Real-Time Spatio-Temporal Location Tracking and Processing of Moving Objects

Youying Shi[1] Abdeltawab M. Hendawi[2] Hossam Fattah[3] Mohamed Ali [1] Jumana Karwa [1]

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

ABSTRACT

There has been an increasing need for tracking moving objects efficiently in real time from telecom providers and the transportation industry in tracking hundreds of millions of moving objects in real time. Another common use case is real time tracking or monitoring of activities of certain people like family or friends when they check in to some place, or share interesting location pictures on a scenic drive, share an entire timeline of photos and videos of several road trips in real time. Existing commercial spatial libraries have good support for stationary geospatial objects. However, there has been a dearth of the same for moving objects. There are several challenges in handling complex computations and storage of this high-frequency, low-latency, real-time moving objects for GeoSpatial applications. This Demo intends to present a reactive GeoSpatial library that provides a streaming processing unit to efficiently handle spatio-temporal operations, and a set of application programming interfaces (APIs) for developers. The demonstration scenarios include SpatioTemporal Timelines, Car Pooling Tracking System, Family Locator, Safety Check and Targeted Ads.

1. INTRODUCTION

During the past decade a lot of research has been done on location-based services, moving objects, traffic jam preventions, whether prediction, etc. Real time tracking of moving objects too have become an important aspect of many of the real world applications and providing efficient and strong support for such applications is very much important. For example, in real time navigational systems, providing efficient and swift handling of the information is of prime importance. Also, positioning information can prevent workers from entering areas exposed to environmental risks like downfalling threats, landslides and so on [4]. Another common use case for realtime data is storing objects in motion and analyzing those data points as they move and generate streams of notifications. Hence, we need here, a stable and efficient publish-subscribe system that would be able to quickly notify interested subscribers efficiently in real time. Handling the volume of data that moving objects create, and slicing and dicing the data to find interesting analytical results is the kind of task for which this library is a great fit. A natural thing to do with moving objects is to plot them on a map and to analyze the data generated by the object as it makes its journey. For example, on a map of transit data for the car pooling system, we can ask how many cars pass by every hour on this route, what are the start and end points of these cars and and the average time these cars take to reach their destination. Unlike stationary geospatial objects, moving geospatial data in real time poses many challenges, in terms of complexity of data structures, their representation, and manipulation, specially in terms of updating the location information in real time. The main challenge here is that with millions of moving objects, it would be really inefficient to process and analyse the same thing from scratch each time. In order to address this primary need, this demo introduces a library [6], that extends the existing SQL Server Spatial Library for efficiently handling moving objects in real time. It provides an API for processing spatial operations in a streaming fashion for 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 incrementally evaluated to make use of as much computations as possible from the previous step. The following section provides the layout and briefly explains the primary components of this library.

2. LIBRARY LAYOUT

Location data are fed to the server continuously from a registered smart device. The computation result would be sent to the visualizer for display after the data is processed. Our library provides a quick and efficient way to track and process real time moving objects. It does this by mapping each moving object to an Observable (i.e., IOservable interface), yet each location update for a moving object will trigger the geospatial computation with the observers who have subscribed to 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 library equipped with the RUM-Tree [5] as the core spatial index. Figure 1 provides a high level description of the library. The primary elements of the layout are as follows:

RxObservers: These are observers that monitor the status between a nonmoving area and a moving object. It stores the location data of a pre-stored static location L locally. It is triggered by the event of location change from the moving objects it is watching. It comes with three different types; each of them utilise different APIs provided by SQLServer Libraries: (i) RxIntersect observer outputs

a boolean, L.STIntersect(g), which stands for whether L intersects g, (ii) RxDistance observer outputs a double, L.STDistance(g), which stands for the distance between L and g, and (iii) RxIntersection observer outputs a SqlGeography object,L.STIntersection(g), which stands for the intersection region between L and g. An observer 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.

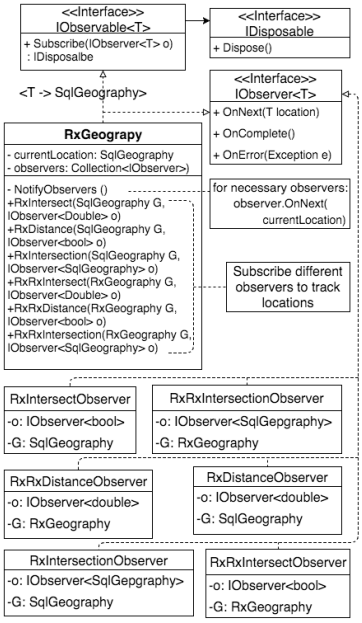

Figure 1: Library Layout

RxRxObservers: Any one of the observers that monitors the status between two moving objects are called Dynamic Observers. This observer is triggered by the event of location change from the observed moving object. Because there are two moving objects involved, these are dynamic observers (and two Rx in RxRxObservers). They come with three different types; each of them utilize the same API as static observers, however this time its between location updates between two moving objects, e.g., g1.STIntersection(g2), which stands for the intersection region between moving objects g1 and g2 where g1 and g2. Similar to the static observers, a call to the OnNext API would result in new location g2 to be fed to the observed RxGeography object.

RUM Tree: 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. 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 differencing 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.Furthermore, the geospatial computation in our library is powered by Spatial data types supported by Microsoft SQL Server Spatial Library [1]. It provides optimized data types such as SqlGeography to store and query objects in a geometric space. Fig 1 demonstrates the the layout of this library. 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. On similar lines, it would be really helpful for commuters to know if they are heading to a potentially unsafe area. Similar libraries in the previous work [2, 3] were originally designed to support operations on stationary objects with limited capabilities for moving objects. We demo one or more of the scenarios and show how is our library would benefit the developer in projecting the updates from moving objects efficiently. To illustrate the usability of the proposed RxSpatial library, we describe a few real-world applications; SpatioTemporal Timelines, Car Pooling Tracking System, Family Locator, Safety Check and Targeted Ads.

3. DEMO SCENARIOS

In this section, we are going to present a few demo scenarios that would greatly benefit from our library. The library and the application scenarios in this demo are implemented in C#, in Visual Studio 2013, and running on Windows 8.1. In the applications snapshot, figures 2 , 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 Spatiotemporal Timeline (iTrajectory)

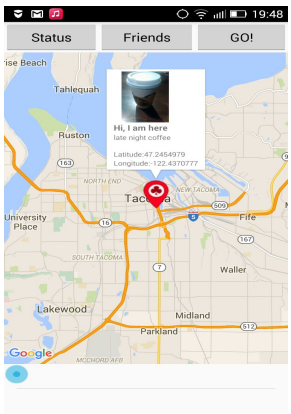

Figure 2: Sharing Location Pictures in Real Time on the Trajectory

This is an app developed for users to share information about their travel stories by sharing their trajectory along with hashtags, pictures and videos along the trajectory in real time. Fig 2 shows the user sharing a pic while picking up coffee from a drive in. More specifically, the social network phone app continuously collects the user's GPS locations along with various other user activities on the phone (e.g., pictures, videos and posts). Alternatively, the user gets the ability to specify a range over the map, and the system would retrieve all of the user's friends traveling within that region, given a date and time. As an example, the user may specify âĂIJfind all my friends in Seattle downtown on ChristmasâĂİ. Hence if a user and his friend are close enough to each other, the watch notification is triggered to both the users giving them the exact location of one another. In this case both users (say g1 and g2) would act as observers to other moving objects that would intersect their region of travel. Hence, the developer could easily use RxIntersection observer that outputs a SqlGeography object, g1.STIntersection(g2), which would easily give the intersection region between g1 and g2.

3.2 Targeted Ads

Spatiotemporal social networks leverage the business of behaviourally targeted ads to include the vicinity to the user being targeted (e.g., shopping mall coupons along the user's trajectory). This

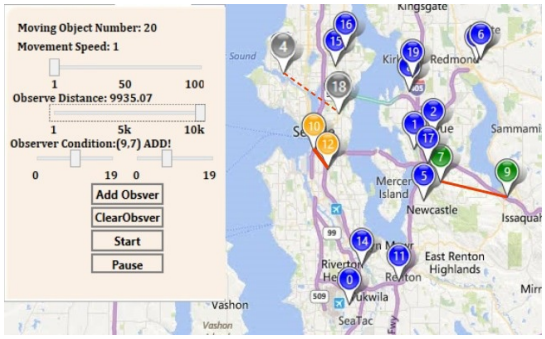

Figure 3: Car Pool Tracking

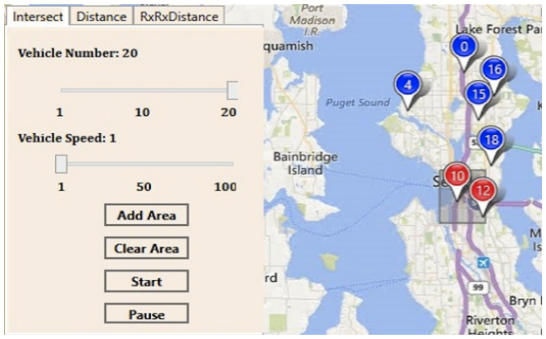

Figure 5: Unsafe Region Tracking

would require the businesses like shopping malls or food courts target customers moving closer to their vicinity with ads. This can again be easily achieved this time by using ESObservers - observers that monitor the status between a nonmoving area (shopping malls) and a moving object (cars) within a substantial range of their business locations to attract more customers.

3.3 Car Pooling Tracking System

In this part, we are going to demonstrate the distance control in a car pooling system. One scenario is knowing about cars that start and end at the same or nearby locations and most likely take the same route at similar timings to a common destination. Examples include, a car starting in Renton and taking the I-5 to reach Redmond between 9 a.m. and 9.30 a.m. In Figure 3, each pin represents a car in the car pooling vehicle system. The collaboration status is tracked between 7 and 9, 10 and 12, and 17 and 1. The red line between cars 7 and 9, show that the cars are watching the movement of each other since they started at nearby locations. 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 and would most likely happen to car pool. The dotted line between 4 and 18, and the grey 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. In this case too, the developer could easily use the RxRxIntersection observer that outputs a SqlGeography object, g1.STIntersection(g2), which would easily give the intersection region between g1 and other g2 moving cars. If the intersection is small till these cars reach their destination like 15 and 16, these cars could car pool.

3.4 Family Locator

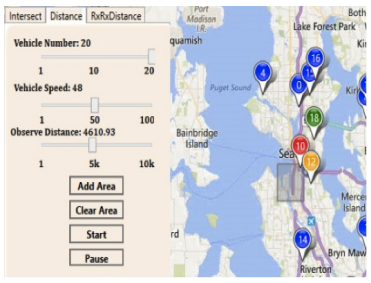

Figure 4: Family Region Tracking

Every child say in the age group of 8 - 16 of a family is designated an area i.e. home or school for example which is supposed to notify his family as soon as he checks in that location. Sometimes, some children are not supposed to move outside these vicinities.Figure 4 demonstrates restricted region tracking scenario: the gray rectangle indicates a restricted region out of which children, (i.e., push pins), are not supposed to exit without their parent. When the child 10 is nearing the exit region, this act will trigger an alert to the supervisor and the parent.

3.5 Safety Check

Another example is warning commuters of potentially hazardous areas when on the move. Figure 5 demonstrates unsafe region tracking scenario where certain drivers are not supposed to get close to restricted regions like accident prone or unsafe areas. When user 10 gets in the restricted area, this will trigger security notification to the subscribed user and also possibly notify the nearest police station.

4. REFERENCES

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

[2] 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.

[3] 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.

[4] 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.

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

[6] Y. Shi, A. M. Hendawi, H. Fattah, and M. Ali. RxSpatial: Reactive Spatial Library for Real-Time Location Tracking and Processing. In *MDM*, Pennsylvania, USA, June 2015.