only keeps track of the object identifiers of dropped objects. A complete in-memory object may require large storage to store its location, focal list, and other attributes (if any), however, an object that is stored in the *shadow* table has only the object identifier. Having the *shadow* table, when a newly coming object P is considered *insignificant* as it satisfies less than k queries and before completely ignoring P, we go through the *shadow* table and make sure if this object was dropped before or not. If P is not in the shadow table, then we only reduce the accuracy of all the queries that P should be part of their answer. In this process, we make sure that no query will have an accuracy that is lower than its minimum permissible one. However, in the case that the object identifier of P is already in the *shadow* table, we go through the queries that are affected before from P and we update their accuracy only if they got affected by p movement. In both case, object P is dropped and its object identifier is stored at the *shadow* table.

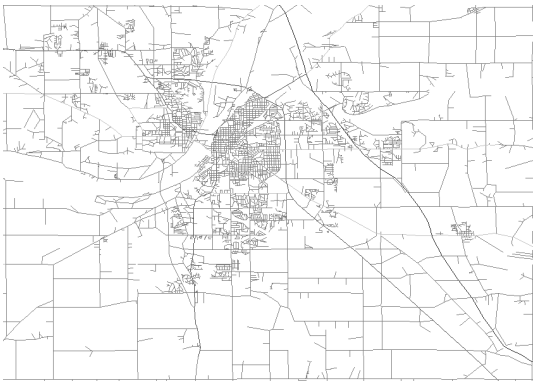


Fig. 11 Greater Lafayette, Indiana, USA.

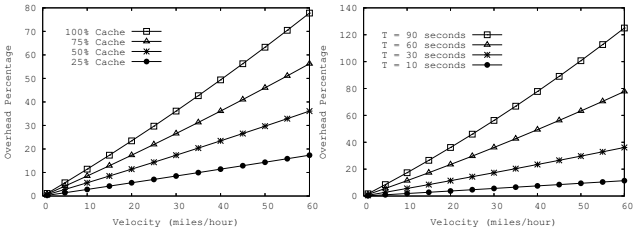
7 Experimental Results

In this section, we study the performance of various aspects of SOLE that include: 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 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 [40,41]. We run PLACE on Intel Pentium IV CPU 2.4GHz with 512MB RAM running Windows XP. As indicated throughout the paper, SOLE deals with queries by their regions that may change their locations (e.g., moving range queries) or change their shapes (e.g., stationary nearest-neighbor queries). It is not our objective in this section to compare various query types with each other, instead, we aim to show the effectiveness of applying SOLE to existing queries. For example, although nearest-neighbor queries generally result in higher cost than that of range queries, the effect of applying SOLE techniques to nearest-neighbor queries is similar to that of applying SOLE techniques for range queries. Thus, without loss of generality, all the presented experiments are conducted using rectangular region queries.

We use the *Network-based Generator of Moving Objects* [8] 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 11 which is almost a square area of side length 28 miles. 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. Sections 7.1 to 7.3 study the effect of the cache size, the velocity, and the gain of having SOLE as a pipelined operator in terms of single query execution. In Sections 7.4 to 7.6, we study the scalability of SOLE. Finally, Sections 7.7 and 7.8 study the performance of load shedding techniques.



(a) Cache (b) Time Interval

Fig. 13 Effect of velocity.

7.1 Single Execution: Size of the Cache Area

Figures 12a-d give the performance of the first 25 seconds of executing a moving query with a square region of a side length 2 miles with no cache, 25% cache, 50% cache, and *conservative* cache (i.e., 100% cache), respectively. Such query represents 0.5% of the whole space. Also, an x% cache area corresponds to setting the c factor in Section 4.5 to $x/100$. Our performance measure is the query accuracy that is represented as the percentage of the number of data objects that lie on the query region to the actual number that should have been in the query region if all moving objects are materialized into secondary storage. Notice that this definition of accuracy is similar to the one used with load shedding accuracy in Section 6.2 and is independent from the query type as it deals with the query region itself regardless of the query type. For all cache sizes, once a query Q is submitted to the system, Q needs a *warming up* period to complete its result. The *warming up* period is represented in Figures 12a-d as the initial line that is (almost) parallel to the vertical axis. Figure 12e provides a zoom on the *warming up* period.

Without caching (Figure 12a), 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 12b). 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 (Figure 12d). Although continuous queries are expected to last for hours and days, we plot only the first 25 seconds of the query execution. The main reason is that these few seconds represent the steady state behavior of the query execution along its course of execution.For example, having 50% cache would always have near-to-optimal result with very few drops in accuracy every now and then, while having 100% caches has always a steady state performance of 100% accuracy. The main reason of of having a very good performance with a non-conservative caching area (e.g., 50%) is that most of the moving objects do not move with the maximum speed. In addition, the only case that a conservative cache would be better than a non-conservative one is the case of a moving object who lies on the boundary of the query region and moves with its maximum speed in one direction. If this case did not take place, then a slightly non-conservative cache approach would achieve a high performance.

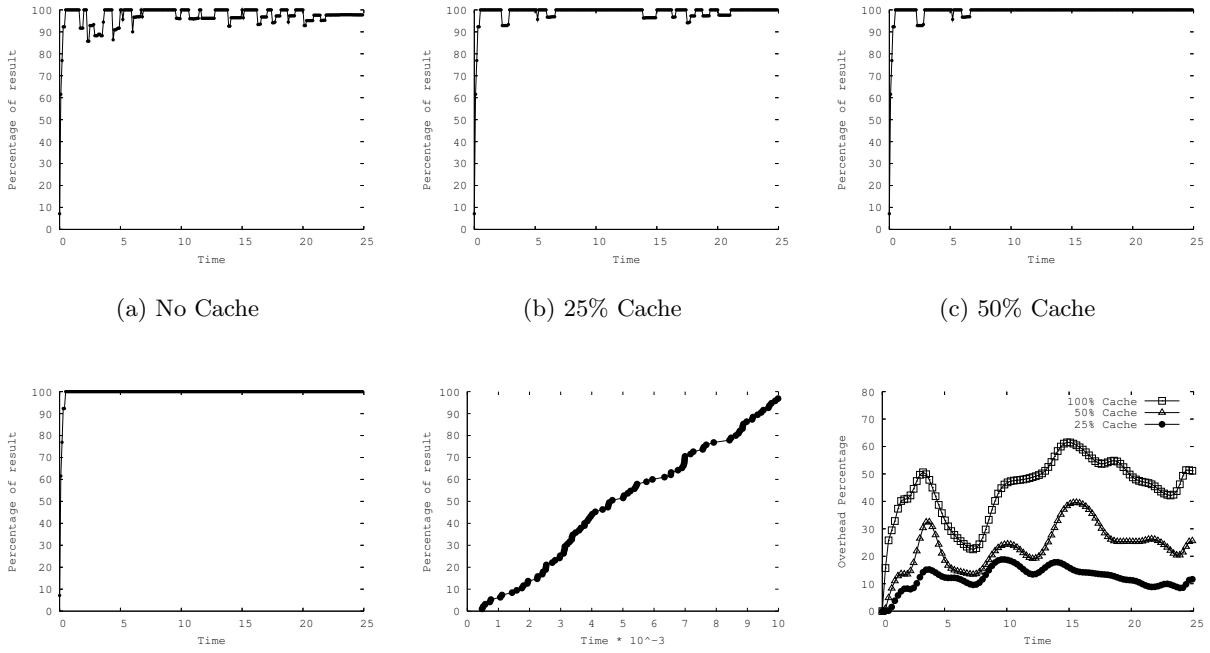
Figure 12e puts a magnifying glass over the *warming up* period (the first 10 msec) of the query execution time of Figures 12a-d. This warming up period corresponds to the uncertainty of new queries that has been described in Section 4.3. The main idea of this figure is to show that the query answer is built progressively and it took very small time from the query execution time to reach to a steady state performance. Thus, the uncertainty that comes from new queries can be amortized by the long running time of execution queries.

Figure 12f 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. These results are consistent with our analytical analysis in Section 4.5.

7.2 Single Execution: Effect of Velocity

Figure 13a gives the effect of the maximum object velocity on the cache overhead for different cache sizes (25%, 50%, 75%, and 100% cache size). The query size is 0.5% of the space while moving objects report their locations every 30 seconds. The maximum object velocity varies from 1 to 60 miles/hour. The increase in the velocity linearly increases the cache overhead. Also, increasing the cache size increases the cache overhead as was depicted also in Figure 12f. In addition, the slope of the effect of velocity over the cache overhead increases with the increase in the cache area.

Figure 13b exploits a similar experiment to that of Figure 13a. The only difference is that we set the cache area to be 50% while running the experiment for different values of the time interval t (10, 30, 60, and 90 seconds). Consistently with the analytical analysis given



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

(d) Conservative Cache (e) Warm up period (f) Cache Overhead

Fig. 12 Cache area in SOLE.

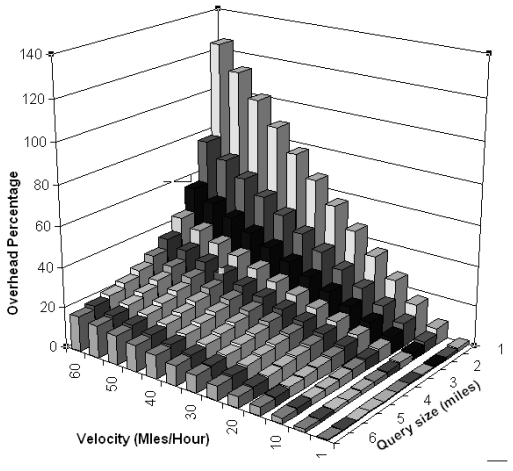


Fig. 14 Effect of velocity and query size.

in Section 4.5, increasing the time interval t results in an increase in the cache overhead.

Figure 14 studies the effect of both the velocity and query size on the cache overhead. The cache area is set to be 50% while the time interval t is set to 30 seconds. The maximum object velocity varies from 1 to 60 miles per hour while the side length of the query square area varies from 1 mile (0.13% of the space) to 6 miles (4.6% of the space). The worst case scenario takes place at the smallest query size (1 mile) with the largest maximum velocity (60 miles/hour). In this case, the cache overhead may exceed 100%. On the other side, for small velocity ($v \leq 10$), the overhead is almost negligible for all query sizes. Similarly, for large query sizes, the cache overhead is almost negligible for all velocities. For example, for query sizes greater than 5 miles, the cache overhead is always less than 20% regardless of the maximum object velocity. Also, for query sizes between 3 and 5 miles, the cache overhead is always less than 40%.

Notice that in this section we have studied the effect of the various parameters on the cache overhead, but not on the query accuracy. The query accuracy is controlled by the size of the cache area. For example, a 100% cache will always result in a 100% query accuracy regardless of the value of other parameters, e.g., velocity v, time interval t, or query size x.

7.3 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 15a gives

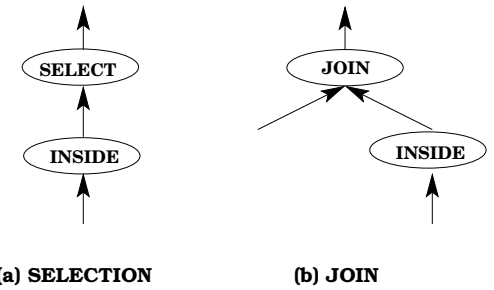


Fig. 15 Pipelined SOLE operators.

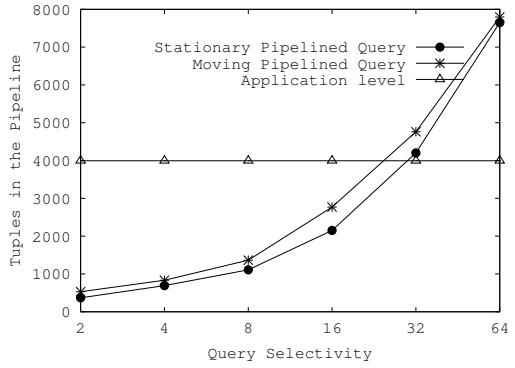


Fig. 16 Pipelined operators with SELECT.

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
```

Figure 16 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,

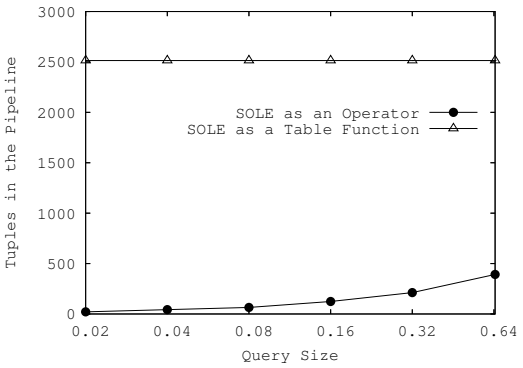


Fig. 17 Pipelined operators with Join.

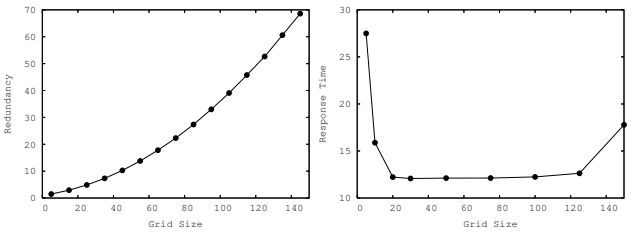
the output of the join is sent to the SOLE algorithm for further processing. However, with the `INSIDE` operator, we can have a query evaluation plan as that of Figure 15b where the `INSIDE` operator is pushed below the *Join* operator. The SQL representation of the above query is as follows:

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

Figure 17 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 16, 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.

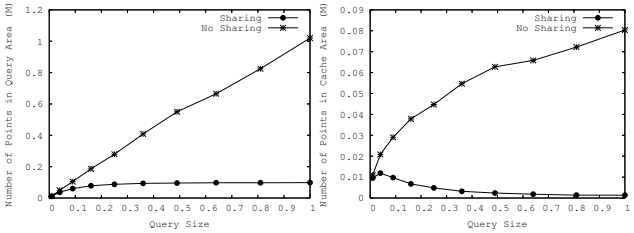
7.4 Scalable Execution: Grid Size

Figure 18 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



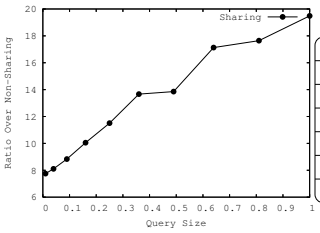
(a) Redundancy (b) Response Time

Fig. 18 Grid Size.



(a) Query area (b) Cache area

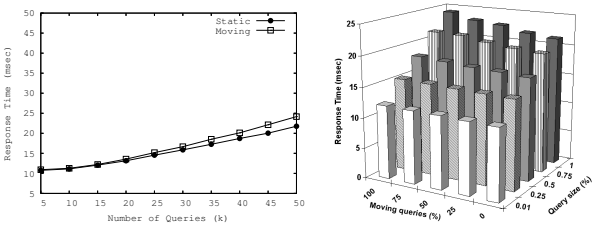
Fig. 20 Data size in the query and cache areas.



%	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. 19 Maximum Number of Supported Queries.



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

Fig. 21 Response time in SOLE.

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.

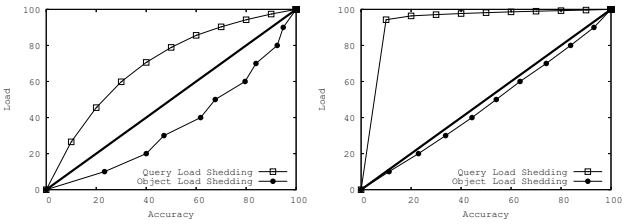
7.5 Scalable Execution: SOLE Vs. Non-Shared Execution

Figure 19 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 19a gives the ratio of the number of supported queries via sharing over the non-sharing case for various query sizes. Some of the actual values are depicted in the table in Figure 19b. 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 20a and 20b 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 20a, 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 20b 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.

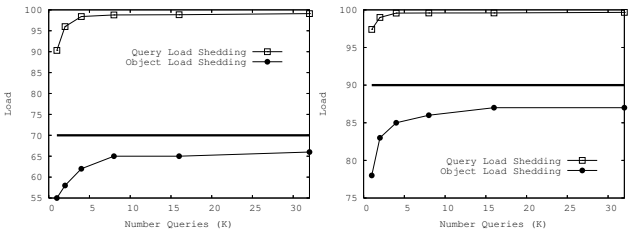
7.6 Scalable Execution: Response Time

Figure 21a gives the effect of the number of concurrent continuous queries on the performance of SOLE. The



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

Fig. 22 Load Vs. Accuracy.



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

Fig. 23 Reduced load for a certain accuracy.

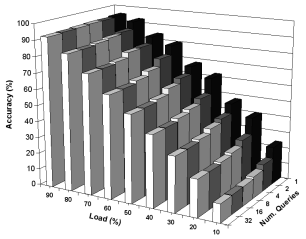


Fig. 24 Performance of Object Load Shedding.

number of queries varies from 5K to 50K. Our performance measure is the average response time. 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 21b 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.

7.7 Load Shedding: Accuracy in Query Answer

Figures 22a and 22b 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 certain 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 22 represents the required accuracy. It is "expected" that if we ask for m% accuracy, we 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 23a and 23b 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.

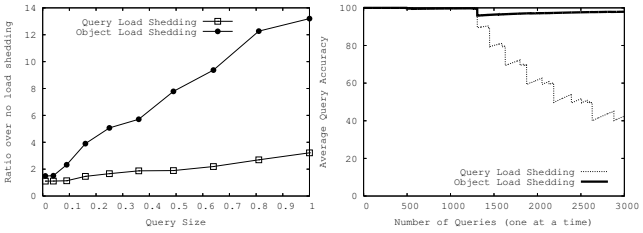
Figure 24 focuses on the performance of *object* load shedding. The required reduced load varies from 10% to 90% while the number of queries varies from 1K to 32K.

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.

7.8 Load Shedding: Scalability with Load Shedding

Figure 25a 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 19b). 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 25b 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. 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.



(a) Ratio (b) Query Arrival

Fig. 25 Scalability with Load Shedding.

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), 120–139 (2003)
3. de Almeida, V.T., Güting, R.H.: Supporting Uncertainty in Moving Objects in Network Databases. In: Proceedings of the ACM Symposium on Advances in Geographic Information Systems, ACM GIS, pp. 31–40. Bremen, Germany (2005)
4. Arasu, A., Widom, J.: Resource Sharing in Continuous Sliding-Window Aggregates. In: Proceedings of the International Conference on Very Large Data Bases, VLDB (2004)
5. 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)

6. 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)
7. Babu, S., Widom, J.: Continuous Queries over Data Streams. SIGMOD Record **30**(3), 109–120 (2001)
8. Brinkhoff, T.: A Framework for Generating Network-Based Moving Objects. GeoInformatica **6**(2), 153–180 (2002)
9. 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)
10. Cammert, M., Heinz, C., Krämer, J., Riemenschneider, T., Schwarzkopf, M., Seeger, B., Zeiss, A.: Stream Processing in Production-to-Business Software. In: Proceedings of the International Conference on Data Engineering, ICDE. Atlanta, GA (2006)
11. Cammert, M., Krämer, J., Seeger, B., Vaupel, S.: An Approach to Adaptive Memory Management in Data Stream Systems. In: Proceedings of the International Conference on Data Engineering, ICDE. Atlanta, GA (2006)
12. 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)
13. 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)
14. Chandrasekaran, S., Franklin, M.J.: PSoup: a system for streaming queries over streaming data. VLDB Journal **12**(2), 140–156 (2003)
15. 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, pp. 379–390 (2000)
16. Cheng, R., Kalashnikov, D.V., Prabhakar, S.: Querying Imprecise Data in Moving Object Environments. IEEE Transactions on Knowledge and Data Engineering, TKDE **16**(9), 1112–1127 (2004)
17. Dai, X., Yiu, M.L., Mamoulis, N., Tao, Y., Vaitis, M.: Probabilistic Spatial Queries on Existentially Uncertain Data. In: Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD, pp. 400–417. Angra dos Reis, Brazil (2005)
18. Das, A., Gehrke, J., Riedewald, M.: Approximate Join Processing Over Data Streams. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD, pp. 40–51. San Diego, CA (2003)
19. Das, A., Gehrke, J., Riedewald, M.: Semantic Approximation of Data Stream Joins. IEEE Transactions on Knowledge and Data Engineering, TKDE **17**(1), 44–59 (2005)
20. 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)
21. 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)
22. Ghanem, T.M., Aref, W.G., Elmagarmid, A.K.: Exploiting Predicate-window Semantics over Data Streams . SIGMOD Record **35**(1), 3–8 (2006)
23. 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)
24. 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)
25. 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)
26. 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)
27. 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)
28. 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)
29. 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)
30. Jürgen Krämer, B.S.: PIPES - A Public Infrastructure for Processing and Exploring Streams. In: Proceedings of the ACM International Conference on Management of Data, SIGMOD, pp. 925–926. Paris, France (2004)
31. Kalashnikov, D.V., Prabhakar, S., Hambrusch, S.E.: Main Memory Evaluation of Monitoring Queries Over Moving Objects. Distributed and Parallel Databases **15**(2), 117–135 (2004)
32. Kalashnikov, D.V., Prabhakar, S., Hambrusch, S.E., Aref, W.G.: Efficient Evaluation of Continuous Range Queries on Moving Objects. In: Database and Expert Systems Applications, DEXA, pp. 731–740. Aix-en-Provence, France (2002)
33. 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, pp. 113–120 (2002)
34. Lazaridis, I., Porkaew, K., Mehrotra, S.: Dynamic Queries over Mobile Objects. In: Proceedings of the International Conference on Extending Database Technology, EDBT, pp. 269–286 (2002)
35. Lee, M.L., Hsu, W., Jensen, C.S., Cui, B., 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)
36. 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, pp. 49–60 (2002)
37. 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)
38. Mokbel, M.F., Aref, W.G.: PLACE: A Scalable Location-aware Database Server for Spatio-temporal Data Streams. IEEE Data Engineering Bulletin **28**(3), 3–10 (2005)
39. Mokbel, M.F., Xiong, X., Aref, W.G.: SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In: Proceedings of the ACM Inter-

national Conference on Management of Data, SIGMOD, pp. 443–454 (2004)
40. 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)
41. 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)
42. 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)
43. 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)
44. 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)
45. Pfoser, D., Jensen, C.S.: Capturing the Uncertainty of Moving-Object Representations. In: Proceedings of the International Symposium on Advances in Spatial Databases, SSD, pp. 111–132. Hong Kong (1999)
46. 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), 1124–1140 (2002)
47. Reiss, F., Hellerstein, J.M.: Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In: Proceedings of the International Conference on Data Engineering, ICDE, pp. 155–156. Tokyo, Japan (2005)
48. 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, pp. 331–342 (2000)
49. Song, Z., Roussopoulos, N.: Hashing Moving Objects. In: Proceedings of the International Conference on Mobile Data Management, MDM, pp. 161–172 (2001)
50. Song, Z., Roussopoulos, N.: K-Nearest Neighbor Search for Moving Query Point. In: Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD, pp. 79–96 (2001)
51. Tao, Y., Papadias, D., Shen, Q.: Continuous Nearest Neighbor Search. In: Proceedings of the International Conference on Very Large Data Bases, VLDB, pp. 287–298 (2002)
52. 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)
53. 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)
54. Trajcevski, G., Wolfson, O., Hinrichs, K., Chamberlain, S.: Managing Uncertainty in Moving Objects Databases. ACM Transactions on Database Systems , TODS **29**(3), 463–507 (2004)
55. Trajcevski, G., Wolfson, O., Zhang, F., Chamberlain, S.: The Geometry of Uncertainty in Moving Objects Databases. In: Proceedings of the International Conference on Extending Database Technology, EDBT, pp. 233–250. Prague, Czech Republic (2002)
56. Wolfson, O., Yin, H.: Accuracy and Resource Concumption in Tracking and Location Prediction. In: Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD, pp. 325–343. Santorini Island, Greece (2003)
57. 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)
58. 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)
59. Yu, X., Pu, K.Q., Koudas, N.: Monitoring K-Nearest Neighbor Queries Over Moving Objects. In: Proceedings of the International Conference on Data Engineering, ICDE, pp. 631–642. Tokyo, Japan (2005)
60. 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, pp. 443–454 (2003)
61. Zheng, B., Lee, D.L.: Semantic Caching in Location-Dependent Query Processing. In: Proceedings of the International Symposium on Advances in Spatial and Temporal Databases, SSTD, pp. 97–116 (2001)