A Benchmarking Framework for Stream Processors[⋆]

Andreas Moßburger, Harald Beck, Minh Dao-Tran, and Thomas Eiter

Institute of Information Systems, Vienna University of Technology
Favoritenstraße 9-11, A-1040 Vienna, Austria
{mossburger,beck,dao,eiter}@kr.tuwien.ac.at

1 Introduction

Stream Processing/Reasoning, an active research topic [5], has been picked up by different communities which developed a diversity of stream processors/reasoners. This however makes empirical evaluation and comparison of these engines a non-trivial task [4]. Different classes of those engines work on different formats of input data, use different languages to formulate queries, evaluate these queries using different semantics and produce different formats of output. To be able to compare such engines, a benchmarking framework that can cope with this wide diversity is needed.

This paper proposes a generic architecture for generating/gathering, streaming data, for evaluating different stream processors/reasoners and for running those evaluations.

We show how our framework can be used with the GTFS domain to evaluate the two representative RDF Stream Processing (RSP) engines C-SPARQL [3] and CQELS [7], and the Spark engine,[1] a powerful tool set for implementing stream processing applications that is widely used in Big Data. In that, we use the Answer Set Programming (ASP) engine clingo[2] as a base-line for comparing the results.

2 Challenges

The following challenges arised in comparing the aforementioned engines:

– The engines work on different data formats: RSP engines need data in some RDF serialization, Spark reads CSV files and clingo needs facts in ASP syntax.
– To provide not only fair comparisons but also reproducible tests, live streaming data has to be captured, so that it can be replayed in a reproducible manner.
– There is no unified way to query the engines: C-SPARQL and CQELS are queried by SPARQL-like queries, while Spark can be accessed by writing Scala functions using its API; clingo instead requires one to write logic programs.
– Output has to be generated in a unified format for easy comparison.

[⋆] This research has been supported by the Austrian Science Fund (FWF) projects P24090, P26471, and W1255-N23.

[1] http://spark.apache.org/

[2] https://github.com/potassco/clingo

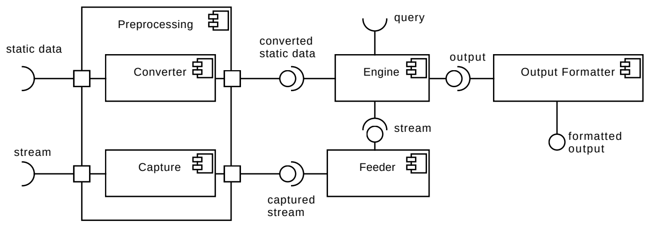

Fig. 1: Component diagram of modules

3 Framework Architecture

We propose a modular architecture composed of small components with clearly defined tasks and interfaces. Each module tackles one of the challenges described above. This architecture is generic enough to be implemented using a wide variety of technologies, from small scripts interacting together to classes in rather monolithic projects. To enable unified access to different engines, lightweight wrappers are utilized.

Figure 1 shows the modules of our architecture and the resulting data flow. The *converter* and *capture* module can be seen as two parts of a *preprocessing* module which is responsible for converting data to a usable format.

Converter is responsible for converting any static data from the provided format to a format that can be read by an engine. In some cases this module may even be omitted.

Capture extracts relevant data from a data stream and stores it, allowing reproducible evaluations. The format of the stored data should be generic, so that only minimal conversions are necessary for a particular engine. Additionally, timing information of the captured data should be stored, so it can be played back authentically.

Feeder is responsible for replaying the captured streaming data to the engines. It allows arbitrary fine control over the streaming process. Data may be streamed using authentic or artificial timing, like streaming a certain amount of data per time unit.

Engine wraps the evaluated engine. Wrappers provide a standardized way of accessing/outputting data for different engines but are not allowed to affect their performance.

Output Formatter converts the output data from different engines to a canonical form.

4 Show Case

4.1 General Transit Feed Specification

A public transport scenario offers ample opportunities for interesting data processing and reasoning tasks. In addition, the open data movement lead to free availability of public transport data for many cities worldwide. Most public transport data is published in the GTFS (General Transit Feed Specification) [1] format developed by Google and the Portland TriMet transit agency. A GTFS feed provides information suitable for trip planning. Moreover, the standard leaves enough room for extensions beyond these capabilities. While GTFS specifies static data useful for trip planning, its accompanying GTFS-realtime extension was developed to describe real-time data.

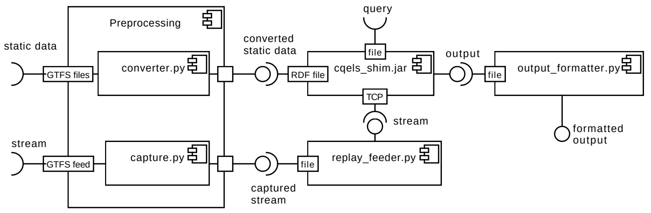

Fig. 2: Component diagram for GTFS and CQELS

GTFS-realtime provides *TripUpdates* and *VehiclePositions*. A *TripUpdate* represents a change to a timetable and consists of possibly multiple delays or new arrival times for single stops of a trip. A *VehiclePosition* tells the position of a vehicle relative to a stop.

The public availability of real world data makes GTFS ideal for evaluating stream processing/reasoning engines.

4.2 Implementation

We implemented the components as Python scripts which communicate using files and TCP connections. This highly transparent working model allows to exchange single components easily and to examine intermediate results. By utilizing powerful and well documented libraries for handling GTFS and RDF data a low entry barrier is maintained. The following scripts are provided (our code is available at [2]):

– `gtfs-converter.py` and `gtfs-capture.py` implement the converter and capture modules, resp. These scripts are specific to the GTFS use case. All other scripts and programs are generic and do not make any assumptions about the data domain.
– `simple_feeder.py`, `replay_feeder.py` and `triple_to_asp.py` provide implementations of the feeder module.
– `output_formatter.py` covers the output formatter module.

For the engine module, different wrappers were implemented to access the engines. Fig. 2 illustrates instantiating the architecture with scripts/programs to evaluate CQELS.

5 Evaluation

This section briefly summarizes the evaluation of the engine described above using our framework. All the queries used in this evaluation are available at [2].

5.1 Functionality

Our first evaluation was a basic test to determine which features of SPARQL 1.1 are supported by C-SPARQL and CQELS, namely `FILTER`, `UNION`, `OPTIONAL`, arithmetic, aggregation such as `COUNT`, `COUNT DISTINCT`, `MAX`, post-processing with `ORDER BY`, joining. We also tested which features could be replicated with Spark and clingo.

As a result, all mentioned features are supported by C-SPARQL and can be covered by clingo, CQELS lacks support for `UNION`, `OPTIONAL` and some aggregations. Spark misses out only `ORDER BY`.

5.2 Correctness

When comparing different engines, the underlying semantics cannot be ignored. Thus, one needs a highly expressive language to serve as an oracle for correct results via simulating input streams. We chose Answer Set Programming as it offers high expressiveness. Moreover, a number of mature ASP engines have been developed; among them clingo is of particular interest, as it showed top performance in many ASP competitions.

Note that C-SPARQL, CQELS and Spark generally output a multiset of tuples as result. Only in queries with an `ORDER BY` clause, the order of the output is not arbitrary. On the other hand, clingo always outputs a set as result. Because of this discrepancy one might argue that ASP is not suitable to serve as an oracle for comparing the other engines. However, if in practical queries multiple occurrence of a result matters, `GROUP BY` and `COUNT` clauses are used. If a result may occur multiple times but it is not of interest how often, `DISTINCT` is used. Therefore, for this fragment, the result will be a set and using ASP as an oracle is perfectly adequate.

In our run of the evaluation, the results of CQELS and Spark conformed to the results predicted by clingo. C-SPARQL produced correct results too, but was missing a few lines of output, compared to the others.

6 Conclusions

We presented a generic benchmarking framework that allows one to easily plug in her own stream processing/reasoning engine and compare it with stream engines developed by different communities. The evaluation is done on real public transportation data in GTFS format, a data domain that allows practical queries to test different interesting aspects of stream processing/reasoning. We provided a set of queries to test query functionalities and compared the results of C-SPARQL, CQELS, and Spark using an oracle built on clingo. For future work, we plan to plug more engines into the framework and extend the query set to cover more stream processing/reasoning aspects.

References

1. General Transit Feed Specification. `https://developers.google.com/transit/`. [Online; accessed: 2016-09-15].
2. sr_data_generator github repository. `https://github.com/mosimos/sr_data_generator/`. [Online; accessed: 2016-09-15].
3. D. F. Barbieri, D. Braga, S. Ceri, E. Della Valle, and M. Grossniklaus. C-SPARQL: a continuous query language for rdf data streams. *Int. J. Semantic Computing*, 4(1):3–25, 2010.
4. M. Dao-Tran, H. Beck, and T. Eiter. Contrasting RDF Stream Processing Semantics. In *JIST*, 2015.
5. E. Della Valle, S. Ceri, F. van Harmelen, and D. Fensel. It's a streaming world! reasoning upon rapidly changing information. *IEEE Intelligent Systems*, 24:83–89, 2009.
6. M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.
7. D. L. Phuoc, M. Dao-Tran, J. X. Parreira, and M. Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *ISWC (1)*, pages 370–388, 2011.