Towards Efficient Stream Reasoning

Debnath Mukherjee, Snehasis Banerjee, and Prateep Misra

TCS Innovation Labs, Tata Consultancy Services
Kolkata, India
{debnath.mukherjee,snehasis.banerjee,prateep.misra}@tcs.com

Abstract. We present a stream reasoning system, QUARKS, which has features like knowledge packets, application managed window and incremental query. Combination of rules and continuous queries along with application optimization has been used to address high performance requirements. Experimental results show that our proposed methodology is effective.

Keywords: stream reasoning, continuous query, rule, SPARQL, RDF.

1 Introduction

With the high volume of data being generated in real time from many sensors, there arises a need for reasoning on the data in real time [6]. Stream reasoning performs reasoning on a combination of real time streams of facts (usually represented as RDF triples) and static or slowly changing facts (known as background knowledge). A triple is of the form <subject, predicate, object>. For example, a fact could be <John, locatedAt, North Street>. Here "John" is the subject, "locatedAt" is the predicate and "North Street" is the object. In the context of stream reasoning, the real time streams of facts are synonymous with event streams.

In this paper, we describe a stream reasoner QUARKS (QUerying And Reasoning over Knowledge Streams) which uses registered continuous SPARQL queries and rule based reasoners. The contribution of this work is a stream processing layer on top of an existing Rete reasoner [4] and SPARQL processor to achieve good performance. QUARKS has three novel features: application managed windows, knowledge packets and incremental queries described in Section 2.

An early work in stream reasoning is C-SPARQL [1], a language for continuous queries over streams of facts combined with background knowledge. It extends the SPARQL language, adding features for RDF streams, windows etc. QUARKS uses continuous queries augmented by rules. CQELS [3] is another prototype which takes a native approach (without using existing data stream management systems or existing SPARQL processors). [2] has presented extensive experimental evaluation of the engine presented in [3]. QUARKS uses existing reasoners and SPARQL processors. [5] had briefly introduced the features of the stream reasoner QUARKS and had presented the results of using incremental queries for ad-hoc ride sharing scenario. In this paper, we show how a combination of rules and queries along with some application optimization can give good performance.

Y.T. Demey and H. Panetto (Eds.): OTM 2013 Workshops, LNCS 8186, pp. 735–738, 2013.

2 Features of QUARKS

In this section, we present some of the salient features of QUARKS.

Knowledge Packets: QUARKS supports processing of multiple triples at a time. A set of triples describing an event is called a knowledge packet (KP). The KP's triples are processed all at once to avoid processing partial knowledge. A KP has a "type" which could be "add" or "delete" depending on whether the KP is being added or removed. A KP has a timestamp, and a name referred to as the "class" of the KP.

Application Managed Window: Consider that a fire has broken out in a building – this leads to a "fire event". Here only the event sender knows when to delete the fire event. Thus a new type of window, called Application Managed Window (AMW) is required where the client application is allowed to control the deletion of events. The time to delete fire event is not determined by some fixed number of events or a fixed temporal duration of events (as in the traditional count and time based windows) – rather it is ad-hoc. In QUARKS, we allow the application to control not only the insertion of KPs but also their deletion. The AMW is a set of triples derived from the event's KPs. Additionally, there are two functions, "ifunc" and "dfunc", which add and remove KPs from the AMW.

Incremental Query: Incremental queries (IQ) compute the incremental matches for a single incoming event in contrast to re-computing the entire match. IQ are implemented using parameterized SPARQL queries. See [5] for more on IQ.

Other windowing mechanisms: In addition to Application Managed Windows, count-based windows and time-based windows are supported. A *count-based window* specified as "COUNT N" maintains the last N KPs of a particular class in the working memory. A *time-based window* specified as "RANGE N [Time Unit]" maintains the KPs of a particular class that were added in the last N time units.

3 System Architecture and Event Processing

The system architecture is depicted in Figure 1. The client application makes calls to the Stream Reasoner API for adding or deleting a KP, and to register listeners for the queries. When a KP is added, it is sent to the Event Manager which consists of queues for KPs that are waiting to be scheduled for processing. When a KP is processed, its triples are added to the working memory (called a Memory Area). The working memory is an in-memory data structure containing known facts including the inferred facts. There are multiple Memory Areas – often one per application. After a KP is added to the working memory, multiple reasoners (such as rule-based reasoners, OWL reasoner, RDFS reasoner etc) act upon the facts to produce entailments.

The reasoners use both the facts in the dynamic knowledge (the KPs) and background knowledge. Our current assumption is that both the relevant parts of the background knowledge and the dynamic knowledge from the KPs fit into main memory. After the processing by the reasoners, the Event Manager calls the Query Runner module to run registered continuous queries for the KP being processed. Note that QUARKS processes the KPs as a single unit: inserting the KP into working memory and then running the continuous queries that are registered for the KP.

The results of running the continuous query are sent to "listeners" which are

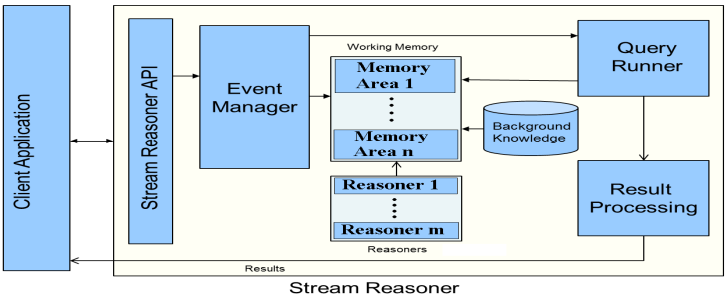

Fig. 1. System Architecture of QUARKS

Fig. 2. Event Processing in QUARKS

registered for the continuous query. The listeners are registered by the client application and return the results to the client application (formatting them if needed).

The event processing flow is depicted in Figure 2. First KP are queued for insertion into working memory (WM), then they are inserted into WM that triggers reasoners including rule based reasoners; finally registered continuous queries are run on the combined background knowledge and dynamic knowledge.

By default, a Rete-based rule reasoner [4] is used to process events – this is because Rete-based rule reasoner is a proven fast pattern matcher, and is one of the main reasons for good performance, but processing using only queries is allowed. Bulk of the logic should be written using rules to ensure good performance. We used Apache Jena framework for both Rete-based reasoning and queries. Note that our methodology of using a combination of rules and queries is not limited to a particular choice of rule engine or framework.

4 Experimental Evaluation and Conclusion

An experimental study was conducted to evaluate the stream reasoner. Four scenarios (Query 1, Query 2, Query 3, Query 4) from the experimental evaluation of CQELS [2] were evaluated using QUARKS. The results are presented in Table 1. Note that scenario Query 3 and Query 4 required two cascaded queries, but maintaining some logic in the client and using a single query instead of using two queries chained together performs better. The metric used was average processing time P defined as:
P = (Elapsed time to process N events) / N
(The elapsed time is adjusted for the first event insert time which is a one-time overhead, since the reasoner does some initializations).

From the result, it is clear that maintaining application logic and a single query instead of using two queries chained together performs better. The reason is that when

Table 1. Timing (in ms) for the scenarios using different methodologies

	Query 1	Query 2	Query 3	Query 4
Rule+Query	0.88	3.62	2.69	2.75
App Logic	-	-	0.98	0.98

the logic is simple, the overheads of using a query is more, and sometimes significant (as illustrated in the results). The above results are of the same order of magnitude as the state-of-the-art, as can be seen in [2]. The experiments were run on a system having Quad Core Intel Core i5 2.67 GHz CPU and 4GB memory.

Application optimizations: The ordering of the patterns in the rules had a significant effect on the performance. The rule pattern ordering guidelines [7] for the Rete engine Jess were found to be quite useful. Also, some of the rules had a subset of patterns that could be resolved from the background knowledge alone. So here an application optimization was to pre-compute these subsets of patterns with a separate rule which generates an entailment E, and replacing the subset of patterns in the original rule by the entailment pattern E. This gave significant performance benefits.

To conclude, in this paper we adopted a combination of Rete-based rule engine and a SPARQL query processor to implement a stream reasoner, QUARKS. Evaluation was done using the stream reasoning queries mentioned in [2]. The results produced are encouraging. The stream reasoner described also has some practical features such as Application Managed Windows, Knowledge Packets and Incremental Queries. In future work, we wish to explore automatically optimizing the ordering of patterns in the rule, automatically generating pre-computation rules, support for temporal operators and parallelization of processing (wherever possible).

Acknowledgments. We thank Dr. Dilys Thomas of TRDDC, Pune and Sounak Dey of TCS Innovation Labs, Kolkata for their comments on this work.

References

1. Barbieri, D., Braga, D., Ceri, S., Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. In: 18th World Wide Web Conference, pp. 1061–1062. ACM (2009)
2. Experimental evaluation of CQELS stream reasoner, `http://code.google.com/p/cqels/wiki/Experiments`
3. Le-Phuoc, D., Dao-Tran, M., Xavier Parreira, J., Hauswirth, M.: A native and adaptive approach for unified processing of linked streams and linked data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 370–388. Springer, Heidelberg (2011)
4. Forgy, C.L.: Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. Artificial Intelligence 19, 17–37 (1982)
5. Mukherjee, D., Banerjee, S., Misra, P.: Ad-hoc ride sharing application using continuous SPARQL queries. In: 21st International Conference Companion on World Wide Web (WWW 2012), pp. 579–580. ACM (2012)
6. Banerjee, S., Mukherjee, D., Misra, P.: 'What Affects Me?' A Smart Public Alert System based on Stream Reasoning. In: 7th International Conference on Ubiquitous Information Management and Communication (ICUIMC). ACM (2013)
7. Efficiency of rule-based systems, `http://herzberg.ca.sandia.gov/docs/70/rete.html`