Ad-hoc Ride Sharing Application using Continuous SPARQL Queries

Debnath Mukherjee
TCS Innovation Labs
Kolkata, India
debnath.mukherjee@tcs.com

Snehasis Banerjee
TCS Innovation Labs
Kolkata, India
snehasis.banerjee@tcs.com

Prateep Misra
TCS Innovation Labs
Kolkata, India
prateep.misra@tcs.com

ABSTRACT
In the existing ride sharing scenario, the ride taker has to cope with uncertainties since the ride giver may be delayed or may not show up due to some exigencies. A solution to this problem is discussed in this paper. The solution framework is based on gathering information from multiple streams such as traffic status on the ride giver's routes and the ride giver's GPS coordinates. Also, it maintains a list of alternative ride givers so as to almost guarantee a ride for the ride taker. This solution uses a SPARQL-based continuous query framework that is capable of sensing fast-changing real-time situation. It also has reasoning capabilities for handling ride taker's preferences. The paper introduces the concept of *user-managed windows* that is shown to be required for this solution. Finally we show that the performance of the application is enhanced by designing the application with short *incremental queries*.

Categories and Subject Descriptors
H.4 [**Information Systems Applications**]: Miscellaneous

General Terms
Performance, Design

Keywords
Ad-hoc ride sharing, stream reasoning, SPARQL

1. INTRODUCTION
In ad-hoc ride sharing, there are essentially two types of users: the ride taker who needs a ride, and the ride giver who offers a ride. So the essential requirement for a ride sharing application is matching the ride giver's route with the ride taker's start and destination locations. However, this problem is complicated by two facts: firstly, the ride giver may be delayed by traffic and may not meet his commitment, and, secondly, the ride giver may not show up at all due to some exigency. The solution described in this paper attempts to mitigate these issues by keeping track of the real time traffic situation and the ride giver's GPS position; and by maintaining a list of alternative ride givers who will be able to offer a ride. This dynamic ad-hoc ride sharing is a non-trivial problem: real time streams of traffic status and ride giver's positions have to be combined and alternative ride givers have to be maintained for each ride taker to ensure the best possible experience for the ride taker.

Copyright is held by the author/owner(s).

WWW 2012 Companion, April 16–20, 2012, Lyon, France.

ACM 978-1-4503-1230-1/12/04.

Some existing ride sharing solutions are: PickupPal[1] and Zim-ride[2]. A full-fledged ride sharing system has been designed in [1]. [2] emphasizes on delivering optimal rides with incomplete spatial and temporal knowledge. [3] views the problem as an optimization problem: minimizing total miles travelled, minimizing total time spent, and maximizing the number of rideshare participants, for all the rides. The current paper mainly focuses on improving user experience through reducing uncertainties and keeping track of alternative ride givers. This is not fully addressed in the cited works.

2. APPROACH TO THE SOLUTION
The solution clearly requires continuous monitoring of the real time situation including incoming streams of traffic status, ride taker and ride giver information. Also, these streams need to be combined with spatial knowledge about routes and places. Also, the user's preferences have to be matched with the available options. If a ride taker wants to ride a SUV, and a ride giver has a Ford Explorer, then the system should be able to reason that since Ford Explorer is an SUV, the ride giver qualifies to give the ride. Based on the above observations, stream reasoning seemed to be a good fit for the problem since there is a combination of streaming information with background knowledge and reasoning is needed.

Background knowledge about the spatial domain is maintained in an ontology, where the following are maintained: “route segments" and “places" as entities, the relation “near" as a relation between place entities, and the relation “on" as a relation between a place and a route segment. It is assumed that there is a traffic reporting application which reports traffic events as the triple < r, s, t > where r is a route segment, s is a status such as “isCongestedAt" and t is a timestamp. The solution architecture is depicted in Figure 1.

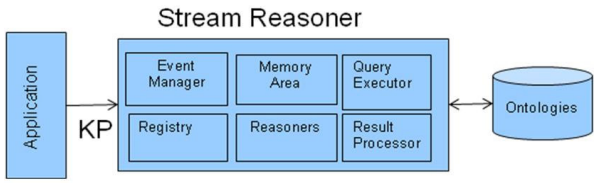

Figure 1. Solution Architecture

The ride giver and ride taker submit their requests to the application which forwards them to the stream reasoner (SR) as a set of RDF triples called knowledge packets (KP). All events sent to the SR have an associated KP. A KP represents a request to the

[1] http://www.pickuppal.com

[2] http://public.zimride.com

SR. An example of a KP is the ride giver KP which contains the route segments the ride giver will traverse, the order in which these segments will be traversed and the car type. On receipt of a KP, the Event Manager of the SR adds the triples to the working memory (called Memory Area) and invokes the SPARQL queries (via the Query Executor) that are registered (in the Registry) for the particular KP (in this case the "ride giver KP"). When the triples are added, reasoners (such as rule-based reasoners) act upon them to produce additional facts. The results produced by running the queries are sent to the Result Processor which invokes listeners registered by the client applications. Note that the working memory also contains the facts from the spatial domain. Also, the KP concept simplifies application programming by disallowing partial requests to be sent to the SR, as requests in this application need to be completely specified. A KP can be of two types: "insert" and "delete" where the former is used in requests to the SR and the latter to remove a request (to implement user managed windows discussed later). Continuous SPARQL queries are triggered by the arrival of a KP. Each query can be triggered by multiple KPs. One way to design the application would be to run the All Rides Query (ARQ) which considers all possible events (ride giver, ride taker and traffic) and finds all pairs of matching ride givers and ride takers. It can be seen that this is a computationally intensive operation. Instead our approach is to build the state of the application incrementally using incremental queries by considering the impact of each event separately. The state of the application consists of a list of ride takers and their corresponding feasible ride givers. The impact of a ride taker who has arrived in the system is calculated by the incremental ride taker query (IncRTQ), which has the following code fragment:

…

```
?rideGiver rg:hasRoute ?route.
?route rg:hasSegment ?routeSeg1.
?route rg:hasSegment ?routeSeg2.
(?1) s:on ?routeSeg1. (?2) s:on ?routeSeg2.
…
```

where rg is the IRI prefix of ride giver domain, s is the IRI prefix of spatial domain. Our implementation of SPARQL allows parameters similar to parameters in SQL statements. (?1) is a parameter representing the ride taker's start point and (?2) represents the ride taker's end point. This query outputs the ride givers for this newly arriving ride taker. These ride givers are updated in the application state. Similarly, the incremental ride giver query (IncRGQ) outputs the feasible ride takers for a newly arriving ride giver, the incremental traffic query (IncTQ) outputs the ride givers who are stuck in traffic and the incremental traffic clear query outputs the ride givers which have come out of a traffic situation and are ready to offer rides again. We assume that the ride giver's mobile device would be able to send an event to the application when the ride giver changes route segments, based on GPS. A continuous query, triggered by a route segment change by the ride giver, checks whether the current route segment is one of the route segments declared by the ride giver at the start of the ride and if not, flags the ride giver as having changed routes.

Another point to note is the need for user managed windows. In this application it can be seen that ride takers and ride givers leave the system at will and there is no fixed lifetime for a ride giver or ride taker request. While previous stream reasoners [4,5] have used time-based or count-based windows, these are not optimal for this application. There needs to be a mechanism to delete a ride giver request when the ride giver finishes the ride (similarly for ride takers). In our solution, when the user finishes the ride, the ride giver sends the request KP back as a "delete request", and the triples corresponding to the request KP are deleted. Request KP may also be deleted on expiry of a specified time period.

3. RESULTS AND CONCLUSION

To simulate a real life ride sharing scenario, we used the Google Maps Places API[3] to fetch data about locations and its nearby establishments. The locations on the route from Washington to New York were used to generate the data. In the simulation, a ride taker randomly provided start and end points, and it was matched with randomly generated routes of the ride givers.

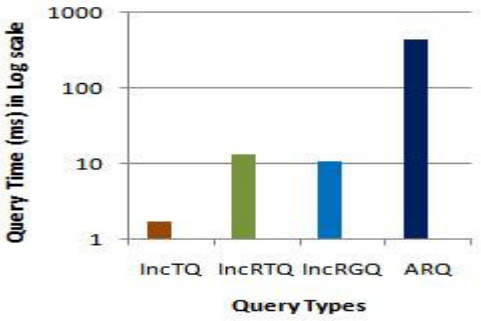

Figure 2: Comparison of incremental queries with ARQ

The comparison of the average times taken by each type of query is shown in Figure 2. It is seen that the All Rides Query is more than an order of magnitude slower than the incremental ones.

In this paper, we have shown the design of an ad-hoc ride sharing application that reduces the uncertainties of the ride. Instead of considering all event types in the continuous query, we show that building the application state incrementally using a single event type boosts performance significantly. We have also introduced the important concepts of knowledge packets and user managed windows. We have shown the architecture of a stream reasoner framework that we have designed. Further work on improving performance of the stream reasoner is planned.

4. REFERENCES

[1] Shao, J. and Greenhalgh, C. 2010. DC2S: a dynamic car sharing system. In *Proceedings of LBSN-2010*, 51-59.

[2] Winter, S. and Nittel, S. Ad-hoc Shared-ride Trip Planning by Mobile Geosensor Networks. 2006. *International Journal Of Gegraphical Information Science*, volume 20, 899-916.

[3] Agatz, N., Erera, A., Savelsbergh, M., Wang, X. 2010. *Sustainable Passenger Transportation: Dynamic Ride-Sharing*. Technical Report. ERIM Report Series, Research in Management.

[4] Barbieri D, Braga D., Ceri S., Valle E. and Grossniklaus M. 2009. C-SPARQL: SPARQL for continuous querying. In *Proceedings of WWW-2009*, 1061-1062.

[5] Anicic D., Fodor P., Rudolph S. and Stojanovic N. 2011. EP-SPARQL: A unified language for event processing and stream reasoning. In *Proceedings of the WWW-2011*, 635-644.

[3] http://code.google.com/apis/maps/