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

Youying Shi\textsuperscript{1} Abdeltawab M. Hendawi\textsuperscript{2} Hossam Fattah\textsuperscript{3} Mohamed Ali \textsuperscript{1}

\textsuperscript{1}Center for Data Science, Institute of Technology, University of Washington, WA, USA
\textsuperscript{2}Department of Computer Science, University of Virginia, VA, USA
\textsuperscript{3}Microsoft Corporation, WA, USA

\{youyings, nhdali\}@uw.edu
hendawi@virginia.edu
hofattah@microsoft.com

ABSTRACT

Current commercial spatial libraries implemented strong support on functionalities like intersection, distance, and area for various stationary geospatial objects. The missing point is the support for moving object. Performing moving object real-time location tracking and computation on server side of GIS application is challenging because of high user volume of moving object to track, time complexity of analysis and computation, and requirement of real-timing. In this Demo, we present the RxSpatial, 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 the demonstration scenarios that show how RxSpatial is employed in real-world applications. The demonstration scenarios include criminal activity tracking, collaborative vehicle system, performance analysis and an interactive internal inspection.

1. INTRODUCTION

In many timing-sensitive GIS applications, real-time location tracking enables immediate feedback when certain condition is met, (e.g., entering risky region \cite{7}. For instance, in some child safety systems, children wearing smart devices are tracked. When they get near forbidden or dangerous area, the application needs to send alert to parents on time so that they can stop the children from getting hurt. In the scenario of autonomous vehicle clusters, a cluster of driverless cars is 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 before he arrives. In social networks, when a friend is nearby, it would be great to get push notifications of location information of the friend. In safe routing engines, the commuters’ location information and the disastrous weather zone require real-time processing to generate a reasonable live saving emergency route to evacuate.

Similar libraries in the previous work \cite{2, 10} were originally designed to support operations on stationary objects with limited capabilities for moving objects.

This paper introduces the Reactive eXtension Spatial library, RxSpatial for short. The RxSpatial offers a front-end programming interface for developers. This front-end provides new interfaces that are called RxGeography such that each interface derives from the IOpenable interface of the Microsoft reactive framework. The RxGeography implements modified versions of the methods found in the STGeography 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 RXIntersect to continuously monitor and detect the intersection of two moving objects (i.e., 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 geometries, the intersection operation is not carried over from scratch between the moving object’s location and all geometries with 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., IOpenable interface), yet each location update for a moving object will trigger the geospatial computation with 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 the intersection as well as other geospatial operations.

Fortunately, we release the RxSpatial library equipped with the RUM-Tree \cite{11} as the core spatial index. RUM-Tree is chosen based on its ability to handle moving objects update in more efficient way compared to other conventional index structures, e.g., grid. Furthermore, the geospatial computation in the RxSpatial library is powered by Spatial data types supported by Microsoft SQL Server Spatial Library \cite{2}. It provides optimized data types such as SqlGeography to store and query objects in a geometric space. Various methods are provided to handle these spatial data types. This underline SQL Server Spatial Library follows the Open Geospatial Consortium “Simple Feature Access specification” \cite{9} which is in-
herited by our proposed RxSpatial library. Moreover, the RxSpatial library provides strong support for asynchronous program development 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) [10]. Thus, the RxSpatial library can query the asynchronous data streams using LINQ operator [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 are feed to the server continuously from a registered smart device. The computation result would be sent to the visualizer for display after the data being processed.

As shown in Figure 1, the Reactive Spatial Libray mainly consist of multiple RxGeography objects, a global set of SqlGeography objects in which each SqlGeography object represent the location information of a static polygon area on earth sqlServerSpatial, and two important type of observers. They are RxObservers, whose functionality is monitor status between on non-static object and a static polygon area, and RxRxObservers, whose functionality are monitor status between two non-static objects. RxGeography object and moving smart device have one-to-one correspondence. 'Rx' represents Reactive Extension that we chose for this library. RxGeography comes with interface for observers to subscribe the location change of this device. When a location change of a smart device happens, the location data is being stored in an SqlGeography object g. The RxGeography object collects g, and then send g to the observers subscribing this devices.

2.1 RxObservers

Any one of the observers that monitors the status between a non-moving area and a 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 location change from the moving objects it is watching. Because only one RxGeography object is involved, there is just one ’Rx’ in the name of any type of RxObserver. RxObservers comes with three different types; each of them utilizes different APIs provided by SQLServer Libraries: (i) RxIntersect observer outputs a SqlGeography object, G.STIntersect(g), which stands for whether G intersects g. (ii) RxDistance observer outputs a double, G.STDistance(g), which stands for the distance between G and g, and (iii) RxIntersection observer outputs a SqlGeography object, G.STIntersection(g), which stands for the intersection region between G and g.

In RxGeography class, by calling RxIntersect, RxDistance, RxIntersection APIs, the RxObserver of the specified type can subscribe the movement of this RxGeography object. By OnNext API, new location g is fed to the RxGeography object; each RxObserver will be notified with g, finishing the corresponding computation based on g, and then update the output.

2.2 RxRxObservers

Any one of the observers that monitors the status between two moving objects are called RxRxObserver. RxRxObserver is trigged by the event of location change from the observed moving object. Because there are two RxGeography objects involved, there are two ’Rxs’ in the name of any type of RxRxObserver. RxRxObservers comes with three different types; each of them utilizes different APIs provided by Sql Server Libraries: (i) RxRxIntersect observer outputs a SqlGeography object, g1.STIntersect(g2), which stands for whether g1 intersects g2. (ii) RxRxDistance observer outputs a double, g1.STDistance(g2), which stands for the distance between g1 and g2, and (iii) RxRxIntersection observer outputs a SqlGeography object, g1.STIntersection(g2), which stands for the intersection region between g1 and g2.

In RxGeography class, by calling RxRxIntersect, RxRxDistance, RxRxIntersection APIs, the RxRxObserver of the specified type can subscribe the movement of this RxGeography object. By OnNext API, new location g2 will be fed to the observed RxGeography object; each RxRxObserver will be notified with g2, finishing the corresponding computation based on g2 and the location of observing RxGeography object g1, and then update the output.

The main users of this library are developers. The advantage of designing the architecture like this is that all the internal interfaces, observers and data structure do not need to be maintained by developer. The exposed APIs are clear and easy to program. Thus, developers are able to program using this library with little knowledge about Sql Spatial library.
2.3 Data Structure

The data structure we chose is RUM-Tree, R-Tree with update memos. RUM-tree is leveraged 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 a R-Tree; ‘update memos’ acts as a memory-based auxiliary structure that would help differentiating obsolete objects from the newest objects. The ‘update memos’ is a dictionary whose key is unique object id and value is a single update memo. Each update memo consist 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 with 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 an 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 memo, 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 newest 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: N1, N2 and N3; location 1 and 2 are in N1, location 3, 4 and 5 are in N2. Location 6, 7 and 8 are in N3. In this example, observer1 moved from location 1 in N1 to 6 in N3; observer2 moved from location 7 in N3 to 2 in N1; observer3 moved from location 8 in N3 to location 3 in N2. 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 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 are going to 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 the performance evaluation glanced. 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 [8] for the Monitoring Criminal Activity scenario. Two sets of synthetic moving objects on the Washington state USA [5, 4] are used for the Collaborative Vehicle System and the performance examination. We also employ small set of real objects [6] for visual inspection of the results quality.

In the applications snapshot, 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 Observe: to add a pair of mutually observing observers in RxRxObserver when the user selects the id of a pair of push pins and clicks the Add Observe 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 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 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).

4. REFERENCES