Microsoft Reactive Framework Meets Spatio-Temporal Databases

Hossam Fattah[1] Abdeltawab M. Hendawi[2] Youying Shi[3]
Jayant Gupta[3] Mohamed Ali[3]

[1]Microsoft Corporation, WA, USA
hofattah@microsoft.com
[2]Department of Computer Science, University of Virginia, VA, USA
hendawi@cs.virginia.edu
[3]Center for Data Science, University of Washington, Tacoma, WA, USA
{*youyings, jayantg, mhali*}*@uw.edu*

***Abstract*—Microsoft SQL Server Spatial (*SqlSpatial*) Library is primarily developed to efficiently evaluate operations on stationary objects. When it comes to real-world applications that deal with moving objects which require real-time tracking and processing such as connected vehicles, here, the limitations float to the surface. Unfortunately, the *SqlSpatial* library has very limited operations in this domain. To overcome these limitations, this paper presents the *RxSpatial* library developed to real-time processing of spatio-temporal operations on moving objects. This paper introduces the front-end application programming interfaces (APIs) for developers to build their applications, and the back-end containing the streaming-fashion processing for the spatio-temporal operations. The superiority of the *RxSpatial* over the basic *SqlSpatial* is demonstrated throughout an extensive experimental evaluation on real and synthetic data sets.**

I. Introduction

Several spatial libraries have been developed by Microsoft, IBM and Oracle either as part of their database management systems or as stand-alone libraries. These libraries have substantially advanced the state of the art in geospatial computing and provided efficient implementations of various spatial functions such as intersection, distance and area over spatial objects. These libraries have been a perfect fit for an era where digital maps and Geographic Information Systems (GIS) were ramping up and booming. However, as time goes by, GIS systems have dramatically changed. It is no longer the case where cartographers take out their measuring tapes in the city or in wilderness to draw their maps, or to find the intersection/distance between different properties. Instead, GPS devices, hand-held devices and mobile technologies have pumped up the requirements on geospatial computing to a higher bar. Consider the scenario where a moving object (e.g., a vehicle) is on the road and a standing geofencing query is interested in continuously reporting the intersection of the vehicle's location and a large set of geofences (e.g., representing the boundaries of shopping malls). Because existing spatial libraries are designed for static objects, the intersection operation between the vehicle's location and each geofence is repeated over and over again from scratch every time a location update is received. The concept of incremental stream processing has not been considered yet in existing spatial libraries; neither on the user's interface side nor in the underlying spatiotemporal query processor.

In this paper, we address real time spatiotemporal stream query processing and introduce the *Reactive eXtension Spatial library*, *RxSpatial* for short. *RxSpatial* (1) blends the Microsoft SQL Server Spatial Library with the Microsoft Reactive Framework, (2) integrates spatial index structures for moving objects with the query processing pipeline, and (3) provides incremental spatial operations to evaluate queries in a data streaming fashion.

The proposed library provides a front end, which is a programming interface for developers who are familiar with the Microsoft Reactive Framework for .NET applications. This front end introduces new interfaces that are called *RxGeography* and *RxGeometry* to provide the real time versions of the *SQLGeography* and the *SQLGeometry* classes of the SQL Server Spatial Library, respectively. *SQLGeography* and *SQLGeometry* are the classes that encompass methods for spatial operations over geodetic and planar spatial data types, respectively.

Each one of the *RxGeography* and *RxGeometry* interfaces derives from the *IObservable* interface of the Reactive Framework. As a quick background, the *IObservable* interface represents a provider for push-based notifications. *IObservable* defines a *subscribe* method that multiple *observers* can subscribe to in order to get notifications of real time updates. The *RxGeography* and *RxGeometry* implement modified versions of the methods that are found in the *SQLGeography* and the *SQLGeometry* classes of the SQL Spatial Library, respectively. For example, the *SQLGeography* class implements a method called *SQLIntersects* to detect if two "stationary" geography objects intersect. Its real time *RxGeography* counterpart implements two methods: (1) a method called *RxIntersects* to continuously monitor and detect the intersection between a moving object (represented by an RxGeography object) and a stationary object (represented as SQLGeography), and (2) a method called *RxRxIntersects* to continuously monitor and detect the intersection of two moving objects (each one is represented an RxGeography).

Thanks to the Reactive Framework, the *RxSpatial* library provides strong support for asynchronous program development and event-based programs, with a smooth learning curve. The wide community of the .NET Developers would be able to represent asynchronous data streams of spatial objects using an IObservable¡T¿ data type [3], where T is a spatial data type (*SqlGeography* or *SqlGeometrty*). Hence, the *RxSpatial* library can be used to query the asynchronous data streams using LINQ [1], which is the Microsoft .NET way of embedding SQL queries.

In addition to the programming interface front-end, the *RxSpatial* library provides a back-end for processing spatial operations in a streaming fashion. The contribution at the back end level lies in the incremental stream processing of various geospatial operations. As an example, to detect the intersection of a moving object and a set of geofences, the intersection operation is not carried over from scratch between the moving object's location and all geofences upon the receipt of a location update. Instead, the intersection operation is incrementally evaluated to make use of as much computations as possible from the previous step. Moreover, the location, direction and speed of the moving object are used on top of a spatial index structure to efficiently process the intersection. Fortunately, we release the *RxSpatial* library equipped with the RUM-Tree [27], an efficient index structure and a modified *R-Tree* for moving objects.

Several timing-sensitive and real time geosptial applications would benefit from the *RxSpatial* Library. For example, in some child safety applications, children are tracked using GPS devices or smart phones. When they get near forbidden or dangerous area, the application needs to send an alert to parents on time. In the scenario of collaborative vehicle systems, a cluster of vehicles is on the way heading to their destinations, where every vehicle is a moving object to be tracked and controlled. If a vehicle is too far away from the cluster or too close to its neighboring vehicles, an action should be taken to adjust its speed and direction so that the vehicles maintain a reasonable communication distance and avoid collisions. In behaviorally targeted advertising campaigns, when a driver is close to a shopping mall, coupons, recommendations, and possibly a parking spot information need to reach the driver's smart device on time. In social networks, when a friend is nearby, it would be great to receive a notification of the friend's location as soon as he appears in the vicinity of his friend. In safe routing engines [14], the commuters' location information and the disastrous weather zone require real-time processing to generate a reasonable life saving emergency route to evacuate. In the scenario of tracking criminal activities [19], criminals and sex offenders under probation are forced to wear ankle bracelets that stream their GPS locations. Local authorities are expected to be notified immediately once a criminal steps into a school zone or steps out of a curfew zone predefined by law enforcement agents. In the context of this paper, we give examples and figures to illustrate the usability of the proposed *RxSpatial* library in two key domains: criminal activity tracking systems, and collaborative vehicle systems.

The rest of the paper is organized as follows. Section II gives a brief background of various technologies used within *RxSpatial*. Section III presents the library's user interfaces with its convenient APIs, while Section IV discusses the library's back end with its underlying data structures and algorithms. Section V describes some real-world application scenarios that would utilize the proposed library. Section VI provides experimental results that show the performance of the underlying real time query processing engine. The paper is concluded in Section VII.

II. Background

Before we present the details of *RxSpatial*, this section highlights the techniques, systems and libraries that are used within the implementation of *RxSpatial*. It also overviews several previous attempts to leverage the exiting Microsoft SQL Server Spatial Library with real time streaming capabilities.

A. Microsoft SQL Server Spatial Libraries

The Microsoft SQL Serevr Spatial Library provides an easy to use, robust, and high performance environment for persisting and analyzing spatial data. The library [2] provides data types for spatial objects such as points, lines and polygons, both in geometrical (planar) and geographical (geodetic) representations. The SQL Server Spatial Library adheres to the Simple Feature Access standard of the Open Geospatial Consortium [24]. In addition to the data type support, the library implements various spatial operations on top of these data types, e.g., intersection, distance, and area. The SQL Server Spatial Library has been widely used to provide spatial support for both SQL Server based and .NET based applications. However, the original library is tuned to address the needs of static non-moving object scenarios.

There has been few earlier attempts to add real time support for spatial operations in the SQL Server Spatial Library. An early attempt to integrate the Microsoft SQL Server Spatial Library with Microsoft StreamInsight [6] has been studied in [16]. Microsoft StreamInsight is a commercial data stream management system for processing long-running continuous queries over data streams. It has the ability to correlate stream data from multiple sources and to execute standing queries on a low-latency query processor to extract meaningful patterns and trends. The approach in [16] adopts the extensibility framework of StreamInsight [4] to implement an example set of spatiotemporal operations, e.g., KNN search, and range search; as user defined operators and integrates these operators with the query pipeline.

In contrast to the extensibility approach taken by [16], the work in [17] presents and contrasts two approaches to support real time spatiotemporal operations: the native approach and the extensibility approach. The native support approach deals with spatial attributes as first class citizens, reasons about the spatial properties of incoming events and, more interestingly, provides consistency guarantees over space as well as time.

The *GeoInsight* [10] system is another platform that blends the Microsoft SQL Server Spatial Library with StreamInsight to support geo-streaming applications, in general, and to support Intelligent Transportation Systems (*ITS*), in particular.

GeoInsight has been used to analyze the traffic information that is streamed out of loop detectors in Los Angeles county. Moreover, several spatiotemporal analytic queries that are executed in real time have been demonstrated in [18]. The demo scenario is based on the Microsoft Shuttle Service where GPS readings are generated and streamed by shuttles as they move around the Microsoft main campus in Redmond, WA. Tracking the activities of criminals and sex offenders in real time using ankle bracelets, then, performing behavioral mining on top of their GPS traces has been another interesting scenario for the SQL Server Spatial Library [19]. Given these previous attempts, we believe industry has reached a point where a full fledged real time spatiotemporal library is on track for development and production.

B. Microsoft Reactive Extension

The Reactive Manifesto [15] mentions that reactive systems should be responsive, resilient, elastic, and message driven. The Microsoft Reactive Extension (or the Reactive Frameowrk) [26] is a reactive programming extension that can be utilized within various .NET applications and, moreover, it goes hand in hand with the Microsoft LINQ [1] (Language Integrated Query). The Reactive Framework follows the Reactive Manifesto principles, and makes the development of reactive applications more convenient. Developers only need to focus on the business logic instead of dealing with various asynchronous issues.

The Reactive Framework provides two main interfaces: the *IObservable* and the *IObserver* interfaces. *IObservable* is a generic interface that wraps a source of message events (or a stream of events). *IObservable* provides a *Subscribe* method as part of the interface APIs to enable stream consumers to subscribe to the underlying stream source and to receive update notifications of incoming stream events.

The *IObserver* interface represents the stream consumer (or, in other words, the recipient of the stream events). An *IObserver* class subscribes to an *IObservable* class and adds callback functions so that the *IObserver* can be notified when a new event is generated from the stream source. These callback functions are called: *OnNext*, *OnError* and *OnEnd*, and are used to notify the stream consumer of new events, error messages and termination signal coming from the stream source, respectively.

The *RxSpatial* library utilizes the Microsoft Reactive Framework to provide a convenient programming interface for developers. *RxSpatial* implements two interfaces: the *RxGeography* which derives from *IObservable¡SQLGeography¿* and *RxGeometry* which derives from *IObservable¡Geometry¿*. Hence, *RxGeography* and *RxGeometry* are the data types that represent moving spatial objects in geodetic and planar domains, respectively. In other words, *RxGeography* and *RxGeometry* are the data types that represent the stream of location updates that are generated by moving objects as they roam the space. More details on the design and implementation of these data types are presented in Section IV.

It is worthy to mention that in addition to the .NET framework (and mainly the C# programming language), some other programming languages also support the reactive extensions. For example, *RxJava* [11] is a Java library that implements the *ReactiveX* API specifications [25]. In [7], geospatial analysis of taxicab data is implemented using *RxJava*.

C. Spatio-temporal Indexing

Various spatial libraries come equipped with spatial index structures for fast access of geospatial objects. Existing libraries are tuned for stationary non-moving objects and, hence, are equipped with index structures that are not well tuned for moving objects (e.g. R-Trees). The *R-Tree with Update Memos* (or the *RUM Tree*) is an enhancement of the original R-Tree. It is designed to reduce the cost of object updates in the R-Tree. Thus, it is a suitable choice for GIS applications with frequent location updates or moving objects. In R-Tree, the update of an object has two steps: an insertion of new entry and a deletion of the old entry. With the aid of update memo, RUM-Tree avoids the effort of finding and deleting the old entry during the update process; thus the cost of update becomes similar to the cost of insertion and, later on, a garbage collection mechanism handles the memory cleanup work.

When building an application using *RxSpatial*, the two major concerns are: (1) the scalability in terms of the number of moving objects and (2) the scalability in terms of location update frequency, that is, the number of location updates received per unit time from a single moving object. An interesting performance analysis study that compares the RUM-Tree against other candidate data structures is found in in [22]. This study provides large-scale experiments with 2 million to 20 million moving objects. The experiments provide a comparison between the RUM-Tree, the R*-Tree [23] and the Frequently Updated R-Tree (FUR-Tree) [13]. The performance measures mainly consist of two parts: (1) the cost of disk access (I/O cost) during the update and search operations and (2) the memory utilization.

From a memory utilization perspective, the R*-Tree's strategy aims for an optimized memory and costs less memory compared to other data structures. The memory cost of the FUR-Tree is comparable to the R*-tree. However, the RUM-Tree suffered from a higher memory cost due to its lazy garbage collection strategy and due to the existence of obsolete entries in the tree nodes. Therefore, RUM-Tree demands higher, yet still affordable, memory requirements.

During the update operation of an object's location, the R*-Tree needs to search for the object's old location using an expensive top-down tree navigation, followed by a deletion of this old entry and, then, the addition of an entry for the new object's location. The cost of the FUR-Tree update operation is positively correlated to the displacement distance of the object. The RUM-Tree's performance is relatively stable and is less than the R*-Tree and the FUR-Tree variants. During the search operation, the R*-Tree benefits from the continuous adjustment during top-down updates so it has the lowest search cost. FUR-Tree needs to check and process more nodes and is less performant compared to the R*-Tree. The RUM-tree also suffers from the presence of obsolete entries and, hence, the RUM-Tree may cost more time during the search operation.

Since the RUM-Tree performs significantly better in the update operation and have slightly lower performance in the search operation compared to other candidate data structures, the RUM-Tree becomes appealing under the frequently moving object settings. Also, with a minor sacrifice on the memory utilization side, *RxSpatial* adopts the RUM-Tree as its index structure choice for frequently moving objects.

III. The RxSpatial Library's Front End

As we mentioned earlier in Section II, the Microsoft Reactive Framework introduces the *IObservable* and the *IObserver* interfaces to represent stream sources (producers) and stream sinks (consumers), respectively. Moreover, the original Microsoft SQL Server Spatial Library introduces the *SQLGeography* and the *SQLGeometry* classes to encompass geospatial operations against geodetic and planar spatial objects, respectively. The *RxSpatial* library's front end utilizes the two paradigms and introduces the *RxGeography* and the *RxGeometry* classes to provide geospatial operations against geodetic and planar spatial "moving" objects, respectively. For brevity, we focus our discussion on the *RxGeography* class, and the same concepts would apply to the *RxGeometry* class.

As shown in Figure 1, the *RxGeography* class is designed to represent a moving object and to be attached to a geospatial stream source (e.g., a mobile device that streams GPS readings). Hence, *RxGeography* needs to be an observer of the location stream that is coming from the moving object and, consequently, implements the *IObserver¡T¿* generic interface with its three methods: *OnNext*, *OnComplete* and *OnError*. Note that *T* is the data type that represents the object's location and *RxSpatial* decided to use the static *SQLGeography* class to represent the object's location. This means that *RxGeography* is an *IObserver¡SQLGeography¿* and the object location can be a point as well as any spatial data type that *SQLGeography* can represent (i.e., point, line, polygon). The stream source or moving object would utilize the *IObserver's* methods to send its location updates. As an example, a phone app would call the *RxGeography.OnNext(newLocation)* method every time a new location is read by the GPS device. It would call the *RxGeography.OnNext(error)* and *RxGeography.OnComplete()* methods to signal an error or to terminate connection, respectively.

On the other side, *RxGeography* allows other observers to read the location of its underlying stream source. Hence, *RxGeography* also implements the *IObservable* interface with its *Subscribe* method. Stream consumers that are interested in the moving object's location subscribe to the RxGeography to receive location updates of that object. Figure 2 gives a class diagram for the main components of *RxSpatial*.

For every method (or spatial operation) provided by the *SQLGeography* class in the original SQL Spatial Library, the *RxGeography* in the proposed *RxSpatial* library provides two real-time streaming versions of this method. Let's consider the intersection operation as an example. While the *SQLGeography.STIntersects* method (in the SQLGeography class of the original library) detects the intersection between two static spatial objects, the *RxGeography* class implements the following two methods:

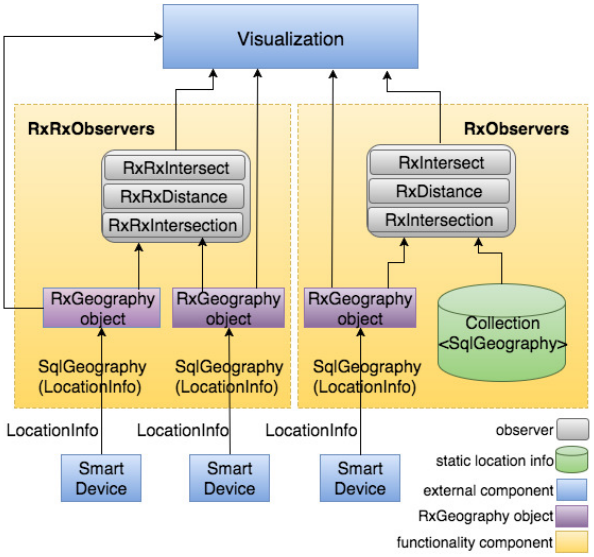


Fig. 1. An overview of the *RxSpatial* Library

- *RxGeography.RxIntersects*, which detects the intersection between a moving object and a static object.
- *RxGeography.RxRxIntersects*, which detects the intersection between two moving objects.

Note that an operation between a static object and a moving object is prefixed by a single "Rx", while an operation between two moving objects is prefixed by the double "RxRx"notation. Also, note that both *RxGeography.RxIntersects* and *RxGeography.RxRxIntersects* take two input parameters. To understand these parameters, let's first review the parameters of the *SQLGeography.STIntersects* that detects intersection between two static objects. *SQLGeography.STIntersects* takes a single parameter which is another static *SQLGeography* object and returns a boolean value that represents whether the two objects intersect or not. Example lines of code may look like:

```
SQLGeography o1, o1;
...
bool isIntersecting = o1.STIntersects(o2);
```

Both *o1* and *o2* in the above example are of type *SQLGeography*. The corresponding streaming version *RxGeography.RxIntersects* takes two parameters: (a) another static *SQLGeography* object and (b) an observer of type *IObserver¡bool¿*. Example lines of code may look like:

```
RxGeography mo1;
SQLGeography o2;
IObserver <bool> isIntersecting;
...
mo1.RxIntersects(o2, isIntersecting)
```

In the above example, *mo1* is a moving object represented by an *RxGeography* data type, *o2* is a static *SQLGeography* object and *isIntersecting* is an *IObserver* of type boolean. Every time the moving object updates its location, the intersection operation's result is evaluated and the boolean result is pushed as a notification to the *isIntersecting* observer that monitors the output stream. Note that, in contrast to the single boolean

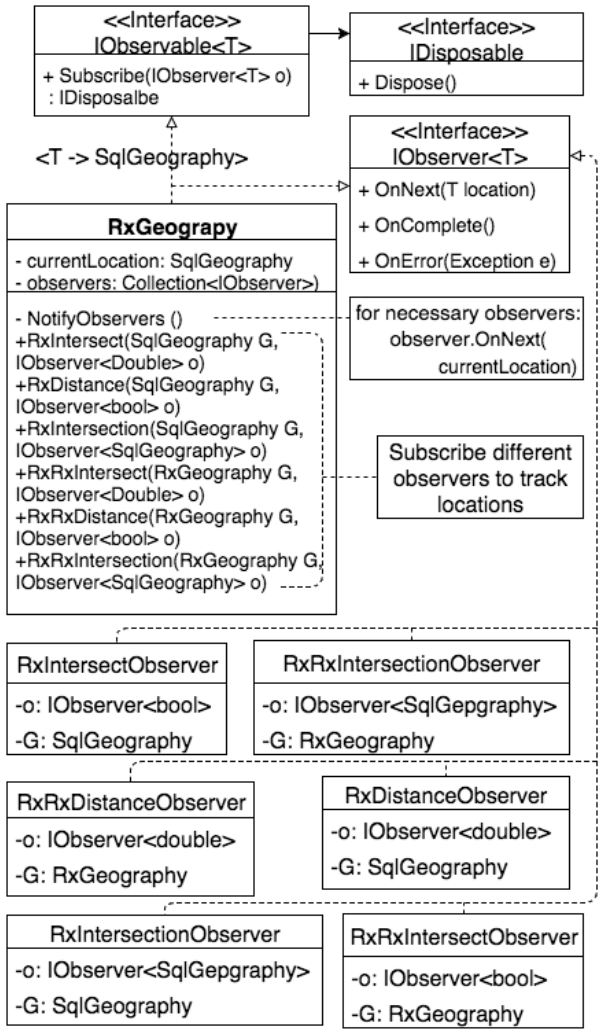


Fig. 2. Class Diagram For the Main Components of *RxSpatial*

.

output value of intersecting two static objects, the output of the *RxIntersects* operation is a stream of boolean values.

The *RxGeography.RxRxIntersects* operation takes also two parameters: another moving object represented by another *RxGeography* object and an observer of type *IObserver¡bool¿*. Example lines of code may look like:

RxGeography mo1, mo2;
IObserver<bool> isIntersecting;
...
mo1.RxIntersects(mo2, isIntersecting),

where *mo1* is a moving object represented by an *RxGeography* data type, *mo2* is another moving *RxGeography* object and *isIntersecting* is an *IObservable* of type boolean. Figure 2 illustrates few methods implemented by the *RxGeography* class as well as the data types that represent the various *IObserver* data types that hold the output stream of various geospatial operations.

	Single global index	Per object index
Main memory		X
Update time		X
Search time	X	
Scalability	X	

TABLE I. COMPARISON OF INDEXING ALTERNATIVES

IV. THE RXSPATIAL LIBRARY'S BACK END

While the *RxSpatial*'s front end provides the programming convenience for developers, the *RxSpatial*'s back end is responsible for the efficient processing of various spatial operations. While *RxSpatial* aims at fixing and stabilizing the user's APIs through which a developer writes his application, we believe that enhancing the back end is a long term process of incubating existing/evolving research directions and optimizing the performance of the chosen algorithms. This section describes the various research directions that are on the radar of *RxSpatial* and draws the roadmap for future versions of the library. This section also discusses various implementation alternatives and lists their pros and cons. However, this section and the experimental section (Section VI) scope down the discussion to the choices made by the first version of *RxSpatial* and leave a lot of room for improvement in future versions.

As we mentioned earlier, existing spatial libraries are designed for static non-moving objects. *RxSpatial* is designed for real time processing and is tuned for incremental query processing to fit in the data streaming paradigm. To achieve this goal, *RxSpatial* provides a general framework where various spatiotemporal index structures and incremental processing techniques can be plugged in. More specifically, *RxSpatial* aims at integrating the following research directions within the query pipeline:

1) Spatial index support for moving objects, where an index is used to track the moving object's current location.
2) Incremental query processing, where an incremental spatial join algorithms are utilized to produce early and low latency results upon the receipt of a location update.
3) Search space pruning using locality, speed and direction, where a moving object joins with other static or moving objects in the vicinity taking into consideration the speed and direction of the moving object to perform caching along the object's expected trajectory.

The current version of *RxSpatial* focuses on the first direction, that is, integrating spatiotemporal index structure for moving objects within the reactive query processing pipeline. In this paper, we study the feasibility and the impact of integrating various index structures (e.g., Grid Files, R-Trees, RUM-Trees) on the performance of the library. Future versions of the library are expected to tackle the directions of incremental query and space pruning and to integrate various techniques [9], [12], [21] within the library.

In a typical workload settings, we expect a large number of moving objects to be tracked, and we also expect a large number of spatial operations (e.g., distance and intersection) to take place among these objects. The performance of these spatial operations would benefit from indexing the current

locations of moving objects. Imagine that we draw a graph (call it, the *observability graph*), where the nodes are the moving objects and an edge in this graph represents a binary spatial operation between two moving objects. We argue that the density of connected components in this graph would highly impact the design choice of how to index the moving objects.

We highlight two "extreme" design alternatives for indexing the moving objects as follows:

- **Single global spatial index,** that keeps track of all moving objects under consideration by the system in one global index structure.
- **Per moving object spatial index,** that is every moving object keeps track of all its subscribers ($list(s_1, s_2, ...s_n)$) in a separate index owned and maintained by that moving object.

Table I summarizes the pros and cons of each alternative. A single index for all moving objects would be efficient in its memory footprint because every object is tracked and indexed only once in a single index that contains all moving objects. On the other side, maintaining an index per moving object means that a moving object, that is involved in n spatial operations with n other moving objects, will be inserted and tracked in all the n indexes maintained by these moving objects. Consequently, the insertion and update cost is lower in the single global index approach. However, a global single index would suffer from bad performance in processing various spatial operations because, with the movement of an object, a big index structure has to be searched to retrieve a small subset of moving objects that are subscribed or are involved in a spatial operation with that moving object. A per moving object index structure that indexes only subscribed objects would be efficient in processing various spatial operations and would scale in terms of the number of moving objects and the number of stream observers.

The current version favors scalability and adopts the *per moving object index structures*. However, future versions of the *RxSpatial* library are expected to invest in a hybrid approach that balances the above two extremes and to utilize the density of connected components in the *observability graph*. Clusters or subsets of moving objects that are identified to be densely connected subgraphs are declared to be related and to be collectively tracked in a separate spatial index. Note that the hybrid approach is not part of the *RxSpatial* current version. Experiments in Section VI are limited to the current version with its *per moving object index structures* choice.

A. Rum-Tree Within RxSpatial

In this section, we demonstrate how the RUM-Tree index is employed inside our *RxSpatial* library. We do this through an illustrative example that consists of a RUM-Tree and three GPS-enabled cell phones of green, blue and orange colors, Figures 3. Those phones represent three different moving objects.

Initially, Figures 3(a) demonstrates how observers are organized in the RUM-Tree inside the *RxSpatial* library. The space is partitioned into three Rectangles: N_1, N_2 and N_3; location 1 and 2 are in N_1. location 3, 4 and 5 are in N_2. Location 6, 7 and 8 are in N_3. In Figures 3(b), before time t0, the initial position of green cell phone is in location 8 and of blue cell phone is in location 1. Location could be a node in a road network or a point of interest (POI). The assumption is that blue cell phone was added at time t-1 and green cell phone was added at time t-2. None of these cell phones ever moved to another place since added to the RUM-Tree. Therefore in the updated memo, the record of green cell phone is: cnt=1, which denotes only 1 object in the RUM-Tree, and timestamp=t-2, which means the latest timestamp that green cell phone is added is t-2. The record of blue cell phone is: cnt=1, timestamp=t-1. Then, the orange cell phone is added to position 4 at t0 in Figures 3(b). At location 4 of the R-Tree, the object of orange cell phone is added. Then, in the updated memo, the record of orange cell phone is inserted with cnt=1, timestamp=t0.

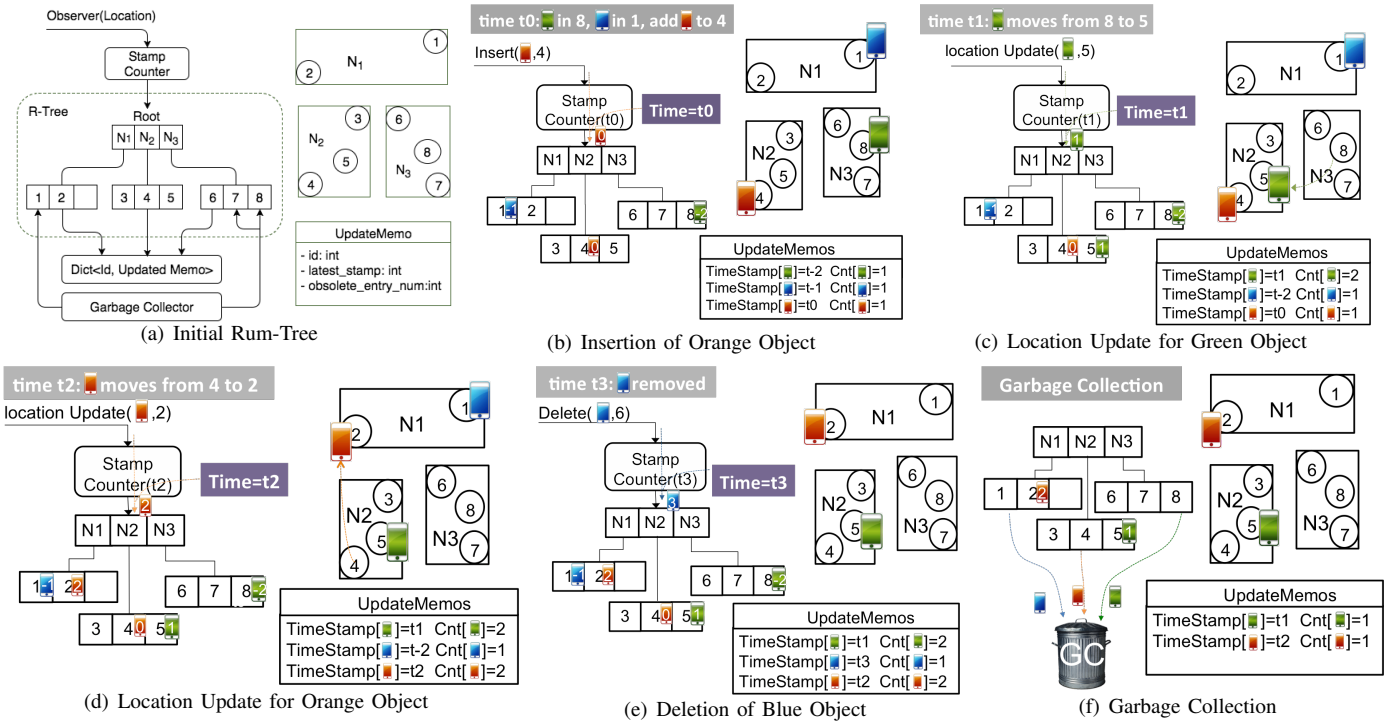
In Figures 3(c), the green cell phone moves from location 8 to location 4 at time t1. Thus, its corresponding entry in updated memos is: cnt=2, timestamp=t1. The old object in 8 is now obsolete as its present timestamp is t-2 which is older than its latest timestamp in the updated memo t1. In Figures 3(d), the orange cell phone moves from location 4 to location 2 at time t2. Its corresponding entry in the updated memo is: cnt=2, timestamp=t2. The old object in location 4 is now obsolete as its timestamp t0 is older than its latest timestamp in the updated memo t2. In Figures 3(e), the blue cell phone is removed from the entire RUM-Tree. Then its corresponding entry in the updated memo is: cnt=1, timestamp=t3. The count did not increase, only the timestamp increased. Thus, the only object of blue cell phone in the RUM-Tree becomes obsolete.

In Figures 3(f), a round of garbage collection towards the three cell phones is performed. Each time the garbage collector checks an object in R-Tree node, it first checks its timestamp and then checks its corresponding timestamp in the updated memo. An object is treated as an obsolete object, if its timestamp is earlier than its timestamp in the updated memo entry. Each time an object is removed from the R-Tree, the cnt field in its entry in the updated memo reduces by 1. The obsolete objects of the three cell phones are now removed from the RUM-Tree. For the blue cell phone which is removed, the cnt field is 0, which will make its entry to be deleted from the updated memo.The count field, cnt, of the other two cell phones reduced to 1. The *RxSpatial* sends notifications based on the latest non-obsolete objects location.

V. Real-World Applications

In this section, we provide example applications that could be built using the *Reactive Spatial Library*. We basically consider two scenarios: (1) criminal activity tracking systems and (2) collaborative vehicle systems. These two types of systems are considered for the following reasons:

1) Both application scenarios are very time sensitive. Response time is crucial both in preventing future criminal offenses and in controlling the movement of vehicles in the right direction and speed.
2) Both application scenarios require the usage of intersection and distance operations (a) between moving objects and static polygons, or (b) between moving objects and other moving objects.



(a) Initial Rum-Tree (b) Insertion of Orange Object (c) Location Update for Green Object

(d) Location Update for Orange Object (e) Deletion of Blue Object (f) Garbage Collection

Fig. 3. Integrating RUM-Tree (R-Tree With Updated Memo) Inside *RxSpatial*. Best Viewed in Color.

3) The two types of systems examine the *RxSpatial* functionalities under different settings. While criminal tracking systems are characterized by a large set of low-frequency streaming sources, collaborative vehicle systems are characterized by a relatively smaller set of high-frequency streaming sources.

A. *Monitoring Criminal Activity*

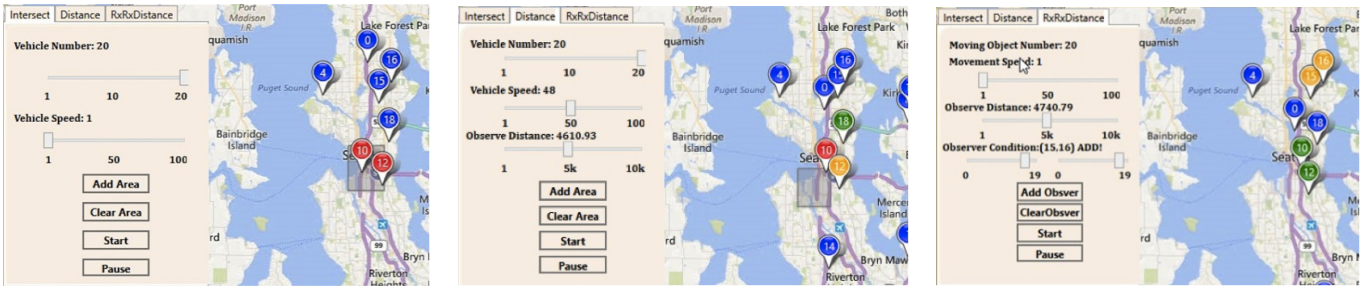
According to the criminal justice system, court orders may require a criminal on bail or under probation to wear an ankle bracelet. This ankle bracelet continuously streams the location of that criminal or offender to a monitoring system. Each offender with a tracking device is assigned a designated confinement zone (e.g., a city or a county) to which the offender is detained to, and a set of restricted zones (e.g. school zones) to which the offender is obliged to stay away from. Also, some offenders are not allowed to meet up with each other to reduce the possibility of forming a new gang or conspiring for a new offense. For more details on this application scenario, the reader is referred to [19].

To get the reader familiar with the user interface (UI) and the system simulator depicted in Figure 4, we list the main functionalities of the UI as follows:

- Each moving object is represented by a pushpin and is assigned a number for identification. A blue pushpin means that the moving object did not trigger any alert, while a pushpin with a different color means that the moving object has triggered an alert.
- On the left side of each figure is the control panel. "Start" and "Pause" buttons are used to start and pause the movement of objects, respectively.
- The "Speed" and the "Number of Objects" sliding bars control the speed and the number of moving objects that are considered in the simulation.
- The "Add Area" and "Clear Area" buttons are used to specify confinement or curfew zones.
- Based on a specific scenario, more sliding bars are utilized to specify various thresholds including the threshold on the distance at which two criminals are to be considered in close proximity and, consequently, an alert is triggered.

Figure 4(a) shows the scenario of tracking a criminal entering a restricted zone or leaving a confinement zone. Under the cover, this scenario requires the intersection of the moving criminal's location with the set of stationary polygons that represent the restricted/confinment zones. The *RxIntersects* method is utilized to compute the intersection between the moving object and the stationary polygons. From the figure, we see that offenders 10 and 12 entered a restricted zone triggering an alert (represented by red colored pushpins).

Figure 4(b) shows the scenario where the *RxDistance* method is used to calculate, in real time, the proximity between an offender and a polygon. Offender 12 gets into the "watching distance" of the restricted region and, hence, triggers a warning (represented by a yellow colored pushpin). Note that offender 12 has not entered the restricted zone yet. However, the distance between him and the restricted zone is below a specific "watching distance" threshold. Hence, a warning is



(a) The restricted/confinement zone alert (b) The restricted proximity to a zone warning (c) The mutual proximity among criminals scenario

Fig. 4. Using the *RxSpatial* library in Criminal Activity Tracking Systems. Best Viewed in Color

raised instead of an alert.

While the previous two scenarios involve a spatial operation between a moving object and a set of stationary objects, Figure 4(c) shows a scenario that involves a spatial operation among two or more moving objects. The *RxRxDistance* method is used to compute proximity between pairs of offenders. Once a moving offender is within a specific distance within any other moving offender, a warning is initiated signaling that the two offenders are likely to meet up. In the figure, offender 10 and offender 12 are close enough and are likely to meet up. Hence, a "restricted meetup warning" is raised up by the system. The same warning also applies to offender 15 and offender 16 in the same figure.

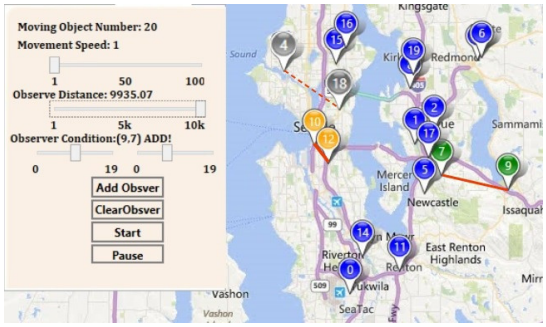


Fig. 5. Distance Monitoring in Collaborative Vehicle Systems

B. Collaborative Vehicle System

In collaborative vehicle systems, cars within a certain distance from each other can share data together. For example, the leading car at the front could share the route status with the following cars at the back. The leading car may even decide to take an alternative route and to send back a message indicating that a traffic congestion or an accident has occurred on the road. In these collaborative scenarios, the connection between two cars is stable if and only if the two cars are within a specific distance, called the *valid communication distance*.

Another example is that, when driving on a highway, the collaborating cars need to ensure a *minimum safe distance* within each other to avoid collision. Meanwhile, they need to maintain a valid communication distance that allows for the direct wireless communication between each other. When a car is too close (compromising the safe distance), or too far away (exceeding the valid communication distance), an action needs to be taken to adjust the speed of this car.

In Figure 5, every car is a moving object that is tracked by the system. The collaboration status is tracked between cars 4 and 18, cars 10 and 12, and cars 7 and 9. The dotted line between cars 4 and 18, and the gray color of the pushpins show that the cars 4 and 18 are watching the movement of each other, but the connection between them is not strong enough as the distance exceeds a limit. The solid thick line between cars 10 and 12, and the yellow color of pushpins show that they are within a stable working distance. The thin line between cars 7 and 9, and the green color of the pushpins show that they are within a working distance but are likely to disconnect if they get further apart. The blue color of other cars show that they did not intend to watch the movement of any cars so they are not trying to connect and, hence, there is no need to audit the connections. Under the cover, a self join *RxRxDistance* operation is continuously evaluating the pairwise distances between every single pairs of cars. The self join *RxRxDistance* operation becomes even more challenging given the high rate of the input streams that are received by the system and that represent the vehicle location updates.

VI. Experimental Evaluation

In this section, we evaluate the performance of the *RxSpatial* Library against the traditional SQL Server Spatial Library. Moreover, we evaluate the performance of different implementations within the *RxSpatial* Library. More specifically, we integrate different spatial index structures within the library to store and retrieve the current locations of moving objects. The implementation of all experiments are in C# in Visual Studio 2013, Windows 8.1. A data set of real trajectories collected by Microsoft Researchers around the Seattle area is used to test the validity of the implementation [5], [8]. However, for performance and scalability evaluation, different data sets for moving objects are generated using the *MNTG* traffic generator [20].

The performance parameters under evaluation include:

- **The number of stream data sources**, which represents the number of moving objects.

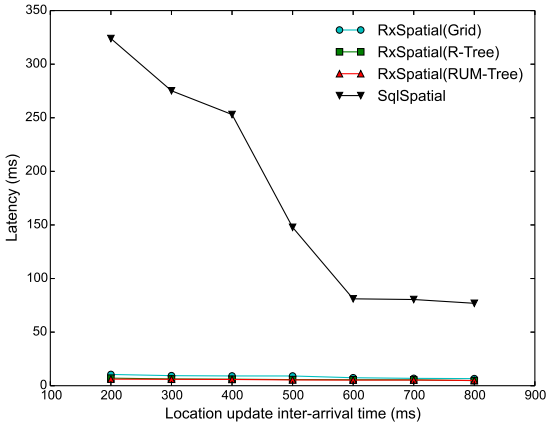
- **The location update inter-arrival time**, which represents the time difference between consecutive location updates from a moving object.
- **The number of stream observers,** which represents the numbers of objects that subscribe to a stream source or that subscribe to the outcome of a spatio-temporal operation.

The performance measures considered in this work:

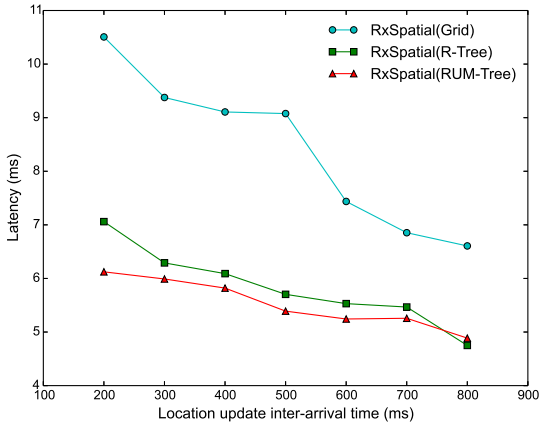
- **Maximum load**, which represents the maximum number of moving objects that can be tracked and analyzed using spatio-temporal operations, without dropping any incoming location update.
- **Latency**, which represents the average processing time per location update. The time is measured from receiving of update at the system's input buffers till the update's result is propagated to the output stream.
- **Number of dropped location updates**, which represents the number of incoming location updates dropped by the system under heavy system loads.

The performance is tested against the following variants of the Microsoft SQL Server Spatial Library and the proposed *RxSpatial* library:

- **The original Microsoft SQL Server Spatial**, where there is no native support for incremental streaming operation. Each spatial operation is carried over from scratch upon the receipt of every location update.
- **The RxSpatial using a Grid index structure**, where the space is divided into a rectangular grid, and the space indexed using a *Grid File* like index structure. This approach is characterized by simplicity in the implementation of the chosen index structure.
- **The RxSpatial using an R-Tree index structure**, where the current locations of moving objects are indexed using the original R-Tree.
- **The RxSpatial using a RUM-Tree index structure**, where the current locations of moving objects are indexed using an R-Tree with update memo (or a RUM-Tree) as described in Section II and Section IV.



(a) Comparison between the original Microsoft SQL Server Spatial Library and proposed RxSpatial variants



(b) Comparison among the proposed RxSpatial variants

Fig. 6. The effect of location update inter-arrival time on the system's latency

A. *Scalability in terms of the location update inter-arrival time*

This section evaluates the scalability of the various implementation variants in terms of the location update inter-arrival time. Figure 6 evaluates the effect of the location update inter-arrival time on the average processing latency per location update. The x-axis denotes the inter-arrival time between two location updates, while the y-axis represents the average latency. This experiment spawns 2000 *RxIntersects* operations on top of a single moving object. This experiment mimics the scenario of evaluating the intersection of a moving vehicle's location against 2000 geofences that represent the polygons of shopping malls and points of interest.

Figure 6(a) clearly reveals that the *RxSpatial* variants outperform the original SQL Server Spatial Library. Further, at the lowest inter-arrival time of 200 (fastest input rate) RUM-Tree shows around 55 times better computational performance over the base-line SQL Server Spatial indexing. Figure 6(b) compares only the *RxSpatial* variants and clearly shows that RUM-Tree implementation exhibits better performance under low inter-arrival time (high input rate). Our results show that use of RUM-Tree for indexing results 13.3% and 41.7% decrease in latency compared to R-Tree and GRID based indexing. As inter-arrival time gets large (i.e., the stream rate gets lower), both the R-Tree and RUM-Tree implementations exhibit similar performance due to the abundance of processing cycles.

Figure 7 evaluates inter-arrival time between location updates against the the maximum number of moving objects without dropping any incoming update. Our results show that lesser moving objects can be supported for smaller update inter-arrival time. This is due to an increased load on the system resulting in lesser processing cycles. Among the variants SQL Server Spatial Library has the lowest throughput. Amongst the *RxSpatial* variants either Tree based implementations perform better than the simple grid file implementation.

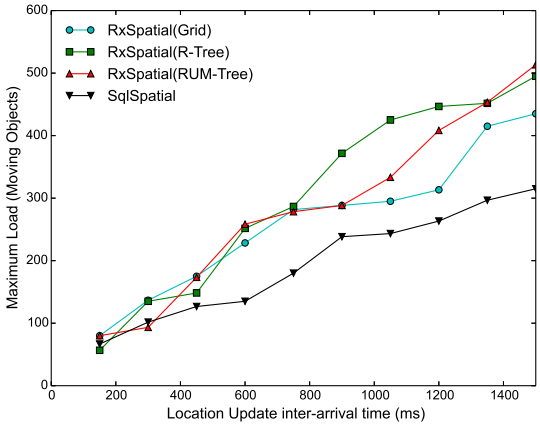


Fig. 7. The effect of location update inter-arrival time on the system's maximum load

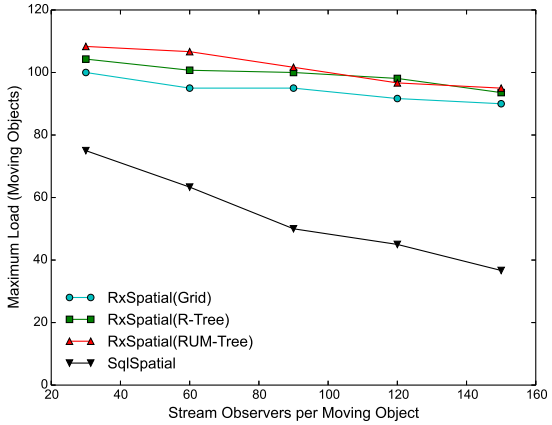
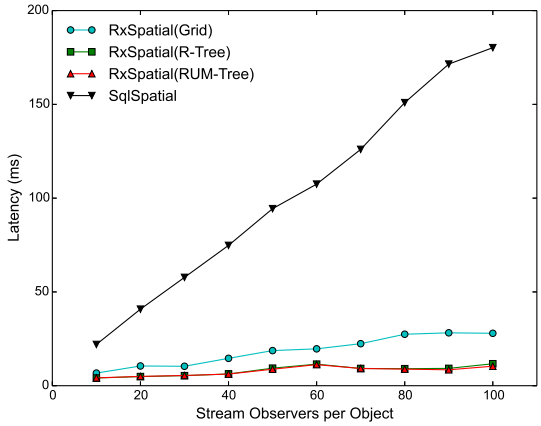
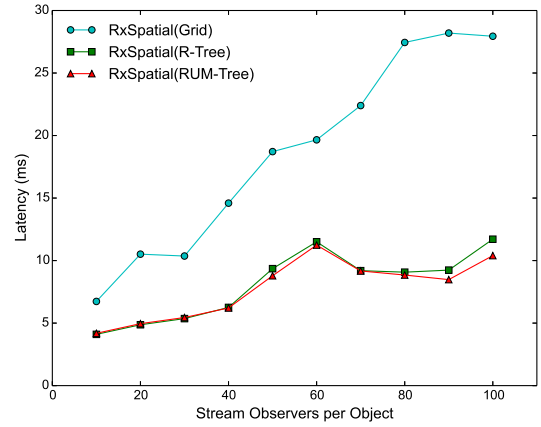


Fig. 8. The effect of the number of stream observers on the system's maximum load



(a) Comparison between the original Microsoft SQL Server Spatial Library and proposed RxSpatial variants



(b) Comparison among the proposed RxSpatial variants

Fig. 9. The effect of the number of stream observers on latency

Overall, RUM-Tree shows an improvement of 3.7%, 18% and 63% over R-Tree, Grid based and List based (SQLSpatial) indexing respectively in terms of maximum load, when update inter-arrival time is kept at 1500ms.

B. Scalability in terms of the number of stream observers

Figure 8 evaluates the effect of the number of stream observers that subscribe to a spatial operation on the system's maximum load. Imagine the scenario where a group of cars that follow a single leading car are observing the location updates of that leading car, and/or the distance between them to that leading car. This scenario features a large number of stream observers that observe the generated location updates of a stream source (moving object). The x-axis is the number of observers subscribing to a single *RxGeography* object or to the output stream of a single operation (*RxIntersection, RxDistance, RxRxIntersection, or RxRxDistance* operation. The y-axis is the system's maximum load (number of moving objects) the system can support before dropping incoming location updates.

From the figure, the more observers that subscribe to a stream source, the lower the number of incoming stream sources the system can support. When the number of observers grows from 30 to 150, the throughput drops by around 10 input stream sources for Grid (from 100 to 90) and Tree (from 105 to 95) based index structures. However, the throughput drops by around 40 (from 80 to 40) stream sources for the original SQL Server Spatial Library. This experiment shows the resiliency of the *RxSpatial* variants to increasing the number of stream observers.

Experiments shown in Figure 9 evaluate the effect of the number of observers on the processing latency. Each observable object is required to send location updates to all the observers subscribed to it. This affects the latency of the

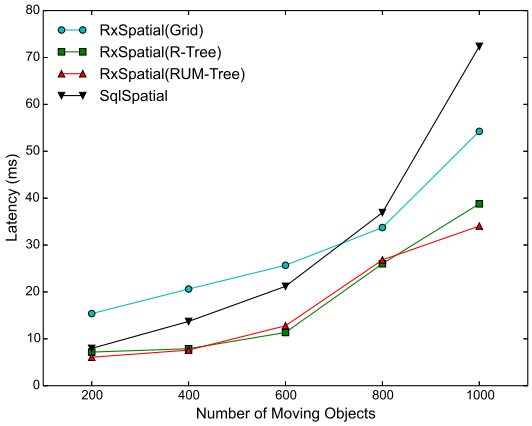


Fig. 10. The effect of the number of Moving Objects on the system's latency

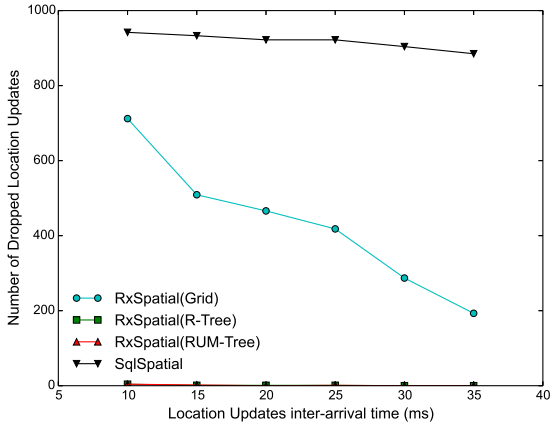


Fig. 11. The effect of location update inter-arrival time on the number of dropped location updates

location updates to be sent out by the moving object. Figure 9(a) illustrates the performance of the original SQL Server Spatial Library and all *RxSpatial* variants. The figure clearly shows that List based indexing (SQLSpatial) is time consuming compared to *RxSpatial* variants. Further, Figure 9(b) zooms in to consider the *RxSpatial* variants. The results show that Tree-based indexing performs better thatn Grid based indexing. Overall, for 100 observers RUM-Tree shows 11.2%, 62.81% and 94.22% decrease in latency over R-Tree, Grid and List (SQLSpatial) indexing respectively.

C. Variation in System Latency with change in the number of input streams

Figure 10 evaluates the performance of the system as we increase the number of moving objects the system is observing. Each moving objec denotes an input stream that sends location updates at regular intervals. The y-axis is the latency as measured by the average processing time per location update. The figure show that under light loads, (around 200 moving objects), the average response time is comparable across all techniques. However, as the number of objects increases, the *RxSpatial* library variants outperform its corresponding implementations in the original SQL Server Spatial Library. Overall, for input streams of 1K Moving Objects, RUM-Tree shows 12.2%, 37.22% and 54.95% decrease in latency over R-Tree, Grid and original *SqlSpatial*, respectively.

D. Evaluating the system's load shedding behavior under heavy loads

In this section, we stress test the system with a high input rate stream of location updates to evaluate the load shedding behavior of various implementation variants under consideration. Similar to Section VI-A, this experiment registers 2000 *RxIntersects* operations on top of a single moving object. However, the stream inter-arrival time between consecutive location updates is significantly reduced (in the range of 5

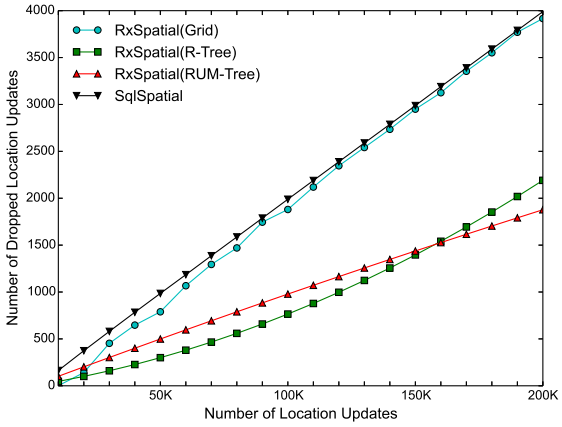


Fig. 12. The effect of the number of moving objects on the number of dropped location updates

- 40 ms) to simulate high input rate streams. For smaller inter-arrival time between consecutive location updates (higher input rates), the system's input buffer gets full and the system has to shed the load by dropping incoming location updates in order to cope up with the high rate of incoming location updates.

Figure 11 shows the load shedding behavior of all variants against the location update inter-arrival time. As expected, the original SQL Server Spatial library shows a higher number of dropped location updates because of the high processing cost the system takes to process a single location update. The Grid implementation of *RxSpatial*, although better than the original SQL Server Spatial Library, showed a high number of dropped location updates. This behavior is attributed to the fact that the Grid index degenerates to a linear list, or to a small number of linear lists condensed in few cells with the non-uniformity in distribution of indexed objects over space. The R-

Tree and RUM-Tree variants showed resiliency in terms of load shedding. With an increase of inter-arrival time to over 15ms (for R-Tree) and to over 10ms (for RUM-Tree), no dropping of incoming location updates has been observed. Hence, the R-Tree and RUM-Tree variants are efficient enough to cope up with all incoming location updates at these rates.

Figure 12 shows the number of Dropped Location Updates as we increase the number of Location Updates sent by moving objects. In this experiment each moving object sends 200 requests to 50 Observers at an interval of 50ms. So, each moving object is sending a total of 10,000 (200 * 50) location updates. We increased the number of moving objects from 1 (10K updates) to 20 (200K Updates) and observed the results for each data structure. We kept the system's input buffer to 10. We find that the R-Tree and RUM-Tree variants dropped between 190%-200% less updates from incoming stream than SqlSpatial and Grid based data structures for 200K location updates. These observations show that Tree based data-structures can handle more incoming updates compared to Grid based and List based data structures.

To summarize the performance results across the previous subsections, the experimental study shows that the performance of the *RxSpatial* library with its streaming capacities outperforms the original non-streaming *SqlSpatial* libraries. It also shows that investing in integrating the proper index structures that are tuned to handle moving objects with frequent location updates is absolutely worthy and justifies the implementation overhead, specially under heavy workloads. Both the R-Tree and RUM-Tree implementations show better performance than the simple Grid index, with the RUM-Tree showing better performance than the original R-Tree under high input rates.

VII. Conclusion

In this paper, we presented the *RxSpatial* library with its front end that enables developers, who are familiar with the Microsoft .NET Reactive Framework, to build applications with real-time spatiotemporal capabilities. While the *RxSpatial* Library's front end provides the programming convenience, the library's back end provides the efficiency in stream query processing over spatial moving objects. *RxSpatial* provides incremental evaluation of spatial operations and utilizes efficient index structures for moving objects. As an example, we discussed the integration of Grid files, R-Trees, and RUM-Trees inside *RxSpatial*: (1) to index moving objects, and (2) to incrementally evaluate streams of location updates. We described real-world scenarios and conducted experimental study to examine the performance of *RxSpatial*.

References

[1] V. S. 2015. LINQ (Language-Integrated Query). https://msdn.microsoft.com/en-us/library/bb397926.aspx, Dec. 2015.

[2] S. S. 2016. Microsoft SQL Server Spatial Libraries. https://msdn.microsoft.com/en-us/library/bb933790.aspx, Dec. 2015.

[3] N. F. 4.6 and 4.5. IObservable Generic Interface. https://msdn.microsoft.com/library/dd990377.aspx, Dec. 2015.

[4] M. Ali, B. Chandramouli, J. Goldstein, and R. Schindlauer. The extensibility framework in microsoft streaminsight. In *ICDE*, 2011.

[5] M. Ali, J. Krumm, and A. Teredesai. ACM SIGSPATIAL GIS Cup 2012. In *ACM SIGSPATIAL GIS*, pages 597–600, California, USA, Nov. 2012.

[6] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR*, 2007.

[7] D. Christoph, P. Thomas, and J. Hans-Arno. Geospatial event analytics leveraging reactive programming. In *ACM DEBS*, pages 324–325, Oslo, Norway, June 2015.

[8] A. M. Hendawi, J. Bao, M. F. Mokbel, and M. Ali. Predictive Tree: An Efficient Index for Predictive Queries On Road Networks. In *ICDE*, Seoul, South Korea, Apr. 2015.

[9] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, pages 237–248, 1998.

[10] K. S. Jalal, D. Ugur, A. Mohamed, A. Afsin, and S. Cyrus. Geospatial stream query processing using Microsoft SQL Server StreamInsight. *VLDB*, 3(1-2):1537–1540, Sept. 2010.

[11] D. Karnok. RxJava. https://github.com/ReactiveX/RxJava, Dec. 2015.

[12] O. Kwon and K. Li. Progressive spatial join for polygon data stream. In *GIS*, pages 389–392, 2011.

[13] L. M. Li, H. Wynne, J. C. S, C. Bin, and T. K. Lik. Supporting Frequent Updates in R-trees: A Bottom-up Approach. In *VLDB*, pages 608–619, Berlin, Germany, Sept. 2003.

[14] Y. Li, S. George, C. Apfelbeck, A. M. Hendawi, D. Hazel, A. Teredesai, and M. Ali. Routing Service With Real World Severe Weather. In *ACM SIGSPATIAL GIS*, Texas, USA, Nov. 2014.

[15] R. Manifesto. Reactive Systems. http://www.reactivemanifesto.org/, Dec. 2015.

[16] J. Miller, M. Raymond, J. Archer, S. Adem, L. Hansel, S. Konda, M. Luti, Y. Zhao, A. Teredesai, and M. Ali. An Extensibility Approach for Spatio-temporal Stream Processing using Microsoft StreamInsight. In *SSTD*, Minneapolis, MN, USA, Aug. 2011.

[17] A. Mohamed, C. Badrish, S. Balan, and K. Raman. Spatio-temporal stream processing in microsoft streaminsight. *Data Engineering*, 33(2):69, June 2010.

[18] A. Mohamed, C. Badrish, R. B. S, and K. Ed. Real-time spatio-temporal analytics using Microsoft StreamInsight. In *ACM SIGSPATIAL GIS*, pages 542–543, California, USA, Nov. 2010.

[19] D. Mohammed, F. Olajumoke, J. Lars, S. Niko, Y. Brett, N. Joe, and A. Mohamed. Safe step: a real-time GPS tracking and analysis system for criminal activities using ankle bracelets. In *ACM SIGSPATIAL GIS*, pages 512–515, Florida, USA, Nov. 2013.

[20] M. F. Mokbel, L. Alarabi, J. Bao, A. Eldawy, A. Magdy, M. Sarwat, E. Waytas, and S. Yackel. MNTG: An Extensible Web-based Traffic Generator. In *SSTD*, pages 38–55, Munich, Germany, Aug. 2013.

[21] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.

[22] S. Y. N, X. Xiaopeng, and A. W. G. The RUM-tree: supporting frequent updates in R-trees using memos. *VLDB Journal*, 18(3):719–738, 2009.

[23] B. Norbert, K. Hans-Peter, S. Ralf, and S. Bernhard. The R*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, New Jersey, USA, May 1990.

[24] OGC. Open Geospatial Consortium. http://www.opengeospatial.org, Dec. 2015.

[25] ReactiveX. An API for asynchronous programming with observable streams. http://reactivex.io, Dec. 2015.

[26] RX. Microsoft Reactive Extensions. https://msdn.microsoft.com/en-us/data/gg577609.aspx, Dec. 2015.

[27] X. Xiaopeng and A. W. G. R-trees with update memos. In *ICDE*, pages 22–22, Georgia, USA, Apr. 2006.