Breaking the Black Box - Using Background Knowledge for Efficient Stream Reasoning

Birte Glimm and Markus Brenner

The University of Ulm, Germany
{birte.glimm, markus.brenner}@uni-ulm.de

Abstract. Current approaches to stream reasoning neglect knowledge about the system as a whole. We present first steps towards self-describing streams by outlining a possible definition of the data produced by different streams. We give an outlook on future paths and how such descriptions can be used to improve reasoning about the streamed data.

1 Introduction

Building artificial intelligences and, more specifically, companion systems, requires technologies to capture human knowledge about the domain that the system will interact with. The semantic web provides us with technologies like OWL and RDF, which can be used to build machine usable knowledge bases (also called ontologies) for this purpose. By modeling explicit knowledge, a reasoner can be used to derive additional, implicit information.

Similar to how classical relational databases store data, triple stores are used to store knowledge bases and to provide access via a query language, namely SPARQL. Modern triple stores use a technique called materialization to speed up data access by making implicit knowledge explicit. Doing so is costly and, therefore, many triple stores assume only infrequent updates to the stored data (e.g. once every other week).

Now assume a system, which uses some domain about fitness to provide users with assistance for planning and executing a workout specific to their needs. Such a system will frequently receive changes to its world view (for example information that a workout has been completed) from different sources (sensors, other system components, . . .). To cope with this, a knowledge base needs to provide reasoning services on dynamic data.

Stream reasoning is the idea of combining triple stores with data streams, which provide changes to the stored knowledge. The goal is to be able to perform reasoning on vast amounts of data in real time. An often used example to visualize the concept is a traffic control system, which receives data from thousands of sensors and is supposed to reason on the actual traffic conditions.

In the context of our fitness application, a stream of data might come from a pulse sensor. With additional knowledge about the physique of different age groups and the age of the current user, the stream reasoning system might deduce, whether the current amount of exertion is appropriate for an exercise.

Current stream reasoning approaches handle the streams of data as if they were black boxes. They assume no structure on the incoming data and utilize sliding windows to make the calculation of the materialization bearable, combined with some other approaches [6, 2]. In this paper, we outline the concept of self-describing streams: streams aware of the types of data they contribute to the system, allowing for further optimizations in the materialization algorithms.

Section 2 introduces the necessary basics for this paper. Section 3 introduces our contribution and outlines the future direction of our work. Section 4 provides a short conclusion and outlook.

2 Preliminaries

The following sections of this paper introduce the basic concepts needed to understand our approach.

2.1 Semantic Web

Originating from the idea of enriching web pages with additional semantic information, the semantic web provides us with technologies for the construction of knowledge bases for use with different kinds of applications.

Knowledge is written down in RDF triples of the form

```
subject predicate object .
```

denoting, that the *subject* is related to the *object* by a property *predicate*.

By defining semantics and special keywords, we can, for example, express that someone is a man and that this man is married to someone else

```
John rdf:type Man.
John marriedTo Mary.
```

This example makes use of the predefined RDF keyword *rdf:type*, which expresses that the *subject* of the triple belongs to some group or class of things we denote by the *object* of the triple.

RDF only defines simple semantics. RDF Schema (RDFS) allows more advanced constructs. E.g., it is common knowledge that every thing we identify as a *Man* (that is, it belongs to the class *Man*) is intuitively also a *Person*. RDFS provides the keyword *rdfs:subClassOf*, which expresses exactly this relation:

```
Man rdfs:subClassOf Person.
```

The Web Ontology Language (OWL) further extends the semantics of the triples, allowing for the definition of complex classes, e.g. using a disjunction as in statements like "every person is a man or a woman". OWL defines different subsets of itself, differing in the provided reasoning features.

The triples of an ontology can be divided into different parts or boxes, the most common being the T-Box and the A-Box. Without going into too much detail, the T-Box is analogous to the schema of a relational data base, providing terminological knowledge, and the A-Box can be interpreted as the actual data, providing assertional knowledge.

Knowledge bases consist of explicitly stated information, such as the triples above. They also have some implicit information: For example combining the triples about John with our statement about Man and Person results in the information, that John is also a Person. Obtaining such implicit information is the job of a reasoner.

More on RDF can be found in [7] and more on OWL in [8].

2.2 Triple Stores

Similar to how relational databases are used to store data for common applications, triple stores are used to store the triples of a knowledge base. To enable easy access to the stored knowledge, they also provide a query language. The most widely used one is SPARQL [4]. The interplay of a triple store and an application is roughly depicted in Figure 1. Similar to SQL it allows the application to specify what data should be retrieved by matching a query onto the triples. As opposed to relational data, a knowledge base should not only provide information explicitly stored, but also information derived using reasoning.

Most of the time reasoning is costly and queries are usually of a rather high frequency, while often requesting similar data. Therefore common triple stores employ a technique called materialization to limit the actual reasoning to a minimum. This technique uses a reasoner to derive all implicit information hidden in the explicit triples (such as the "John is a Person" information in the example above) and adds them to the triple store. The store still keeps track of which triples where originally there and which were added to the materialization. When the information in the triple store changes, because new triples are added or removed, the materialized triples are dropped and the store uses the reasoner again to add implicit information.

Materialization itself is costly due to the reasoning involved and therefore the data is only updated very infrequently. Furthermore, estimating the effects of an update is already very costly and therefore recalculating the complete materialization is often the more reasonable approach.

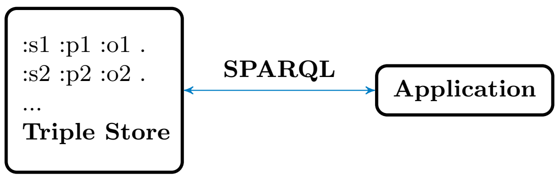

Fig. 1. Triple store interacting with an application

2.3 Stream Reasoning

Stream reasoning [10] is the idea of processing vast amounts of steadily incoming data (for example from sensors) by using background knowledge from ontologies to reason about the meaning of the data. Consider for example the setting of a big factory, in which several thousand sensors monitor different parts of the production process. A stream reasoner could use this data to continuously reason about the productivity of the factory, check for errors and potential problems etc.

In contrast to static triple stores, where all data is equally relevant, streaming data becomes obsolete after some time when new data arrives. Therefore reasoning needs to be done continuously as the data changes, which means that the maintenance of the materialization is a major problem.

Furthermore, instead of one-time queries to the knowledge base, stream reasoning considers on-going queries, which keep producing results as new data is inserted. In our factory setting we usually don't want to query only once if everything is alright, but we want to keep asking this question.

C-SPARQL [1] is one approach to stream reasoning and an extension to the SPARQL Query Language. It processes streams as RDF streams [3], which consist of RDF triples annotated with timestamps, and evaluates the streaming data over its modified query language. A more detailed overview of the general architecture of a stream reasoning system can be found in [3] and [9].

To the core features of C-SPARQL belongs the notion of windows. A window essentially describes which triples are evaluated with the query at each point of time. It consists of a time range r and a time step s. Every s time units, the window slides forward and is evaluated over the triples seen in the last r time units. This helps to reduce the amount of calculation needed in the materialization step. In particular, [1] uses two optimizations: First of all, the materialization is maintained via the general algorithm presented in [6]. Furthermore, [2] presents an extension, which uses the fact, that the triple store already knows upon arrival of a triple, when it will be discarded again.

3 Self-Describing Streams

We previously introduced the concept of stream reasoning and RDF streams. Current works do not make any assumption about the data a stream contains. Instead, streams are handled as black boxes producing arbitrary triples. Usually, this is not the case. When constructing an application using a stream reasoner, we already have knowledge about the data a stream will produce. Furthermore we can assume that connecting a data stream to the system will not occur very often and in particular all streams will most likely be connected during the set-up of the system. We present an approach to specifying knowledge about the streams such that a stream reasoner can use this information to optimize the calculation of the materialization. The idea of self-describing systems (or components) is not new. For example, in the semantic web area, SPARQL Service Descriptions

[11], allow SPARQL endpoints (interfaces, which provide access to a triple store via SPARQL) to describe their capabilities and the data they provide.

First of all, we assume that the provided data can be partitioned into distinct events. Such an event might be a new pulse reading from a sensor or perhaps a more sophisticated event from some preprocessing component, which delivers more complex data. Furthermore, even though the actual data of an event might differ, it is possible to group similar events. In the pulse sensor example, the actual pulse readings only differ in the reading itself.

We therefore introduce the notion of *stream statement*. A *stream statement* describes classes of events and consists of two parts: First, a definition of one or more triples (forming an A-Box assertion) with template variables. Actual events replace these template variables with actual elements. E.g., the triple

```
sensor1 pulseValue t1.
```

with the template variable t1 could be part of a *stream statement* and a corresponding event could produce the triple

```
sensor1 pulseValue "80"^^xsd:integer.
```

We make the restriction to A-Box assertions, as streams should mostly only deliver actual data to the system.

The second part of a *stream statement* provides restrictions on the template variables, by specifying the elements which can replace them. These can be literals (such as the "80" above, which is typed as an XML-Schema datatype to clarify its semantics), fresh elements never seen before, or elements which have already occurred in the ontology or in other streams.

Each template variable can only have one specified source. If an event can reference an element from several sources, it will be described using several different *stream statements*, which will be grouped in a *statement group*.

Stream statements can be used as follows: Upon connecting streams and triple store, each stream will provide all contributed statements. The triple store can then decide, how it will react upon receiving an actual event through a stream.

At runtime, whenever a stream experiences an event, it will tell the triple store to which statement group it belongs. Note that the stream cannot decide efficiently if an element is new to the triple store, even when it knows which source the element will have. Therefore the stream only declares which statement group the event belongs to and the triple store has to decide on the actual stream statement, which should be an easy lookup on the nature of the provided element. Therefore stream and triple store together can classify incoming events efficiently, allowing for optimized materialization routines depending on the incoming element.

3.1 Where We Are so Far

So far we are still exploring the idea of self-describing streams. The definition and syntax of *stream statements* needs to be formalized. The usefulness of the approach will have to be verified first as well, but declaring the information provided by a stream provides at least the possibility to speed up the materialization by neglecting parts of the knowledge base, which clearly will not contribute new triples in combination with the new data. This is possible through the use of existing modularization algorithms such as [5].

We expect further optimizations to be possible, also empowered by the fact that preprocessing the stream descriptions happens only at setup, allowing for computationally expensive algorithms to be used.

4 Conclusion

We have presented a new approach to formalizing knowledge about streams in stream reasoning. Careful considerations give rise to hope for a speed up of the materialization time. Should the presented approach lead to a significant improvement, then perhaps future stream reasoners can consider different notions of time and allow for different triple lifetimes, e.g. keeping data about users until explicitly deleted, while still dropping pulse data frequently.

References

1. Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: a continuous query language for RDF data streams. *Int. J. Semantic Computing*, 4(1):3–25, 2010.
2. DavideFrancesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. Incremental reasoning on streams and rich background knowledge. In *The Semantic Web: Research and Applications*, volume 6088 of *LNCS*, pages 1–15. Springer, 2010.
3. Emanuele Della Valle, Stefano Ceri, Davide Francesco Barbieri, Daniele Braga, and Alessandro Campi. *A first step towards stream reasoning.* Springer, 2009.
4. Steve H. Garlik, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/.
5. Bernardo Cuenca Grau, Ian Horrocks, Yevgeny Kazakov, and Ulrike Sattler. Just the right amount: extracting modules from ontologies. In *Proceedings of the 16th Int. Conf. on World Wide Web*, pages 717–726. ACM, 2007.
6. Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. Maintaining views incrementally. *ACM SIGMOD Record*, 22(2):157–166, 1993.
7. Frank Manola and Eric Miller, editors. *RDF Primer.* W3C Recommendation. W3C, February 2004.
8. W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview.* W3C Recommendation, 27 October 2009. Available at http://www.w3.org/TR/owl2-overview/.
9. Heiner Stuckenschmidt, Stefano Ceri, ED Valle, and Frank Van Harmelen. Towards expressive stream reasoning. In *Proceedings of the Dagstuhl Seminar on Semantic Aspects of Sensor Networks*, page 241, 2010.
10. Emanuele Della Valle, Stefano Ceri, Frank van Harmelen, and Dieter Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24(6):83–89, 2009.
11. Gregory Todd Williams. SPARQL 1.1 Service Description, October 2010.