Streaming Knowledge Bases

Onkar Walavalkar, Anupam Joshi, Tim Finin, and Yelena Yesha

Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County
Baltimore MD 21250, USA
onkar1|joshi|finin|yeyesha@cs.umbc.edu

Abstract. With the advent of pervasive computing, we encounter many scenarios where data is constantly flowing between sensors and applications. The volume of data produced is large, so is the rate of the dataflow. In such scenarios, knowledge extraction boils down to finding useful information i.e. detecting events of interest. Typical use cases where event detection is of paramount importance are surveillance, tracking, telecommunications data management, disease outburst detection and environmental monitoring. There are many streaming database applications built to deal with these dynamic environments. However, they can only deal with raw data – not with streaming facts. We argue that much like a new database approach had to be developed to deal with streaming data, a new approach will be required to deal with streaming facts expressed in the languages of the Semantic Web. Existing reasoners use techniques that load the whole RDF graph in main memory and carry out queries on it. This approach is of little use in real-time reasoning for streaming scenarios and takes considerable amount of time. In this paper, we combine a continuous query processors with Semantic Web techniques to build a reasoner that can deal with streaming facts. We describe our technique, and present empirical validation of its efficacy.

1 Introduction

The idea of creating context aware environments has interested the research community, especially over the last half a decade. This has been facilitated by the spread of pervasive computing – a growth in small sensors, wearable or handheld devices, and wireless networking technologies. However, the focus has been twofold. At the lower layers, the attention has been on getting data from the sensors and on creating architectures and frameworks that will support the integration of their data (e.g., [1–3]). At the upper layers, efforts have typically focused on using the sensed information to infer a context model. This has taken several forms, ranging in complexity to simple numerical threshold type systems (if $temperature > X$ then start air conditioner) to more complex systems that seek to build rich models of the domain and then reason over them.

Much of the existing literature describes reasoning using relatively simple rules or constraints directly defined over sensed variables. Our work in building the Context Broker [4] system has shown that domain models described in

languages such as RDF and OWL can augment the ability to sense context. A recent example of this approach is the KitchenSense project [5] at the MIT Media Lab. It uses embedded sensors and networked kitchen appliances to augment the human-appliance interaction. It uses the ConceptNet [6] ontology and directly associates facts in the OpenMind [7] knowledge base with sensor states.

Reasoning over domain knowledge bases has generally been on sensed data that update at a relatively low frequency – a person entering a room, a device being turned on, a particular chemical being detected in the air etc. Other systems share this feature, including our own prior work such as COBRA [4], and more recently, Trauma Pod [8]. Unfortunately, such approaches do not scale to a large class of problems where context awareness is needed or can have high utility. Consider, for instance, a context aware system to help first responders to an emergency, or surgeons in a trauma related operation during the golden hour, where there is a lot of dynamically changing facts and information that humans needs to pay attention to and potentially respond. During surgery for instance, vital signs data is constantly generated by sensors placed on the body. Similarly, in the operating room of the future[1], it is expected that various tracking technologies would generate streams of data about the locations and movements of personnel, equipment, medicine and patients.

Recent advances in database systems, in particular the technologies related to streaming databases, let us create standing queries, such as monitoring for a conditions like $SystolicBP$¿170 or $presentSystolicBP - pastSystolicBP > 0.1xPastSystolicBP$, that can be efficiently evaluated on the streaming data. However, the output from these is streaming knowledge (Systolic BP is High and Increasing), and the medically significant facts need to be inferred from queries over such knowledge streams. The knowledge in these streams is often in Semantic Web languages, which have been used to create ontologies in a variety of domains, including medicine. The creation of technologies that allow efficient querying over streaming knowledge is a prerequisite to creating complex context aware systems. Given the increasing use of Semantic Web languages to encode knowledge and facts, what is needed are knowledge bases that can evaluate standing queries over streams of RDF and OWL data. We point out that this is different from streaming data, because inference can allow streaming facts to generate other facts which might be in the query terms.

There are several reasoners built to handle Semantic Web data, for instance [9]. However, they perform poorly in case of streaming facts. This is because they try to carry out the reasoning process, which involves computationally heavy inferencing methods like traversal of RDF graphs, at runtime. The incoming data rate is much higher than the time taken by the reasoners for inferencing. The key insight that underlies this work is that if we could split the process and try to pre-compute some facts beforehand, it would better handle streaming knowledge. The other insight is that the work done by the database community in creating streaming database engine can then be leveraged by storing the pre-computed rules in database tables, using the engine to deal with streaming facts

[1] http://www.cimit.org/orfuture.html

and using simple database queries joining the incoming stream with the rule tables to perform the inferencing.

2 Related Work

2.1 Streaming Databases

The problem of mining fast data streams has been studied in various ways over the years. Babcock et al. [10] outline the research carried out and the challenges in the streaming databases. The main problem is that data is not present in the form of persistent relations, but rather arrives in rapid, time-varying continuous streams. The actual data tuples may have some relationship between their attributes i.e. they might be relational tuples. However, since they arrive at runtime with some unpredictability, they need to be handled differently.

One of the early works in this field was done in the Tapestry project [11] which supported continuous queries by modeling data streams as append-only databases. It implemented join operations by changing the continuous query into an incremental query as new records arrived. Aurora [12] is an experimental streaming database management system. It solved the problem of producing a timely output by implementing a scheduler for load-shedding based on the QoS specified by the application. It also focused on implementing scalable triggers as required in streaming scenarios.

TelegraphCQ [13] is another streaming database engine built at University of California at Berkeley. It handles continuous queries by routing tuples through query modules which are non-blocking versions of standard relational operators like joins, aggregations, selection and so on. The query gets executed step by step as the modules are pipelined. It uses the open source code base of UC Berkeley's PostgreSQL database system. We use this as the base of our system.

2.2 Semantic Web Reasoners

Many Knowledge Base systems have been developed for processing Semantic Web data. They differ in the approach they take towards reasoning. Some systems store the resource graph in main memory, while others use secondary memory for persistent storage and provide better scaling. Not all knowledge bases provide complete reasoning. There are some differences in their degree of reasoning. Key differences in knowledge base systems for Semantic Web data - especially OWL data - are examined by Guo et al. in [9].

Sesame [14] is a system built for efficient persistent storage and querying of metadata in RDF and RDF Schema. It uses a database repository for storage purposes. Bossam [15] is a RETE algorithm based forward chaining reasoner engine. It supports reasoning using negation-as-failure and classical negation. Also it supports remote binding for co-operative inferencing between multiple rule engines. FaCT++ [16] is a dedicated description logic reasoner and extends a lot of the features and optimization techniques of the FaCT System [17]. It's mainly

designed to experiment with new tableaux algorithms and optimization techniques. The Racer system [18] uses the Tableaux algorithm and has a complete reasoner for OWL-DL supporting both Tbox and Abox reasoning.

Pellet [19] is another description logic reasoner that supports OWL-DL. It's implemented in Java and supports conjunctive query answering, rule support and axiom pinpointing. It provides a command-line interface and an interactive Web interface that does not require any installation. Besides, it has a DIG server implementation and provides APIs to talk to RDF and OWL toolkits like Jena [20] and the Manchester OWL-API [21]. The OWL inference engine of F-OWL [22] is based on F-logic, an approach for defining frame-based systems in logic. This system is implemented using XSB and Flora-2. It uses a reasoner designed for FOL subset and in that, is similar to Jena [20] and Jess.

There are quite a few Semantic Web toolkits which provide other abilities apart from reasoning. The Wilbur [23] toolkit facilitates RDF processing using Common Lisp. The BRAHMS [24] system implements efficient storage for RDF data and is better suited for fast semantic association discovery. It is implemented in C++, achieves good memory management and scales well. Redland [25] provides another toolkit for RDF data and interfaces to store and modify instances of RDF graphs in C, Perl, Python, Tcl and other languages. Jena [20] is a java-based toolkit that provides a host of capabilities for building Semantic Web applications. It provides APIs for processing of RDF, RDFS, OWL and SPARQL. It has a rule-based inference engine.

None of the systems described above handle streaming semantic data. As mentioned, streaming nature of data poses different types of problems and to our knowledge none of the reasoners make any attempts at special handling for it. We utilize a continuous query processor to build a subsumption reasoner that can deal with streaming knowledge. Given an RDFS or OWL ontology, we precompute the transitive closure of all classes on the rdfs:subClassOf relationship and store the class-subclass pairs in a database table. At run-time we just need to query the database to identify subclass events of the event of concern. We evaluate the performance of our reasoner against that of Jena on streaming facts.

3 System Design

The operation of the streaming knowledge base is divided into two major functional components - Ontology Pre-processor and Stream Processor. We will look at each of them separately.

3.1 Ontology Pre-processor

The input to this stage is one or more OWL or RDFS ontology files. To process the ontology file we use the InfModel class provided by the Jena toolkit [20] to compute the following five relationships. which are stored in tables in the TelegraphCQ database.

rdfs:subClassOf. We compute the transitive closure of each class on the rdfs:subClassOf relationship, i.e. calculate every class that the concerned class is a subclass of and store this.

rdfs:subPropertyOf. We compute the transitive closure of each property on the rdfs:subPropertyOf relationship, i.e. calculate every property that the concerned property is a subproperty of and store this.

rdfs:range. We compute all the classes that form the range of every property in the given ontology and store these.

rdfs:domain. We compute all the classes that form the domain of every property in the given ontology and store these.

owl:inverseOf. We compute the inverse property of every property and store these.

3.2 Stream Processing Engine

TelegraphCQ [13] is a data stream processing engine developed at University of California at Berkeley. It accepts a stream of data tuples as input. The queries run continuously over this stream. Queries are processed by routing tuples through query modules, which are pipelined non-blocking versions of standard relationship operators like joins, selections, projections, groupings, aggregations and duplicate elimination. It has adaptive routing modules which are able to re-optimize the query plan continuously while the query is running. It has special modules called Eddies which decide how to route data to other modules. In Continuous Query mode it allows two ways to specify a time window over which the query is to be evaluated. It has a WINDOW clause as well as a GROUP BY SLIDE BY clause. It has a landmark window mode, where the start point of the window is fixed and end point moves continuously hence increasing the window size continuously. It also has a sliding window mode where both start point and end point move continuously, thereby window size remains constant but window keeps on sliding ahead. It also has a static data store like a traditional database management system, which leverages the code base of PostgreSQL. This is where we store the relationships precomputed from the ontology during the preprocessing stage.

We need to define a *schema* and create *streams* that should get the incoming data. We define the following stream for our reasoner application.

```
create schema reasoner;

create stream reasoner.instances
    (instancesSub varchar(150), instancesPred varchar(150),
    instancesOb varchar(150),tcqtime TIMESTAMP TIMESTAMPCOLUMN)
    TYPE UNARCHIVED;
```

```
alter stream reasoner.instances add wrapper csvwrapper;
```

The stream attributes correspond to subject, predicate and object respectively of an RDF triple [26]. The special column tcqtime is required for recording the time of arrival of the tuple. The stream is altered to be associated with a csvwrapper so that the data source can send the RDF triple in a comma separated format.

3.3 Standing Query Evaluation

In the most basic query, the stream processing engine is given a class of interest I. Any incoming triple which has a subject that is an instance of this class directly or of any of its subclasses is to be detected and flagged. When an incoming triple of the format *S, rdf:type, C* arrives, we want to flag S as an event of concern if C is I or any of its subclasses. We first find out the concerned class' id from the resources table in database and store it in a variable $lClassinstance$. The following static query is issued to achieve this.

```
select id from resources where class = I
```

A key feature of TelegraphCQ that we leverage allows joining a static database table with a windowed stream. We can now join the class of the incoming triple with the statically computed transitive closure of classes which is stored in the database, without having to do the inferencing at runtime. We use the following query to carry out our instance detection.

```
select reasoner.instances.instancesSub
  from resources, classtree, reasoner.instances
  [RANGE BY _interval_ seconds
   SLIDE BY _slide_ seconds]
  where reasoner.instances.instancesOb = resources.class and
        reasoner.instances.instancesPred =
          'http://www.w3.org/1999/02/22-rdf-syntax-ns#type' and
        resources.id = subclass and superclass = lClassInstance;
```

In addition to the basic subclassing, we provide support for other inferences in the query. For instance, if our standing query asks for instances of class man, and we get a triple that says that John is the father of Mary, we want John to satisfy this query. To achieve this, we need to peek at the datastream and insert into it some inferred triples. We have implemented an intermediate stream handler which uses the following continuous query to inspect each triple.

```
select feedbackSub, feedbackPred, feedbackOb
  from reasoner.feedback
  [RANGE BY _interval_ seconds
   SLIDE BY _slide_ seconds]
```

We store the subject, predicate and object of the current triple in variables $feedbackSub$, $feedbackPred$ and $feedbackOb$ respectively. The following reasoning types are handled.

rdfs:range. If we have an incoming triple "sub pred obj" and from the ontology we know that "pred rdfs:range x", then it implies that *obj rdfs:type x*. Now if x is the class of concern or a subclass of the class of concern, we would want to detect obj as an instance of concern even though the triple "obj rdfs:type x" never appeared in our stream. The intermediate stream handler inspects the input stream and injects the additional implied triple. The following static query is used to detect if the range of the predicate in any of the triple statement is in the database.

```
select prop.class
  from resources prop, resources pred, rangeinfo
  where pred.class = 'feedbackPred' and
        prop.id = rangeid and
        pred.id = predrange;
```

rdfs:domain. If we have an incoming triple "sub pred obj" and from the ontology we know that "pred rdfs:domain x", then it implies that *sub rdfs:type x*. If x is the class of concern or a subclass of the class of concern, we would want to detect sub as an instance of concern even though the triple "sub rdfs:type x" never appeared in our stream. Again, it is the duty of the intermediate stream handler to inject the additional implied triple. The following static query is used to detect if the domain of the predicate in any of the triple statement is in the database.

```
select prop.class
  from resources prop, resources pred, domaininfo
  where pred.class = 'feedbackPred' and
        prop.id = domainid and
        pred.id = preddomain;
```

rdfs:subPropertyOf. If we have an incoming triple "sub prop obj" and from the ontology we know that "prop rdfs:subPropertyOf superProp", then it implies that *sub superProp obj*. If we also know that "superProp rdfs:range x" from the ontology, we can detect *obj rdfs:type x*. Similarly we can infer for rdfs:domain of "superProp". If x is a class of concern or one of its subclasses, we can detect obj (or sub in case of rdfs:domain) as an event of interest even though the triple "obj rdfs:type x" never appeared in the stream. In this case, the intermediate stream handler injects both the above additional triples - one for superproperty and one for range (or domain) of superproperty. The query used to detect superproperties is as follows.

```
select superprop.class
  from resources superprop, resources subprop, propertyTree
  where subprop.class 'feedbackPred' and
        subprop.id = subproperty and
        superprop.id = superproperty
```

owl:inverseOf. If we have an incoming triple "sub prop obj" and from the ontology we know that "prop owl:inverseOf invProp", then it implies that *obj invProp sub* Now if we know that "invProp rdfs:range x" from the ontology, we can detect *sub rdfs:type x* If x is a class of concern or one of its subclasses, we can detect "sub" as an event of interest even though the triple "sub rdfs:type x" never appeared in the stream. In this case, the intermediate stream handler injects both the above additional triples - one for inverse property and one for range (or domain) of inverse property. The query used to detect inverse properties is as follows.

```
select inv.class
  from resources inv, resources pred, inverseinfo
  where pred.class = 'feedbackPred' and
        inv.id = inverse and
        pred.id = prop
```

The intermediate stream handler tries to enhance the stream instead of trying to replace triples. The stream that is sent to the event detection stage contains all the triples that appeared in the original input stream. It also includes the additional triples inferred and injected by the intermediate stream handler. The original triples are kept in the stream for possible enhancements to our event detection stage in the future.

4 Evaluation and Results

4.1 Building a Jena Server

The default way of doing "rdfs:subClassOf" reasoning on streaming knowledge would have been to use an existing Semantic Web reasoner and do the reasoning process at runtime. We developed an application - which we will refer to as Jena Server - using Java RMI technology that uses Jena to store the RDF graph in main memory, listens to incoming triples, adds them to the RDF graph and infers whether the subject of the current statement is an instance of the class of concern.

4.2 Building a Hashtable based Server

We tried another approach to evaluate what effect pre-computing ontological relationships and storing them in some fast access data-structures has compared to the using a stream database application. In this approach we built a server application - which we will refer to as Hash Server - using Java RMI technology. The Hash Server uses Jena to pre-compute the class-subclass relationships and stores them in a hashtable in main memory. The key of the hashtable is a class and the value is a set of classes that the key class is a subclass of, directly or via a transitive relationship.

4.3 Datasets

To validate our approach, we have covered different ontology sizes from 118 KB to 23.1 MB. The ontologies have different numbers of subclasses from 49 to 56,731 and with different depths of subclass tree from 2 to 9.

Ontologies

We used ontologies of different sizes for our experimental purpose. The testing was done for the following four ontologies.

1. The Spire project ontology for classification of plants [27] - This contains a set of ontologies classifying various plants and invasive species. The total size of ontologies used for our experiments was about 11991 KB with 58,852 different entries in resources table (i.e. rdf:resource entries) and 498,110 class-subclass relationships (including transitively derived relationships).
2. http://www.cyc.com/2004/06/04/cyc - The ontology size for this was about 23.1 MB. It had 62,419 records in the resources table and 985,925 class - subclass relationships (including transitively derived relationships).
3. http://www.mindswap.org/2004/multipleOnt/FactoredOntologies/ // FactoredBeer/human_activities_partition3.owl - This has an ontology size 1115 KB. The number of class-subclass relationships is 12,157 (including transitively derived relationships).
4. http://www.mygrid.org.uk/ontology - This is an ontology of size 118 KB and had 5106 class-subclass relationships (including transitively derived relationships).

Classes of concern

We use the term class of concern for the class whose instances we are trying to detect. We tested our system for the following classes.

- http://spire.umbc.edu/ontologies/InvasivesOntology.owl#ThingOfConcern - This class has 3014 subclasses with a maximum depth of three.
- http://spire.umbc.edu/ontologies/EthanPlants.owl#Plantae - This class has 56,731 subclasses as per our ontologies. The subclasses have a maximum depth of nine.
- http://spire.umbc.edu/ontologies/EthanPlants.owl#Digitaria - This class has 67 subclasses as per our ontologies. The maximum depth for subclasses is two.
- http://spire.umbc.edu/ontologies/EthanPlants.owl#Tracheobionta - There are 40,080 subclasses for this class, with a maximum depth of subclasses equal to eight.
- http://spire.umbc.edu/ontologies/EthanPlants.owl#Asteridae - This class has 12,538 subclasses. The maximum depth of subclasses is eight.
- http://www.cyc.com/2004/06/04/cyc#Individual - This class has nearly 25,000 subclasses with the largest subclass depth of typically seven.

– http://www.cyc.com/2004/06/04/cyc#EmbryophyteSubkingdom - This class has 49 subclasses and the maximum depth for subclasses is 2.
– http://www.mindswap.org/2004/multipleOnt/FactoredOntologies/FactoredBeer/human_activities_partition3.owl#Top3 - This class has 1108 subclasses with maximum depth for subclasses equal to 3.

4.4 Comparisons

Experimental Setup

We compared the performance of our approach with the two alternatives described above. The experiments were carried out on a Dell Inspiron 1405 laptop machine with Intel Pentium Dual Core Mobile T2080 1.73 GHz processor with 2 MB processor cache and 2 GB RAM. The machine ran the kubuntu 6.0 linux distribution. Our Streaming Knowledge Base implementation used TelegraphCQ installation 2.0 and Jena framework installation 2.5.3.

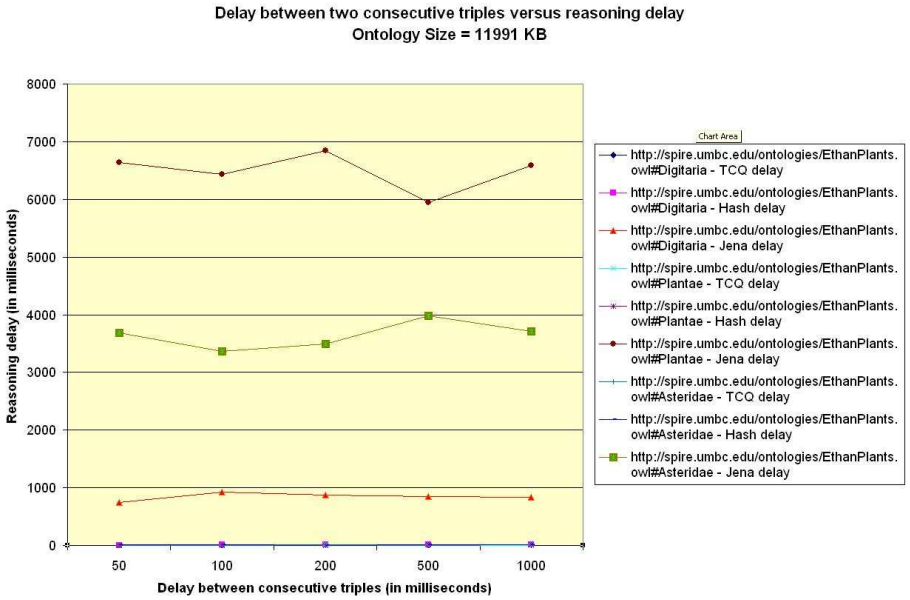

Fig. 1. Comparison of Jena-server, Hash-server and our approach for classes with 67 , 56731 and 12538 subclasses

Calculation of delay

In all approaches, the delay is calculated as follows. We send a stream of triples as input with a fixed time interval - say x milliseconds - between two consecutive

triples. If there are n triples in the input stream then the last triple will be delivered after a gap of (n-1)*x milliseconds after the first triple. Ideally, in case of no processing delay on part of the stream KB, the last response should arrive immediately after the last triple is delivered. However there is some processing delay - say d per triple. So the last triple will be sent after a delay of (n-1)*(x+d) and we will get a response to the last triple after a delay of (n-1)*(x+d) + d. We measure the time interval between sending the 1st triple and getting a response to the last triple, say D. From this we can compute d.

$$
\begin{aligned}
D &= (n-1) * (x+d) + d \\
&= n * x + n * d - x - d + d \\
&= (n-1) * x + n * d \\
d &= (D - (n-1) * x)/n
\end{aligned}
$$

As we can see in Figure 1 the processing delay introduced by TCQ approach and Hash-server is very negligible. The processing delay introduced by Jena is fairly large in comparison. However it's just about the same for same class irrespective of the time interval between two consecutive triples i.e. processing delay does not depend on the interval between incoming triples.

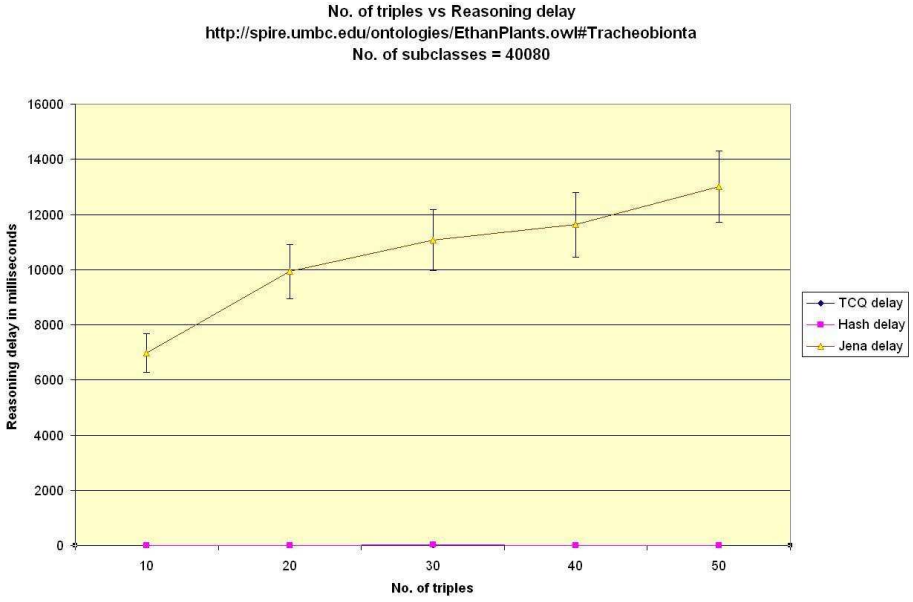

Fig. 2. Comparison of Jena-server, Hash-server and our approach for class with 40080 subclasses

Figure 2 plots the processing delay against the number of triples sent through the stream for one of the ontologies. We can see that as the number of triples increases, the processing delay for TCQ approach and Hash-server is still negligible, however it increases significantly for Jena-server. More over, higher the number of subclasses of the class of concern, greater is the reasoning delay for Jena i.e. as more triples arrive, the reasoning for Jena becomes slower whereas that for TCQ approach and Hash-server still produce an almost real-time response. Similar results are obtained for other ontologies as well.

The main reason for degrading performance of Jena is that it adds every new resource to its memory graph and does inferences on the fly. Reasoning over larger graphs takes more time.

Memory Usage Comparisons

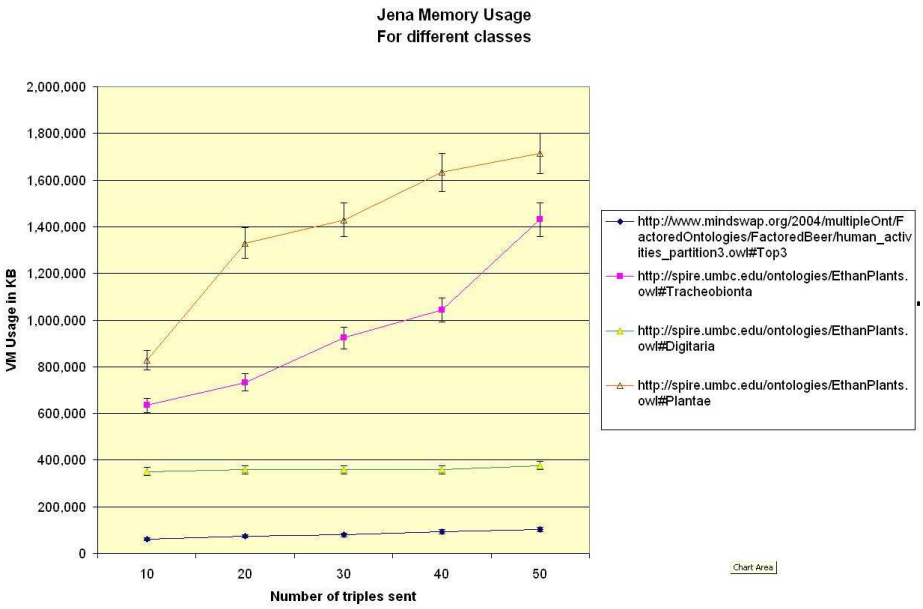

Fig. 3. Virtual memory use for Jena for different classes.

Our experiments indicated that as expected, the virtual memory usage for Jena server keeps increasing as more triples are sent to it. The memory usages for the hashtable approach and the TelegraphCQ approach remain fairly constant. The memory usage for TelegraphCQ approach is calculated by adding the memory usage of the telegraph backend to that of the 2 fetch queries required by the event detector and intermediate stream handler.

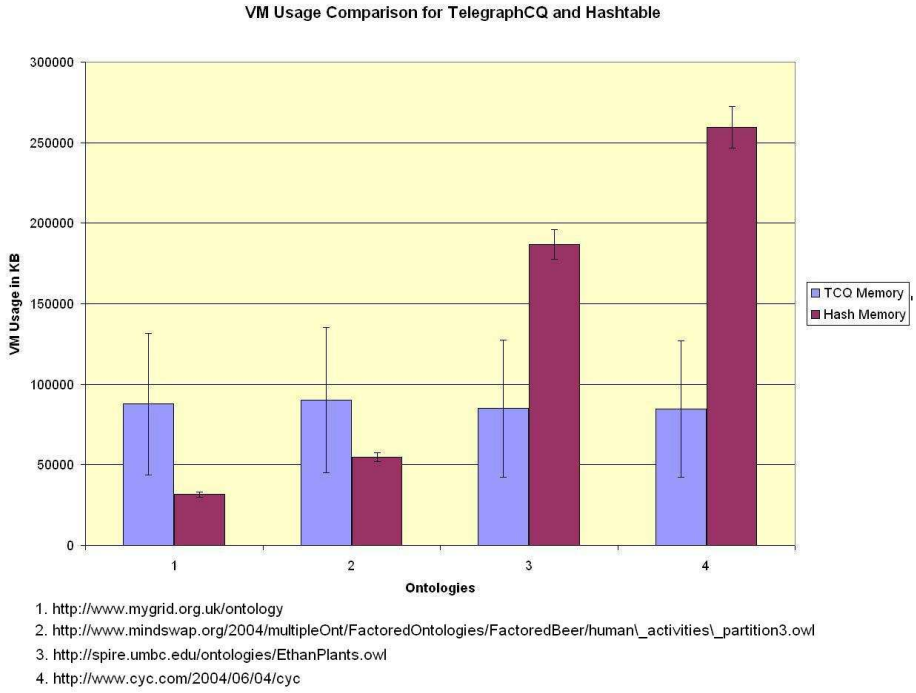

Fig. 4. Virtual memory Use for TelegraphCQ and Hashtable for different classes.

Figure 3 shows the virtual memory usage for Jena for different classes on the left graph. It shows that the memory usage for classes with more subclasses is higher. Notice that even though the class *part:Top3*[2] has more subclasses (1108) than the class *ethan:Digitaria*[3] (67), it uses less memory since the ontology it belongs to is considerably smaller in size. So the virtual memory usage for Jena approach depends on the number of subclasses of the class as well as the size of ontology.

Figure 4 shows the virtual memory usage comparison between the TelegraphCQ and Hashtable approaches. The high Y-error bars on the TelegraphCQ series are due to the fluctuations in memory usage of the telegraphCQ backend. It shows that the virtual memory usage for TelegraphCQ approach is fairly constant for all ontologies. However, the memory usage for Hashtable approach increases with the number of class-subclass relationships. So for larger ontologies the Hashtable approach consumes more memory as it holds the class-subclass relationships in main memory as compared to TelegraphCQ which uses disk stor-

[2] @prefix part: ¡http://www.mindswap.org/2004/multipleOnt/FactoredOntologies/-FactoredBeer/human_activities_partition3.owl#

[3] @prefix ethan: ¡http://spire.umbc.edu/ontologies/EthanPlants.owl#

age. This clearly shows the advantages of building stream reasoners on top of stream databases which are optimized for memory/disk data management.

As we can see from the graphs that as the ontology size increases, so does the reasoning time taken by traditional reasoners. If the class of concern has more subclasses, the reasoning time increases even faster. With the TCQ approach and Hash-server approach we are already pre-computing the subClassOf relationships. Hence the parameters ontology size and number of subcasses for class of concern have very little impact on their processing time. This experiment shows the value of pre-processing the ontology. It is very useful in streaming scenarios as the knowledge source is not slowed down by the server and we get almost real-time event detection.

5 Conclusions and Future Work

In this paper we introduce the problem related to scalably handling queries over semantic data streams. We show that traditional semantic KBs cannot handle stream systems, but that an approach that leverages the work done on stream data systems can be effective. We show empirical results to support this hypothesis. In ongoing work, we are adding support for handling other reasoning types in OWL. Some of the reasoning capabilities from OWL Lite, that can be easily added are those for *owl:equivalentClass*, *owl:sameAs* and *owl:equivalentProperty*.

6 Acknowledgements

Partial support for this work was provided by MURI award FA9550-08-1-0265 from the Air Force Office of Scientific Research and National Science Foundation award ITR 0326460.

References

1. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed diffusion: a scalable and robust communication paradigm for sensor networks. In: MOBICOM. (2000) 56–67
2. Pon, R., Batalin, M.A., Chen, V., Kansal, A., Liu, D., Rahimi, M.H., Shirachi, L., Somasundra, A., Yu, Y., Hansen, M., Kaiser, W.J., Srivastava, M.B., Sukhatme, G.S., Estrin, D.: Coordinated static and mobile sensing for environmental monitoring. In: DCOSS. (2005) 403–405
3. Kulik, J., Heinzelman, W.R., Balakrishnan, H.: Negotiation-based protocols for disseminating information in wireless sensor networks. Wireless Networks **8**(2-3) (2002) 169–185
4. Chen, H., Finin, T., Joshi, A.: Semantic web in in the context broker architecture. In: Proceedings of PerCom 2004. (March 2004)
5. Lee, C.H.J., Bonanni, L., Espinosa, J.H., Lieberman, H., Selker, T.: Augmenting kitchen appliances with a shared context using knowledge about daily events. In: Intelligent User Interfaces. (2006) 348–350
6. Liu, H., Singh, P.: Conceptnet: a practical commonsense reasoning toolkit. BT Technology Journal **22**(4) (2004) 211–226

7. Singh, P., Lin, T., Mueller, E.T., Lim, G., Perkins, T., Zhu, W.L.: Open mind common sense: Knowledge acquisition from the general public. In: CoopIS/DOA/ODBASE. (2002) 1223–1237
8. Agarwal, S., Joshi, A., Finin, T., Yesha, Y., Ganous, T.: A Pervasive Computing System for the Operating Room of the Future. Mobile Networks and Applications **12**(2/3) (December 2007) 215–228
9. Guo, Y., Pan, Z., Heflin, J.: An evaluation of knowledge base systems for large owl datasets. In: The Semantic Web - ISWC 2004, Springer-Berlin/Heidelberg (2004) 274–288
10. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, New York, NY, USA, ACM Press (2002) 1–16
11. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. In: SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (1992) 321–330
12. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. The VLDB Journal **12**(2) (2003) 120–139
13. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S.R., Reiss, F., Shah, M.A.: Telegraphcq: continuous dataflow processing. In: SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (2003) 668–668
14. Broekstra, J., Kampman, A., van Harmelen, F.: Sesame: A generic architecture for storing and querying rdf and rdf schema. In: ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web, London, UK, Springer-Verlag (2002) 54–68
15. Jang, M., Sohn, J.C.: Bossam: An extended rule engine for owl inferencing. In: Rules and Rule Markup Languages for the Semantic Web, Springer-Berlin/Heidelberg (2004) 128–138
16. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006). Volume 4130 of Lecture Notes in Artificial Intelligence., Springer (2006) 292–297
17. Horrocks, I.: Using an expressive description logic: FaCT or fiction? In: Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98). (1998) 636–647
18. Haarslev, V., Möller, R.: Racer: A Core Inference Engine for the Semantic Web. In: Proceedings of the Second International Workshop on Evaluation of Ontology-based Tools. (2003) 27–36
19. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical owl-dl reasoner. Web Semant. **5**(2) (2007) 51–53
20. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, New York, NY, USA, ACM Press (2004) 74–83
21. Bechhofer, S., Volz, R., Lord, P.: Cooking the Semantic Web with the OWL API. In: Proceedings of the First International Semantic Web Conference, Springer (2003) 659–675 Lecture Notes in Computer Science 2870.

22. Zou, Y., Finin, T., Chen, H.: F-owl: An inference engine for semantic web. In: Formal Approaches to Agent-Based Systems, Springer-Berlin/Heidelberg (2004) 128–138
23. Lassila, O.: Enabling semantic web programming by integrating rdf and common lisp (2001)
24. Janik, M., Kochut, K.: Brahms: A workbench rdf store and high performance memory system for semantic association discovery. In: Proceedings of the Semantic Web and Policy Workshop, Galway, Ireland (2005)
25. Beckett, D.: The design and implementation of the redland rdf application framework. In: WWW '01: Proceedings of the 10th international conference on World Wide Web, New York, NY, USA, ACM Press (2001) 449–456
26. : Rdf triples definition : http://www.w3.org/tr/rdf-concepts/#section-triples
27. : Plants classification : http://spire.umbc.edu