RxSpatial: Reactive Spatial Library for Real-Time Location Tracking and Processing

Youying Shi[1], Abdeltawab M. Hendawi[2], Hossam Fattah[3], Mohamed Ali[1]
[1]Center for Data Science, Institute of Technology, University of Washington, WA, USA
[1]{youyings, mhali}@uw.edu
[2]Department of Computer Science, University of Virginia, VA, USA
hendawi@virginia.edu
[3]Microsoft Corporation, WA, USA
hofattah@microsoft.com

ABSTRACT

Current commercial spatial libraries have implemented strong support for functionalities like intersection, distance, and area of various stationary geospatial objects. The missing point is the support for moving objects. Performing moving object real-time location tracking and computation on the server side of GIS applications is challenging because of the large numbers of moving objects to be tracked, the time complexity of spatial computation, and the real-time requirements. In this Demo, we present the *RxSpatial* Library, a real time reactive spatial library that consists of (1) a front-end, a programming interface for developers who are familiar with the Reactive framework and the Microsoft Spatial Library, and (2) a back-end for processing spatial operations in a streaming fashion. Then we provide demo scenarios that show how *RxSpatial* is employed in real-world applications. The demo scenarios include criminal activity tracking, collaborative vehicle system, performance analysis and visualization of the library's internal algorithms.

1. INTRODUCTION

In many time sensitive GIS applications, real-time location tracking enables immediate feedback when certain conditions are met, (e.g., entering a risky region [8]). For instance, in some child safety systems, children wearing smart devices are tracked. When they get near forbidden or dangerous areas, the application needs to send an alert to parents on time so that they can stop the children from getting hurt. In the scenario of autonomous vehicle clusters, where a cluster of driverless cars are on the way heading to the same direction, every car is a moving object to be tracked. If any car is too far away from the cluster or too close to its neighboring cars, actions should be taken to adjust its movement. In advertising applications, when a driver is close to a shopping mall, coupons, recommendations, and possibly a parking spot information needs to reach his smart device. In social networks, when a friend is nearby, it would be good to receive a push notification of the friend's location information. In safe routing engines [8], the commuters' location information and the disastrous weather zone require real-time processing to generate a reasonable life saving emergency route to evacuate. To fulfill the needs of these applications, this paper introduces the *Reactive eXtension Spatial library*, *RxSpatial* for short. *RxSpatial* offers a front-end programming interface for developers. This front-end provides new interface that is called *RxGeography* that is derived from the IObservable interface of the Microsoft Reactive Framework [11]. RxGeography implements modified versions of the methods found in the STGeography class in the SQL Spatial Library. For example, the STGeography class implements a method called STIntersection to detect if two 'stationary' geography objects intersect. The RxGeography implements 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 STGeography). It also implements a method called RxRxIntersection to continuously monitor and detect the intersection of two moving objects,(i.e., both of them are represented as RxGeography).

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. This is achieved by mapping each moving object to an Observable (i.e., IObservable interface) and each location update for a moving object triggers the geospatial computation module of the observers who have subscribed this object's movements. Then, the location, direction and speed of the moving object are used on top of multidimensional index structure to efficiently process various geospatial operations.

Fortunately, we release the *RxSpatial* library equipped with the RUM-Tree [12] as the core spatial index. RUM-Tree is chosen based on its ability to handle moving objects update in an efficient way compared to other conventional index structures, e.g., R-trees and quad trees. Furthermore, the geospatial computation in the *RxSpatial* library is powered by the spatial data types supported in the Microsoft SQL Server Spatial Library [2]. It provides optimized data types such as SqlGeometry and SqlGeography to store and query objects in flat and geodetic spaces, respectively. Various methods are provided to handle these spatial data types. The SQL Server Spatial Library adheres to the Open Geospatial Consortium "Simple Feature Access specification" [10] which is inherited by our proposed *RxSpatial* library. Moreover, the *RxSpatial* library provides strong support for asynchronous program devel-

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2899411

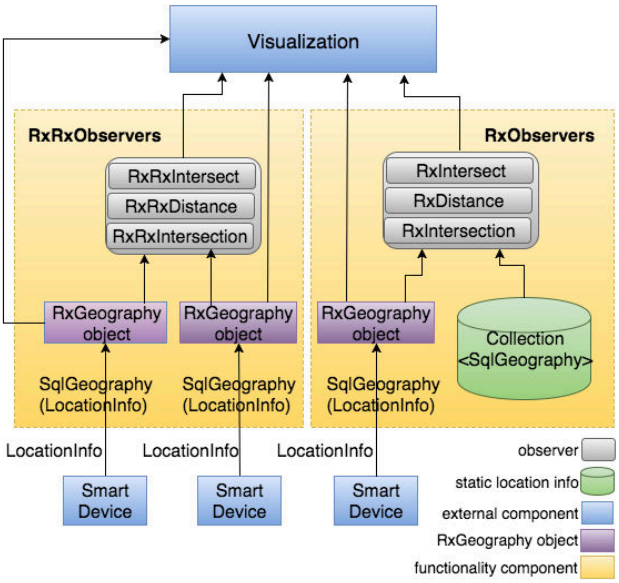


Figure 1: The Architecture of The *RxSpatial* Library.

opment and event-based programs, with a smooth learning curve. Developer is able to represent asynchronous data streams with Observable objects [3]. This is enabled through the use of SqlGeography object as input to the onNext API supported by Microsoft Reactive Extension(Rx) [11]. Thus, the *RxSpatial* library can query the asynchronous data streams using LINQ [1]. It is also able to parameterize the concurrency issues related to the asynchronous data streams using Schedulers. To illustrate the usability of the proposed *RxSpatial* library, we describe two real-world applications; monitoring criminal activity system, and collaborative vehicle system.

2. ARCHITECTURE

Figure 1 provides a high level description of the *RxSpatial* architecture. Location data streams are continuously fed to the server from registered smart devices. Standing spatial queries are also registered within the system. For the sake of this demo, the query result is sent to a visualization module. The *Reactive Spatial Library* mainly introduces two types of observers: *RxObservers* and *RxRxObservers*. *RxObservers* monitor the relationship (e.g., intersection or distance) between one static (i.e., non moving) object and a non-static (moving) object. *RxRxObservers* monitor the relationship between two moving objects.

An RxGeography object and a moving object have one-to-one correspondence. A moving object's location is represented in the library by an RxGeography object. RxGeography comes with an interface for observers (both RxObservers and RxRxObservers) to subscribe to the location updates of its moving object. When a location update of a smart device is received, the new location is stored inside the RxGeography object. Then, it is propagated to all subscribing observers.

2.1 RxObservers

Any observer that monitors the status between a moving object and a non-moving object is called *RxObservers*. It stores the location data of a pre-stored static location G locally. It is triggered by the event of a location update from the moving object that is being monitored. As a naming convention, there is just one 'Rx' in the name of any type of RxObservers because there is only one moving object involved. *RxObservers* comes with three different types; each of them utilizes different APIs provided by the SQL Server Spatial Library: (i) *RxIntersect* observer that outputs

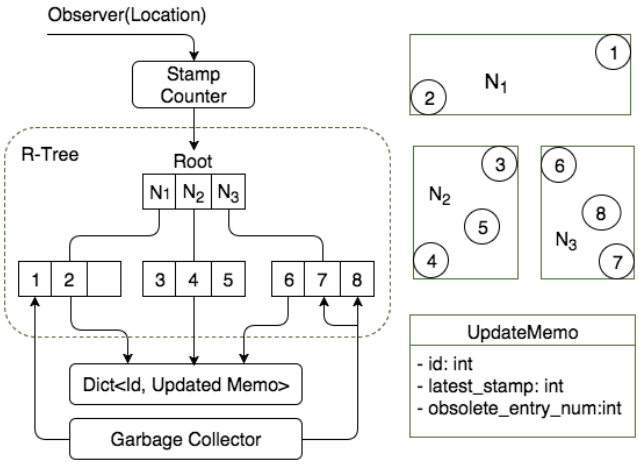


Figure 2: Integrating RUM-Tree (R-Tree With Updated Memo) Inside *RxSpatial*.

a boolean value. This internally calls the STIntersect API, e.g., G.STIntersect(g) indicates whether a static object G intersects a moving object g, (ii) *RxDistance* observer that outputs a double value, e.g., G.STDistance(g) computes the distance between G and g , and (iii) *RxIntersection* observer that outputs a SqlGeography object, e.g., G.STIntersection(g) computes the intersection region between G and g.

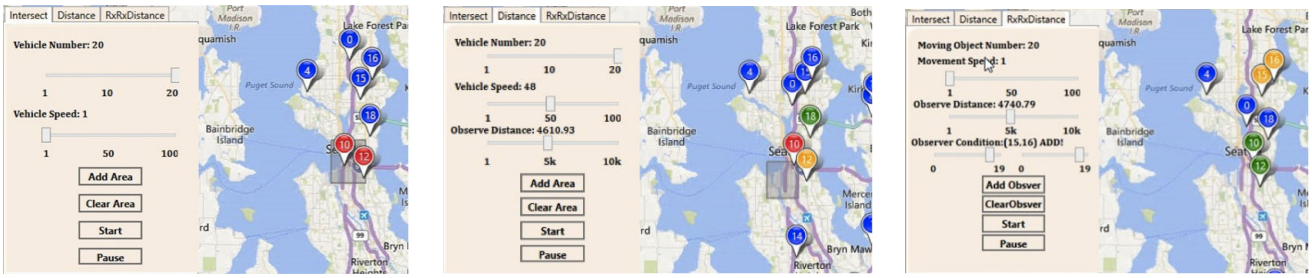
In the RxGeography class, calling RxIntersect, RxDistance, RxIntersection APIs subscribes the *RxObserver* of the corresponding operation type to the movement of this RxGeography object. Every time a new location is fed to the RxGeography object; each *RxObserver* will be notified by calling the OnNext API. This will recompute the corresponding computation, and then new update is streamed to the final output.

2.2 RxRxObservers

Any observer that monitors the relationship between two moving objects is called *RxRxObserver*. *RxRxObserver* is triggered by the event of a location update from anyone of the two observed moving objects. As a naming convention, there are two 'Rx's in the name of any type of RxRxObserver because there are two RxGeography objects involved in the operation. Three different variations of the *RxRxObservers* are currently implemented in the *RxSpatial* library. Each one of them internally dispatches a corresponding API provided by the SQL Server Spatial Library. The variations are: (i) *RxRxIntersect* observer that outputs a boolean, e.g., g_1.STIntersect(g_2) checks whether g_1 intersects g_2, (ii) *RxRxDistance* observer that outputs a double, e.g., g_1.STDistance(g_2) computes the distance between g_1 and g_2, and (iii) *RxRxIntersection* observer that outputs a SqlGeography object, e.g., g_1.STIntersection(g_2) gets intersection region between g_1 and g_2, where g_1 and g_2 are both moving objects.

In RxGeography class, by calling RxRxIntersect, RxRxDistance, and RxRxIntersection APIs, the *RxRxObserver* of the specified type can subscribe the movement of this RxGeography object. By OnNext API, new location update will be fed to the observed RxGeography object. Then, each *RxRxObserver* will be notified with this update to do the consequent computation.

As mentioned earlier, the main users of this library are developers. The advantage of designing the architecture in this way is that all the internal interfaces, observers and data structure do not need to be maintained by the application developer. This isolation allows developers to use the *RxSpatial* with little knowledge about the structure of the underlying SQL Spatial library.



(a) Restricted Region Enter Tracking. (b) Restricted Region Proximity Tracking. (c) Mutual Proximity Tracking.

Figure 3: Using the *RxSpatial* Library For Geofencing And Proximity Distance Operations In Criminal Activity Tracking System.

2.3 Data Structure

The data structure we chose is RUM-Tree, R-Tree with update memos. RUM-tree is used to store observer objects. It consists of three main components: a stamp counter, a R-Tree and update memos. We use stamp counter to add stamp to each insert/delete operation; the observer objects themselves are stored in an R-Tree; 'update memos' act as a memory-based auxiliary structure that help differentiate obsolete objects from the new ones. The 'update memos' are dictionaries whose keys are the unique object ids. Each update memo consists of three parts: a unique object id, the newest stamp of this object id, and the total number of obsolete objects with the same object id that required to be removed in garbage collection stage.

An insertion operation is executed when we store a new observer into the RUM-Tree. It is performed as follows: firstly the observer object is assigned an id; then we add 1 to the stamp counter and the observer is assigned with the current stamp. After that the observer would be inserted into R-tree and stored into a R-Tree node together with its stamp, indexed by its location; then a automatic check would be performed by the library to check whether this id is stored in the Update memos. If this id was found in the Update memos, the library would get the corresponding Update memos, modify the latest stamp with the current stamp and increment the obsolete entry number. On the other hand, if this id wasn't found in the Update memos, the id and Update memo pair would be inserted into the Update memo dictionary for reference by the library, the latest stamp in this Update memo entry would be set to the current timestamp, and the obsolete entry number would be set to 1, which means one Observer with this id would need to be removed.

An update operation is executed when the location of one observer is modified. It is performed with the following two steps: firstly the old entry of this observer is deleted from the RUM-Tree; then an insertion operation is performed. The deletion operation is as following steps: first the library would check whether the object id of this observer already exist in the Update memo dictionary. If it was found in the dictionary, the library would modify the corresponding Update memo with the latest stamp and add increment the obsolete entry number. Otherwise the id and Update memo pair would be inserted to Update memo dictionary by the library for reference; it would also set the latest stamp and set obsolete entry number to 1. We would be able to save the time needed to search and remove the old observer in the R-Tree, therefore speed up the deletion operation. After the deletion operation, the insertion operation is the same. Thus the overall update operation cost of RUM-Tree is less than R-Tree.

Figure 2 demonstrates how observers are organized in RUM-Tree. The area 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 this example, observer1 moved from location 1 in N_1 to 6 in N_3; observer2 moved from location 7 in N_3 to 2 in N_1; observer3 moved from location 8 in N_3 to location 3 in N_2. When a location update with new location 2 was sent from observer1, whose old location is 7, to RUM-Tree, the following steps are executed: firstly the stamp counter is increased and the new stamp is assigned to observer1; secondly; the library would check whether the update memo entry of observer1 exist: if the update memo entry exits, the library would update its stamp and increment the obsolete entry number by 1; Otherwise the library would create a new update memo entry for observer1, assign the newest stamp to the update memo and assign 1 to the obsolete entry, as only one node with exact same id needs to be deleted; then the library would put observer1 in an appropriate node. When a location update in n3 was sent to RxGeography object, the library would notify each observer except for the obsolete ones. In the described case, observer1 in location 6 would get the notification but observer2 and observer3 will not, as 7 and 8 are obsolete areas.

3. DEMO SCENARIOS

In this section, we demonstrate two application scenarios, i.e., *Monitoring Criminal Activity* and *Collaborative Vehicle System*, supported by our proposed Reactive Spatial library, (*RxSpatial*). In addition, we will depict an internal inspection of the system and give glances on the performance evaluation. The (*RxSpatial*) library and the application scenarios in this demo are implemented in C#, in Visual Studio 2013, and running on Windows 8.1. This demonstration is based on real GPS data streamed out of ankle bracelets [9] for the *Monitoring Criminal Activity* scenario. Two sets of moving objects on the Washington state USA [5, 4, 6, 7] are used for the *Collaborative Vehicle System* and the performance examination.

In the application snapshots, figures 3 and 4, we can notice that a numbered push pin represents a moving object. A blue pin means this moving object does not trigger anything or it is not even watched by anything. A pin with other colors means that moving object has some active condition. The control panel is on the left side of each figure. The function of buttons and scroll bars are as follows. *Start*: to start the objects movement, *Pause*: to pause the movement, *Vehicle speed*: to control the moving speed of pins, *Vehicle number*: to control the number of moving objects on the map, *Observer Distance*: to define the distance between observer and observed object which could trigger the watch notification, *Add Area*: to draw a rectangle in the map to add a stationary RxObserver, *Clear Area*: to remove all observers already added, and *Add Obsver*: to add a pair of mutually observing observers in RxRxObservers when the user selects the id of a pair of push pins and clicks the *Add Obsver* button. The color code of the push pin depends on the application scenario.

3.1 Monitoring Criminal Activity

The criminal justice system sometimes requires an offender on probation or on bail to have an ankle bracelet to track his location. Every criminal is designated a restriction area which he is supposed

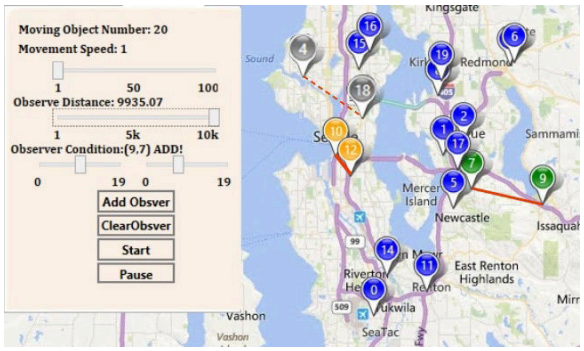


Figure 4: Distance Monitoring in Collaborative Vehicle Sys.

to stay away from. Sometimes, some offenders are not supposed to meet with each other. Figure 3(a) demonstrates restricted region tracking scenario: the gray rectangle indicates a restricted region where certain criminals, (i.e., push pins), are not supposed to enter. When the offenders 10 and 12 enter the restricted region, this act will trigger an alert.

Figure 3(b) demonstrates restricted region proximity tracking scenario where certain criminals are not supposed to get close to the restricted region. We can see that the offender 18 gets in the watching distance of the restricted region and thus triggers the watch notification. Offender 12 gets close enough to the restricted area, and then triggering security notification. When offender 10 gets in the restricted area, this will trigger enter notification.

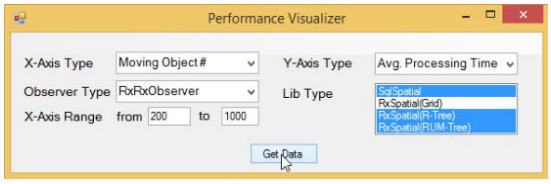
Figure 3(c) demonstrates mutual proximity tracking scenario: Criminal 10 and criminal 12 are not supposed to meet each other. Here in this figure, they are close enough and likely to meet, so the watch notification is triggered. Offender 15 and offender 16 are not supposed to meet either; in this figure they are so close that security notification is triggered.

3.2 Collaborative Vehicle System

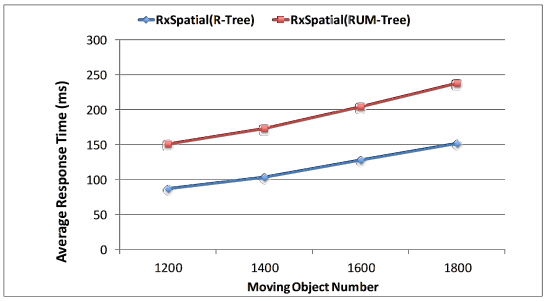
In this part, we are going to demonstrate the distance control in collaborative vehicle system. One scenario is data sharing which can only be guaranteed when the distance between cars is within a working distance. Examples include, the in-front road status and yielding for coming car message at the left turn can be shared from the leading car to following car. In Figure 4, each pin represents a car in the collaborative vehicle system. The collaboration status is tracked between 4 and 18, 10 and 12, and 7 and 9. The dotted line between car 4 and 18, and the gray color of pins, 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 10 and 12 and the yellow color of pins show that they are within a stable working distance. The thin line between 7 and 9, and the green color of pins 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 thus no need to audit the connections.

3.3 Performance Test Result Visualization Demo

In this part, we are going to have a control panel to visualize the performance of the system. Audience will have the ability to decide to examine the RxObserver or RxRxObserver. They can choose x-axis type and y-axis type and other parameters as well. The x-axis types are Update Interval, Observer Number per Object, or Moving Object Number, and the y-axis types could be Average processing time, or Max number of moving objects supported by the system. For example, the control panel configured like Figure 5(a) will generate the results charted in Figure 5(b).



(a) Performance Visualizer control.



(b) Performance Visualizer Chart.

Figure 5: Performance Visualizer.

4. 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] A. M. Hendawi, J. Bao, and M. F. Mokbel. iRoad: A Framework For Scalable Predictive Query Processing On Road Networks. In *VLDB*, Riva Del Garda, Italy, Aug. 2013.

[5] 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.

[6] A. M. Hendawi, A. Khot, A. Rustum, A. Basalamah, A. Teredesai, and M. Ali. COMA: Road Network Compression For Map-Matching. In *MDM*, Pennsylvania, USA, June 2015.

[7] A. M. Hendawi, E. Sturm, D. Oliver, , and S. Shekhar. CrowdPath: a framework for next generation routing services using volunteered geographic information. In *SSTD*, Munich, Germany, Aug. 2013.

[8] 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.

[9] 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*, pages 512–515, Florida, USA, Nov. 2013.

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

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

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