Bachelorarbeit für Informatik und Wirtschaftsinformatik
LVA-Nr. 184.713

Data Generation for Stream Reasoning Benchmarking

Andreas Moßburger
Matrikel-Nr. 1029147
19. Nov. 2015

Contents

1 Introduction

Motivation Today's digital world generates vast amounts of data every second. While a lot of the produced data is rather static and won't change after it has been produced, there's also a significant amount of data that is only valid for a short time. For other applications, older data may stay valid, but only the most recent data is of interest. In those cases we speak of streaming data opposed to static data. Examples of scenarios, where such streaming data may occur, are sensor networks, transportation networks or social networks.

Problem Handling this rising tide of streaming data is a very active area of research and a lot of different data stream management systems (DSMS), stream processors and stream reasoners have been developed. Unfortunately, the diversity of these approaches make empirical evaluation and comparison of these engines a difficult task. 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. Therefore, a benchmarking framework that can cope with this wide diversity is needed. This benchmarking framework should use data that not only allows evaluation of very simple queries, but also of complex query features. Additionally, the used data should represent real life applications, so the engines can be tested under realistic conditions.

Scope In this work, we consider several different stream processing engines. In Section 3 they are explored in depth. First of all, there are the RDF stream processing engines. Their development was sparked by the idea to extend the powerful reasoning capabilities of semantic web technologies to a streaming context. We work with C-SPARQL [18] and CQELS [22] two of the most prominent candidates from this category. Stream processing obviously is also an important topic in the cluster computing community. High performance and throughput is the focus of the engines developed there. Rapid developments are made in this community and some of the projects enjoy a high influx of contributors. We took a closer look at Apache Spark [2], one of the most active projects in the Apache Software Foundation, as a representative of this group. In order to establish a reliable baseline against which we can compare the results of those stream processing engines we need to provide an oracle which tells the expected result of a query. We implemented this oracle by simulating data streaming statically and evaluating queries with an Answer Set Programming (ASP) [21] solver. ASP provides an universal solving mechanism which can easily cover the capabilities of stream processing engines and is powerful enough to go beyond pure processing into stream reasoning. It allows to define expected results in a clean and formal way.

Proposed Solution In Section 4 we propose a generic architecture for generating, or rather gathering, streaming data for evaluation of different stream processing engines and for running those evaluations. It is highly modular and each module addresses a specific challenge that arises from this task. We show the merits of this architecture in Section 5, by implementing it for a public transport scenario and running several evaluations of C-SPARQL, CQELS and Apache Spark, with clingo [19] serving as an oracle.

2 State of the Art

Linear Road [17] is a benchmarking system for DSMSs. It generates data by simulating a variable tolling system for a fictional net of expressways and scores engines based on the maximum size of this net where they can still meet time and correctness constraints. Because Linear Road was specifically developed for evaluation of DSMSs it is based on a relational data model and not suitable for serving as a more generic benchmarking system. SRBench [24] assesses the capabilities of RDF stream processing engines. It does so, using the LinkedSensorData [10] data set containing data of weather stations in the United States. SRBench provides a functional evaluation of RDF stream processing engines, determining which features of the SPARQL [16] language an engine supports and is therefore not particularly suited for evaluating other engines. LSBench [23] simulates social network activities of users to generate data for its evaluations. It evaluates engines considering functionality, correctness and performance. It also addresses the need of covering technical and conceptual differences of engines, but still limits its scope to linked stream data engines.

3 Technical Background

In this section different stream processing engines are introduced. First we describe engines that work on RDF data using extensions of the SPARQL language. In particular we take a look at C-SPARQL [18] and CQELS [22]. Then we cover Apache Spark [2], a cluster computing framework which offers a high level API for querying and manipulating streaming data. Following that, we address the topic of Answer Set Programming, which we use to determine expected results of queries.

3.1 RDF Stream Processors

The Resource Description Framework data format is one of the core technologies of the W3C Semantic Web standards. It's a very simple data model, representing all data as triples of form $s\ p\ o$, which intuitively can be seen

```
<http://kr.tuwien.ac.at/dhsr/trip/5977790> ns1:hasDirection "0" ;
    ns1:hasStt <http://kr.tuwien.ac.at/dhsr/stoptime/59777901>,
        <http://kr.tuwien.ac.at/dhsr/stoptime/59777902>,
        <http://kr.tuwien.ac.at/dhsr/stoptime/59777903>,
    ns1:isonRoute <http://kr.tuwien.ac.at/dhsr/route/1> .

<http://kr.tuwien.ac.at/dhsr/stoptime/59777901>
    ns1:atStop <http://kr.tuwien.ac.at/dhsr/stop/9303> ;
    ns1:hasArrtime 82860 ;
    ns1:hasDeptime 82860 ;
    ns1:isSeq 1 .

<http://kr.tuwien.ac.at/dhsr/stop/9303> ns1:hasName "NW 5th & Couch MAX Station" .
```

Figure 1: Some RDF data in Turtle serialization format.

```
REGISTER QUERY q01 AS
PREFIX ns1: <http://kr.tuwien.ac.at/dhsr/>
SELECT ?stt_id ?arr
FROM STREAM <http://kr.tuwien.ac.at/dhsr/stream> [RANGE 1s STEP 1s]
WHERE {
    {?stt_id ns1:hasArrived ?arr .}
}
```

Figure 2: Query 01 in C-SPARQL syntax.

as binary *predicates* $p(s, o)$ that relate a *subject* s with an *object* o. A serialization of such a data model can be seen in Figure 1. This simplicity gives rise to a powerful tool set to model various kinds of data. The standard language to query and manipulate RDF data sets is called SPARQL [16]. It has an SQL-like syntax and allows to select triples based on patterns. RDF was conceived for representing static data. However, it can be extended so it can also be used in a streaming context. In this case there might exist some static data in addition to a stream of RDF triples representing rapidly changing data, which might only be valid for a short time frame. There exist multiple projects with the aim to adapt the data processing capabilities of SPARQL over RDF formatted static data to a streaming data context. In the following we take a closer look at C-SPARQL and CQELS.

C-SPARQL C-SPARQL [18] extends SPARQL by adding a `FROM STREAM` clause to the grammar of SPARQL 1.1 which allows to reference an RDF stream instead of an RDF graph i.e. a set of triples, as can be seen in the example in Figure 2. It also introduces the notion of *windows*.
Because streaming data typically is produced at a high rate and valid or relevant only for a short time, usually only recent data is of interest. To

```
PREFIX ns1: <http://kr.tuwien.ac.at/dhsr/>
SELECT ?stt_id ?del
FROM NAMED <http://kr.tuwien.ac.at/dhsr/>
WHERE {
    STREAM <http://kr.tuwien.ac.at/dhsr/stream> [RANGE 1s TUMBLING]
    {?stt_id ns1:hasArrived ?del}
}
```

Figure 3: Query 01 in CQELS syntax.

determine which triples of the stream will be considered when evaluating a query, according windows can be defined. Windows can either be defined by the number of triples (e.g. the 50 most recent triples) or by a time interval (e.g. all triples occurring in the last 5 seconds). C-SPARQL evaluates queries at the end of every window, by taking a union of the triples of the current window and an optional static data set and evaluating the query using a regular SPARQL engine as a black box. Because C-SPARQL works on static snapshots of streaming data it pulls from the stream, we say it uses *snapshot semantics* and operates on a *pull principle*. C-SPARQL is written in Java and can be interfaced with by building a Java application on top of it.

CQELS Like C-SPARQL, CQELS [22] extends the grammar of SPARQL 1.1 to allow references to streams and declaring windows, but does so in the place of a graph pattern in the WHERE clause, as can be seen in Figure 3.
Another main difference to C-SPARQL is that CQELS uses a native implementation for evaluating queries, instead of using a SPARQL engine as black box. This allows to adapt to changes in the input data and improve query evaluation strategies. Additionally it can benefit from caching and encoding of intermediate results. Those performance improvements allow CQELS to work on a push principle opposed to a pull principle like C-SPARQL. That means every incoming triple triggers an evaluation of the query and may generate output immediately instead of evaluation only happening at the end of a window. Just like C-SPARQL, CQELS is written in Java and can be interfaced with by building a Java application on top of it.

3.2 Apache Spark

Apache Spark is a cluster computing framework for large-scale data processing. It can access data from a multitude of distributed storage systems like Hadoop Distributed File System, Apache Cassandra, Apache HBase or Amazon S3. Spark works batch oriented, but Spark Streaming extends those capabilities to stream processing by dividing an input stream into so-called *micro batches*. Streaming data can be ingested using a multitude of technologies common in the big data world like Apache Flume, Apache Kafka

```
val windowed = triple_objects.window(Seconds(1), Seconds(1))
val arrived = windowed.filter(_(1).contains("hasArrived"))
val result = arrived.map(x => (x(0), x(2)))
```

Figure 4: Query 01 in Scala using Apache Spark.

or Amazon Kinesis. Apache Spark makes no assumptions about the underlying data model. Its API exposes the data as collections and offers high level transformation functions like windows, map, reduce, filter and join. Programs for querying or manipulating data can be written in Scala, Java or Python. An example for a query written in Scala can be seen in Figure 4. Although typically Spark applications run on clusters, they can also be run in local mode on a single computer.

3.3 Answer Set Programming

Answer Set Programming (ASP) is a logic-oriented form of declarative problem solving for combinatorial problems. It's based on the stable model semantics [20] and is tailored for Knowledge Representation and Reasoning applications. We use Answer Set Programming in order to provide an oracle which tells us the expected result of a query for each window. To do so, we simulate a *tumbling window* on the streaming data by feeding only the data of a single window at a time into the ASP solver. This gives us an expected result for each window. This approach can be compared to how C-SPARQL implements windowing and then uses a SPARQL engine as a black box. However, since we use ASP only as an oracle this process does not have to happen in real-time. As a solver, we use the Potassco Project's clingo [19], which combines the grounder gringo and the solver clasp into a single monolithic system.

4 Architecture for Data Stream Generator

In this chapter we introduce the architecture we developed for evaluating stream processing engines. In the first section we present the challenges that arise from working with widely different engines and what we have done to meet those challenges. Then we go into detail how the modules of our architecture work together and how each of them addresses the needs described.

4.1 Architecture

When trying to compare multiple stream processing engines, diverse approaches in the engines give rise to a number challenges. The aim of our

architecture is to unify interfaces where possible and bridge differences where needed. First of all, the engines are not all provided with high-level interfaces that make them ready to use for our tasks. C-SPARQL and CQELS are written in Java and can be interfaced with by writing a program instantiating their classes. Apache Spark isn't a stream processor per se, but rather a powerful framework and to utilize its stream processing capabilities a program using its API has to be written. clingo can directly work on input files containing data and queries. The next difficulty is that the engines work on different data formats. The RDF engines need data in some RDF serialization, Apache Spark can easily read simple comma separated data files and clingo needs facts in ASP syntax. Therefore any static data needs to be converted to a format that can be read by an engine. In order to provide not only a fair comparison but also reproducible tests, evaluations have to be run on a fixed data set. Therefore, live streaming data cannot be used, but rather has to be captured so it can be used repeatedly. Additionally, replaying this captured data into the engines has to happen in a reproducible manner. Last but not least, output has to be generated in a unified format for easy comparison.
We decided on a modular architecture composed of small components with clearly defined tasks and interfaces. This allows to easily swap out or extend single modules, in order to fulfill new requirements. Each module tackles one of the challenges described above. This architecture is generic enough to be implemented using a wide variety of technologies, whether as small scripts interacting together or as classes in a rather monolithic project. To enable unified access to the different engines, lightweight wrappers or *shims* are utilized. They read data and queries from clearly defined, universal interfaces, bridging the gaps the out-of-the-box provided interfaces leave.

4.2 Module Descriptions

Fig. 5 shows the modules of our architecture and the resulting data flow. As can be seen, there's one module addressing one of the challenges mentioned in the previous section each. The converter and the capture module can be seen as two parts of a preprocessing module which is responsible for converting data to a usable format.

Converter The converter module is responsible for converting any static data from the provided format to a format that can be read by an engine. It typically takes its input from a file and also outputs a file. Depending on how different the provided format of the data and the format needed by the engine are, this module may be very complex or very simple and in some cases not even needed.

- Input: Static data in domain specific format

- Output: Static data in engine specific format

Capture The capture module extracts required data from a data stream and stores it, so it can be used for evaluations again and again. The format of the stored data should be general enough, so that only minimal conversions have to be done for particular engines. Additionally, timing information of the captured data should be stored, so it can be played back just like it was captured.

- Input: Data stream in domain specific format
- Output: Captured streaming data in generic format

Feeder This module 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 the same timing it was captured with or using custom timing, like streaming a certain amount of data per time unit. It is important that this module can reproduce this timing reliably in order to provide a level playing field to all the engines.

- Input: Captured streaming data in generic format
- Output: Data stream, possibly in engine specific format

Engine This module consists not only of the engine to be evaluated, but also of the wrapper that may have been written to make the engine accessible. This allows to read static data and the query used for evaluation from files. The streaming data is typically received through a TCP socket and results are written to a file. Great care should be taken to not introduce artificial bottle necks in the interface to the engine, e.g. by reading data in an inefficient way or by buffering incoming data by accident.

- Input: Query, data stream and optionally static data in engine specific format
- Output: Results in engine specific format

Output Formatter Different engines may output data in different formats. To allow comparisons of results, it is necessary to bring this output to a canonical form. This module addresses that need.

- Input: Results in engine specific format
- Output: Results in canonical, engine agnostic format

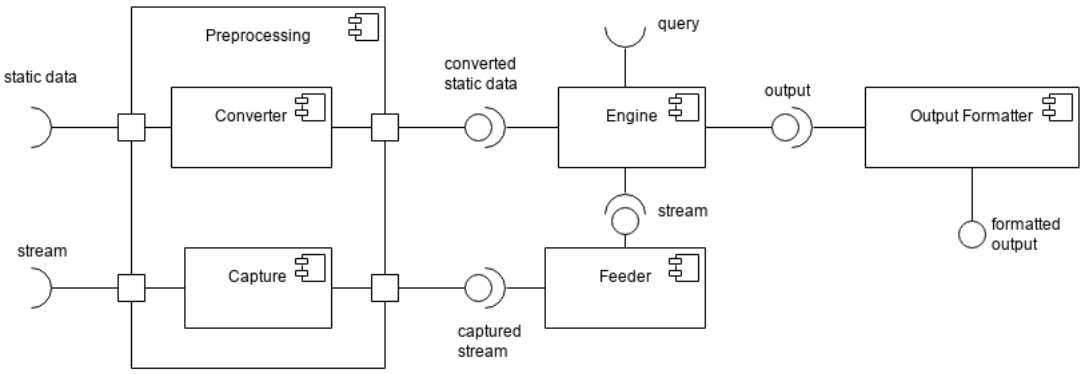

Figure 5: Component diagram of modules

5 Show Case

In this chapter we will illustrate how to apply our architecture to a specific use case. First we introduce the GTFS domain of public transport data. Then we describe how we implemented the architecture described in Section 4 for use with this domain and the engines described in Chapter 3 using a collection of small programs and scripts. Afterwards we depict how to run evaluations with this implementation and how the programs and scripts interact together. Finally we present some evaluations we ran using the GTFS domain.

5.1 GTFS Domain

A public transport scenario offers ample opportunities for interesting data processing and reasoning tasks. In addition, the rise of the open data movement lead to public availability of public transport data for many cities world wide. The vast amount of different data formats used by different transit agencies around the world made the need for a common standard apparent. To meet this need, Google and the Portland TriMet transit agency developed the General Transit Feed Specification (GTFS) [5]. A GTFS feed provides information suitable for trip planning functionality, but the standard leaves enough room for extensions to go beyond these capabilities. A list of publicly available GTFS feeds can be found at the GoogleTransitDataFeed project site [6] or at the GTFS Data Exchange website [7]. While GTFS only specifies static data useful for trip planning, its accompanying GTFS-realtime specification was developed to describe real-time data about updates to the schedule, service alerts and positions of vehicles. The public availability of real world data with decent complexity makes GTFS perfect for evaluating stream processing engines.

```
trip_id,arrival_time,departure_time,stop_id,stop_sequence
AWE1,0:06:10,0:06:10,S1,1
AWE1,,,S2,2
AWE1,0:06:20,0:06:30,S3,3
AWE1,,,S5,4
AWE1,0:06:45,0:06:45,S6,5
AWD1,0:06:10,0:06:10,S1,1
AWD1,,,S2,2
AWD1,0:06:20,0:06:20,S3,3
AWD1,,,S4,4
AWD1,,,S5,5
AWD1,0:06:45,0:06:45,S6,6
```

Figure 6: Part of a stop_times.txt file

5.1.1 Static Data Model

A GTFS feed is provided as a .zip file containing multiple comma-separated values (CSV) files. Each of these files contains entities of a different type. For our needs, the stops.txt, routes.txt, trips.txt and stop_times.txt files are relevant. Fig. 6 shows an example of an stop_times.txt file. We will now describe the files and entities they contain. A complete specification of the files and their fields can be found at the GTFS reference website [8].

Stop A stop entity represents a single stop in the transit network. The only attribute of this entity we use is the stop_name, but latitude and longitude, the identifier (id) of the fare zone and whether wheelchair boarding is possible at the stop might also be of interest.

Route A route represents a line in the transit network. Examples might be the subway line U2 or the tramway line 5. Interesting attributes of a line are its name and the type of the route, meaning is it a subway line, a bus line, or something else. A route can be seen as a set of trips.

Trip A trip represents a single trip on a route, for example the trip on line U2 starting at 10:32 from the stop Karlsplatz towards the stop Seestadt. A trip's attributes may contain the id of the route it is on, the direction the trip is going, whether it is wheelchair accessible or not and whether bicycles can be taken on the trip.

Stop_time A trip consists of several stop_time elements, each describing when the trip is scheduled to reach a stop. The order of the stops is given

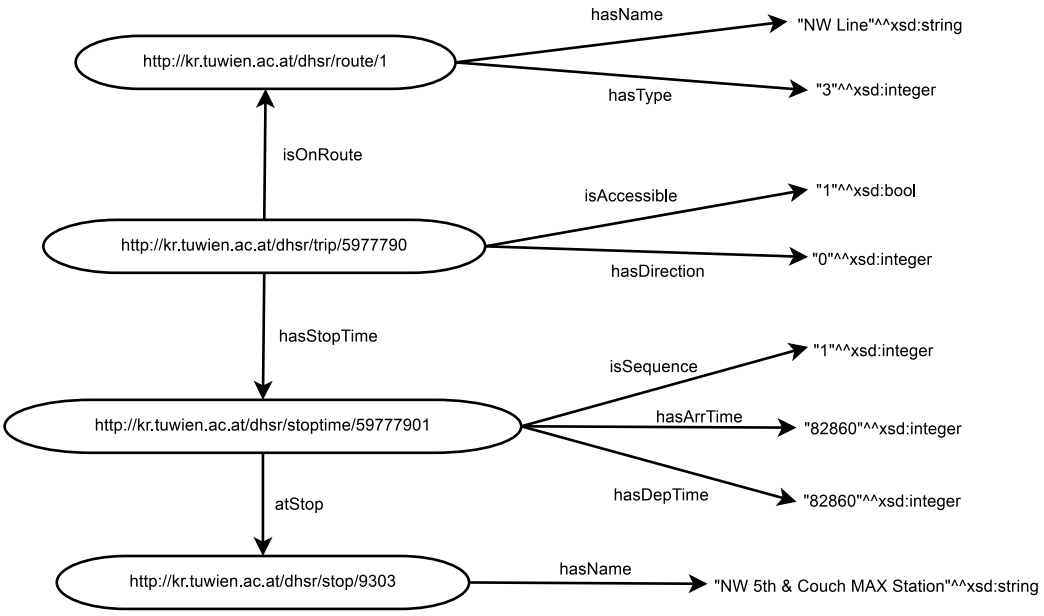

Figure 7: RDF data model of GTFS

by an ascending sequence number. A stop_time entity has an arrival and a departure time.

This GTFS data model and its entities can easily be transformed into a triple based RDF data model. Fig. 7 shows the result of this transformation.

5.1.2 Streaming Data

There are 3 different types of real-time data offered by GTFS-realtime. For our needs, *TripUpdates* and *VehiclePositions* are useful. A *Trip Update* represents a change to the timetable and consists of possibly multiple *StopTimeUpdates*, which each contain either delays or new arrival times for single stop_time entities of a trip. *Vehicle Positions* serve data concerning the position of a vehicle. A trip descriptor tells which trip a vehicle is serving. *VehicleStopStatus* gives the position of the vehicle relative to a stop and can be *Incoming at*, *Stopped at* or *In transit to*. GTFS-realtime feeds can typically be accessed by HTTP protocol and data is provided in a format based on Protocol Buffers [12], a serialization mechanism for structured data emphasizing performance. Fig. 8 shows how entities of those feeds look like. When capturing these data streams we extract the stop_time id using the trip_id and stop_sequence and save it together with the delay or timestamp of the arrival at the stop.

```
entity {
  id: "2913"
  vehicle {
    trip {
      trip_id: "5391900"
      route_id: "93"
    }
    position {
      latitude: 45.3898277283
      longitude: -122.804138184
      bearing: 51.0
    }
    current_stop_sequence: 14
    current_status: STOPPED_AT
    timestamp: 1429474116
    stop_id: "4316"
    vehicle {
      id: "2913"
      label: "93 To Tigard TC"
    }
  }
}
```

(a) Vehicle Position

```
entity {
  id: "5390417"
  trip_update {
    trip {
      trip_id: "5390417"
    }
    stop_time_update {
      stop_sequence: 1
      arrival {
        delay: 32
      }
    }
    stop_time_update {
      stop_sequence: 35
      arrival {
        delay: 32
      }
    }
  }
}
```

(b) Trip Update

Figure 8: Examples of entities from a GTFS-realtime feed

5.2 Implementation

In this section we describe how we realize the modules of our architecture for the GTFS domain use case. We decided to implement the components as small scripts, communicating using files and TCP connections. This highly transparent working model allows to exchange single components very easily and allows to examine intermediate results. By utilizing powerful and well documented libraries for handling GTFS and RDF data a low entry barrier is maintained. Following scripts are provided:
First there are the `gtfs-converter.py` and the `gtfs-capture.py` scripts, which implement the converter and capture modules respectively. These scripts are the only ones specific to the GTFS use case. All the other scripts and programs are generic and don't make any assumptions about the data domain. Then we cover the `simple_feeder.py`, `replay_feeder.py` and the `triple_to_asp.py` scripts which provide implementations of the feeder module. The `output_formatter.py` script covers the output_formatter module. Finally we describe the programs written to provide access to the different stream processing engines. All our code is available at github [14].

Converter The `gtfs-converter.py` script is the implementation of the converter module. It reads an unzipped GTFS data set and converts it to our RDF data model (Fig. 7). The output can be chosen from several RDF serializations or one can choose to output ASP facts. Fig. 9 shows how to use the script. The `--limit` argument can be used to limit the number of trips to convert, in order to obtain a smaller subset of the data set. This can be helpful when running evaluations, because a big data set might significantly hinder performance of some engines. However, it may eliminate data referenced by captured streaming data. The `gtfs-converter.py` script utilizes the transitfeed [15] and RDFLib [13] libraries, which help with reading GTFS data sets and handling RDF data respectively.

- Input: GTFS data set as a number of CSV files
- Output: File containing RDF data set as RDF/XML, Turtle or N-Triples serialization or as ASP facts

Capture The `gtfs-capture.py` script implements the capture module. It captures GTFS-realtime streams. The URLs of the streams which are to be captured have to be configured in the `streams.ini` file found in the directory of the script. The script can capture a trip update stream or a vehicle stream or both. If no `--limit` is given it continues to capture data from the streams until it is stopped. Fig. 10 shows how to call the script. Because incremental fetching of data is currently unsupported in the GTFS-realtime standard, polling a stream will produce a lot of duplicate

```
usage: gtfs-converter.py [-h] [-f {asp,xml,turtle,nt}] [-l LIMIT]
                         gtfs_path output_file

Convert GTFS data set to RDF or ASP.

positional arguments:
  gtfs_path             path to the GTFS data set
  output_file           output file

optional arguments:
  -h, --help            show this help message and exit
  -f {asp,xml,turtle,nt}, --format {asp,xml,turtle,nt}
  -l LIMIT, --limit LIMIT
                        maximum number of trips to convert
```

Figure 9: Usage of gtfs-converter.py script

data. This duplicate data is eliminated at the end of the `gtfs-capture.py` script.

- Input: GTFS-realtime streams in Protocol Buffers format accessed via HTTP
- Output: File containing captured stream triples with timestamps

Feeder We provide several implementations of the feeder module covering different purposes.

simple_feeder.py Our first implementation of the feeder module is the `simple_feeder.py` script. It can be configured with a fixed delay which it will use between each triple it sends. This provides a constant stream of data useful for benchmarking. Fig. 11 shows how to use it. The script simply outputs data to `stdout`, allowing flexible usage. For example, one can pipe it into `netcat` [11] to send it's data to a TCP socket. It can also be piped into the `kafka-console-producer.sh` script provided with an Apache Kafka installation to interface with Apache Spark. In order to not hinder this flexibility, the output data is serialized using JSON [9], a standard format for information exchange on the web.

- Input: Capture file containing stream triples with timestamps
- Output: Stream triples as JSON to `stdout`

```
usage: gtfs-capture.py [-h] [-t {t,v,b}] [-l LIMIT] output_file

Capture GTFS-realtime stream to file.

positional arguments:
  output_file           output file

optional arguments:
  -h, --help            show this help message and exit
  -t {t,v,b}, --type {t,v,b}
                        capture (t)rip updates, (v)ehicles or (b)oth
  -l LIMIT, --limit LIMIT
                        maximum number of triples to capture
```

Figure 10: Usage of gtfs-capture.py script

```
usage: simple_feeder.py [-h] [-d DELAY] capture_file

Stream triples read from capture_file to stdout

positional arguments:
  capture_file

optional arguments:
  -h, --help            show this help message and exit
  -d DELAY, --delay DELAY
```

Figure 11: Usage of simple_feeder.py script

```
usage: replay_feeder.py [-h] capture_file

Stream triples read from capture_file to stdout using the timing
provided in the capture file

positional arguments:
  capture_file

optional arguments:
  -h, --help    show this help message and exit
```

Figure 12: Usage of replay_feeder.py script

replay_feeder.py The `replay_feeder.py` script is another implementation of the feeder module. It uses the timing information captured with the streaming data to replay the stream exactly like it was captured. This provides a stream that represents real world conditions, opposed to the artificial timing of the `simple_feeder.py` script. It's usage is described in Fig. 12. Just like with the `simple_feeder.py` script, data is output to `stdout` serialized as JSON.

- Input: Capture file containing stream triples with timestamps
- Output: Stream triples as JSON to `stdout`

triple_to_asp.py The `triple_to_asp.py` script takes the place of the feeder module when evaluating queries with an ASP solver. We simulate a stream and window operations on this stream by splitting the captured streaming data into files, each representing a window. Our script can simulate the streaming behavior of the `replay_feeder.py` and the `simple_feeder.py` scripts or it can simply create triple based windows. Additionally this script also converts the engine agnostic format of the captured streaming to ASP facts. Fig. 13 shows how to call the script for each of the available operating modes.

- Input: Capture file containing stream triples with timestamps
- Output: Stream triples as ASP facts in files representing windows

Output Formatter Our implementation of the output_formatter module is the `output_formatter.py` script. It only takes the path to a directory as argument, as can be seen in Fig. 14. In this directory there may be several

```
usage: triple_to_asp.py [-h] {replay,simple,triples} ...

Convert captured triples to ASP facts, split into multiple files, each
corresponding to a tumbling window.

positional arguments:
  {replay,simple,triples}
                        operation mode
    replay              mimic the replay_feeder script
    simple              mimic the simple_feeder script
    triples             fixed number of triples per window

usage: triple_to_asp.py replay [-h] window input_file output_directory

positional arguments:
  window            window size in seconds
  input_file        input file containing captured triples
  output_directory  directory where the output files containing facts
                    will be created

usage: triple_to_asp.py simple [-h] window delay input_file output_directory

positional arguments:
  window            window size in seconds
  delay             delay between triples
  input_file        input file containing captured triples
  output_directory  directory where the output files containing facts
                    will be created

usage: triple_to_asp.py triples [-h] size input_file output_directory

positional arguments:
  size              number of triples per window
  input_file        input file containing captured triples
  output_directory  directory where the output files containing facts
                    will be created
```

Figure 13: Usage of triple_to_asp.py script

```
usage: output_formatter.py [-h] [-s] path

Converts results to a single comparable format.

positional arguments:
  path        path to the results

optional arguments:
  -h, --help  show this help message and exit
  -s, --sort  sort results inside a window
```

Figure 14: Usage of output_formatter.py script

subdirectories containing results of evaluations. One example for the structure of such a directory can be seen in Fig. 15. The script will then create a subdirectory called *formatted* and duplicate this directory structure in there, containing result files using a canonical format. This means replacing URL style names used in the RDF processing engines, replacing facts obtained from an ASP reasoner and joining the multiple output files generated by Apache Spark.

- Input: File or files containing results in engine specific format
- Output: File containing results in a canonical, engine agnostic format

Engines Descriptions of the wrappers or *shims* providing access to the engines follow in the next paragraphs.

csparql_shim The `csparql_shim` program encapsulates and provides an interface to version 0.9.5.1 of the C-SPARQL [18] engine, available at C-SPARQL's maven repository [3]. The program reads a query in the extended SPARQL grammar defined by C-SPARQL from a file. It can also read an optional static data set from a file formatted in an RDF serialization. Streaming data can be sent to the engine by establishing a TCP connection to it. This can easily be done by combining one of our feeder scripts with netcat [11]. Any output generated by evaluation of the given query will be written to a specified output file. Fig. 16 shows how to call the program.

- Input: File containing a query, stream triples over TCP, optionally a file containing a static data set in an RDF serialization
- Output: File containing results

```
results
|-- asp
|   `-- query01.txt
|-- cqels
|   `-- cqels_query01.txt
|-- csparql
|   `-- csparql_query01.txt
`-- spark
    |-- Query01-1446839885000
    |   |-- _SUCCESS
    |   `-- part-00000
    |-- Query01-1446839886000
    |   |-- _SUCCESS
    |   |-- part-00000
    |   |-- part-00001
    |   |-- part-00002
    |   |-- part-00003
    |   `-- part-00004
    |-- Query01-1446839887000
    |   |-- _SUCCESS
    |   |-- part-00000
    |   |-- part-00001
    |   `-- part-00002
    `-- Query01-1446839898000
        |-- _SUCCESS
        `-- part-00000
```

Figure 15: Example of results directory structure

```
usage: java -jar CsparqlShim.java port queryfile outputfile [static_dataset]

Provides a standardized interface to the C-SPARQL engine.

propositional arguments:
  port                  port for listening for streaming data
  queryfile             file containing a query
  outputfile            output file

optional arguments:
  static_dataset        file containing a static data set
```

Figure 16: Usage of csparql_shim program

```
usage: java -jar CqelsShim.jar cqels_home port queryfile
                               outputfile [static_dataset]

Provides a standardized interface to the CQELS engine.

positional arguments:
  cqels_home            working directory used by CQELS engine
  port                  port for listening for streaming data
  queryfile             file containing a query
  outputfile            output file

optional arguments:
  static_dataset        file containing a static data set
```

Figure 17: Usage of cqels_shim program

cqels_shim The `cqels_shim` program encapsulates and provides an interface to version 1.0.1 of the CQELS [22] engine, available at the projects Google Code page [4]. The program operates in a similar way as the `csparql_shim` program, reading a query and optionally static data from files, listening for streaming data on a TCP port and writing output to a specified output file. Fig. 17 shows how to call the program.

- Input: File containing a query, stream triples over TCP, optionally a file containing a static data set in an RDF serialization
- Output: File containing results

Spark Processor This program implements a simple streaming data processing engine by leveraging Apache Spark's [2] stream processing API. Since Spark doesn't offer any intermediate language for writing queries, they have to be written as Scala functions in classes. They will be compiled when building the project and can then be loaded during runtime using reflection. Using language features of Scala and Spark's API makes it easy to work on data provided in a CSV format, so the files from a GTFS data set can be directly read into the Processor without any conversion.
Spark is a powerful cluster-computing framework. In order to not impair the capabilities to run on a cluster, we decided to not use TCP sockets for listening to streaming data. Instead we use Apache Kafka [1], a message broker system widely used in the area of cluster-computing. Fig. 18 shows which arguments the program takes. Because of the distributed computing architecture of Spark, output can't be generated in a single file but rather one output file per worker is generated in an output directory.

```
usage: Processor zk_quorum query output_dir [static_dataset]

propositional arguments:
  zk_quorum             Zookeeper quorum, where to listen for
                        streaming data using Kafka
  query                 class name of query to run
  output_dir            directory where to output data

optional arguments:
  static_dataset        file containing a static data set
```

Figure 18: Usage of Spark Processor program

- Input: Scala class containing a query, stream triples over Kafka, optionally a CSV file containing a static data set
- Output: One directory per window containing multiple files containing results

ASP (clingo) We use the ASP solver clingo to provide an oracle that tells us what results to expect for a query. In order to do this we simulate a tumbling window on the captured streaming data using the `triple_to_asp.py` script. Then we call clingo on the data from a single window, the query and optional static data to obtain the expected result for this window. This execution happens off-line, meaning we disregard the time needed to calculate the result of a window.

5.3 Execution

In this section we illustrate how to evaluate queries on the GTFS domain using C-SPARQL, CQELS, Spark and clingo respectively. We particularly focus on which of the previously described scripts and programs are used and how they interact.

RDF stream processors Fig. 19 shows, which scripts are run when evaluating CQELS or C-SPARQL. First of all, the `gtfs-converter.py` script is run to convert a GTFS data set to RDF format. Then GTFS-realtime streams are captured using the `gtfs-capture.py` script. After both static and streaming data set are ready, the evaluation can be run. The `cqels_shim` or `csparql_shim` program is started and given a file containing the converted static data set and a file containing a query. Then either the `simple_feeder.py` or the `replay_feeder.py` script is started with the file containing the captured streaming data and its output is piped into netcat in order to stream

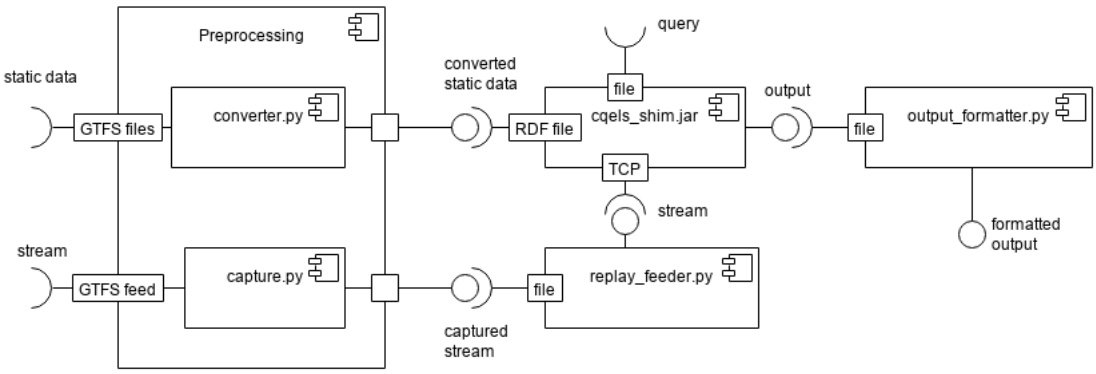

Figure 19: Component diagram for GTFS and CQELS

the data over TCP to the engine. When the evaluation is complete, the output file generated by the engine is run through the `output_formatter.py` script, to allow comparison with the output from other engines.

Spark Processor Fig. 20 shows how the scripts interact when the Spark Processor is used for evaluating queries. The main difference to the case of the RDF stream processors is that the `gtfs-converter.py` script doesn't have to be run, because the Spark Processor can easily work directly on the GTFS files in CSV format. Another difference is that the communication between the feeder and the engine happens using Kafka instead of a TCP connection. Therefore the output of the feeder has to be piped into the `kafka-console-producer.sh` script instead of netcat. Also, the query is loaded from a Scala class using Reflection instead of reading directly from a file, because Spark doesn't provide a parseable language for writing queries.

ASP (clingo) In Fig. 21 the structure of an evaluation using clingo can be seen. The difference here to the case of the RDF stream processors is that clingo expects its input files to contain logic programs. Therefore the `gtfs-converter.py` script has to be called with the `-f asp` parameter in order to generate ASP facts. Additionally the feeder is replaced by the `triple_to_asp.py` script, which generates multiple capture files, representing a single window each. Then clingo has to be called with the file containing the static data set and the file containing the query for each of the capture files. The `output_formatter.py` script then converts the facts representing the results to a format better suited for comparison with the other engines.

5.4 Evaluation

In this section a few evaluations in the GTFS domain are introduced. First is a coarse functionality test. Then some important aspects of correctness

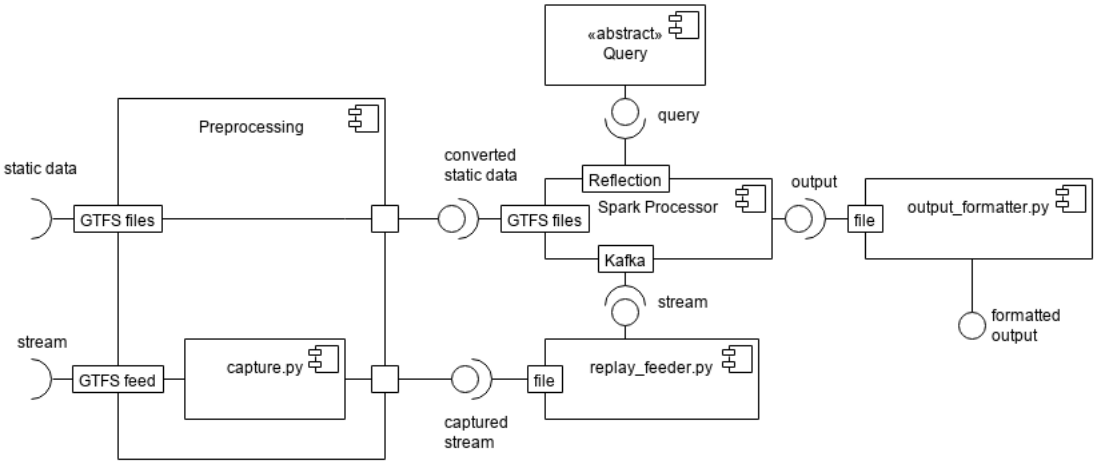

Figure 20: Component diagram for GTFS and the Spark Processor

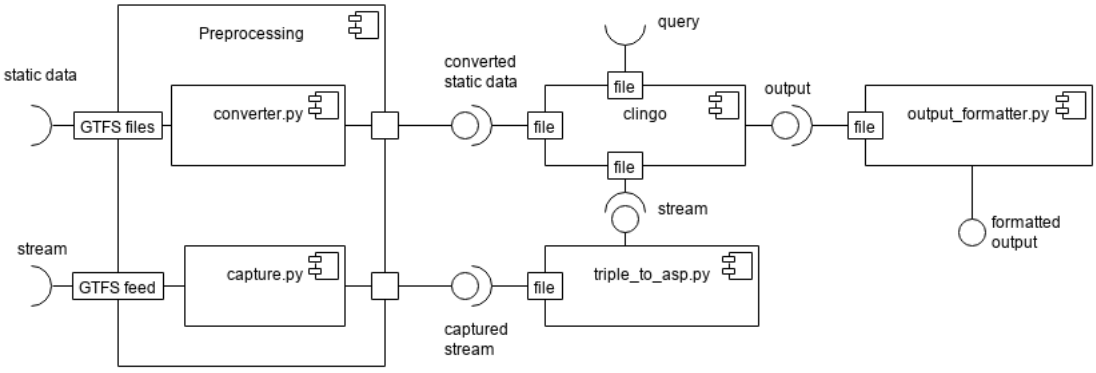

Figure 21: Component diagram for GTFS and clingo

tests are discussed. Finally, an example of an evaluation is described step by step.

5.4.1 Functionality

Our first evaluation was a basic test determining which features of SPARQL 1.1 are supported by C-SPARQL and CQELS, similar to SRBench [24]. We also tested which of those features we could replicate with Spark and clingo. All the queries used in this evaluation are available in a github repository [14]. A description of the queries follows.

Queries

01 Simply output all *hasArrived* triples.

02 Use `FILTER` to output only *hasDelay* triples with a delay greater than a certain value.

03 Use `UNION` to output both *hasDelay* and *hasArrived* triples.

04 Use `OPTIONAL` to output *hasDelay* and optionally a *hasArrived* triple of the same stop.

05 Calculate a value (delay in minutes) directly in `SELECT` clause.

06 Calculate a value (delay in minutes) using a `BIND` clause.

07 Aggregate function `COUNT`.

08 Aggregate function `COUNT DISTINCT`.

09 Aggregate function `MAX`.

10 `ORDER BY`

11 Simple join combining streaming and static data.

12 Simple join combining streaming and static data, using `OPTIONAL` clause.

Table 1 shows the results of this evaluation. As can be seen, C-SPARQL supports all language features that were tested. This can be attributed to the use of a SPARQL engine as a black box inside of C-SPARQL. CQELS is still under heavy development and doesn't support `UNION`, `OPTIONAL`, calculating values in the `SELECT` clause, `BIND` and `MAX`. `ORDER BY` can not be implemented in Spark. Data within a micro batch can be sorted, but sorting across multiple batches conflicts with Spark's concept of operation. Because `clingo` produces an answer set where the order of results is arbitrary it doesn't support any sorting of results either. However, by creating a linear order explicitly, sorting of results can be simulated.

	Queries											
	01	02	03	04	05	06	07	08	09	10	11	12
C-SPARQL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CQELS	✓	✓	-	-	-	-	✓	✓	-	✓	✓	-
Spark	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓
clingo	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Results of functionality test (✓query produced output, - query resulted in error or didn't return anything)

5.4.2 Correctness

When comparing different engines, the underlying semantics cannot be ignored. C-SPARQL, CQELS and Spark generally output a multiset as result. Only in queries with an `ORDER BY` clause, the order of the output isn't arbitrary. As the name suggests, Answer Set Programming solvers such as clingo always output a set as result. Because of this discrepancy one might argue that ASP isn't suitable to serve as an oracle for comparing the other engines. However, in natural queries, if it is of interest, whether a result occurs multiple times, `GROUP BY` and `COUNT` clauses are used. If it is known that a result may occur multiple times, but it is not of interest how often, `DISTINCT` is used. Therefore, for regular queries, using ASP as oracle is perfectly fine. If a query has no `ORDER BY` clause, the arbitrary order of the output makes it difficult to directly compare results. The set semantics allow to sort the results of a single window when comparing. Sorting across window borders however, is not allowed.

In the following example we show how to run an evaluation comparing the output of the engines for query 11 from the previous section.

Example 1 First of all, a GTFS data set has to be obtained. We used the data set provided by Portland's TriMet transit agency, available at `http://developer.trimet.org/GTFS.shtml`. Using the `gtfs-converter.py` script this data set is converted to RDF and ASP formats:

```
$ python sr_data_generator/converter/gtfs-converter.py \
-f turtle gtfs_portland/ data_portland.ttl
$ python sr_data_generator/converter/gtfs-converter.py \
-f asp gtfs_portland/ data_portland.lp
```

Then the URLs of TriMet's GTFS-realtime feeds have to be configured in the `streams.ini` file in the directory of the `capture.py` script. Once this is done, the streams can be captured:

```
$ python sr_data_generator/capture/gtfs-capture.py \
capture_portland.triple
```

Once sufficient streaming data has been captured, the script can be stopped and evaluation can begin. First the C-SPARQL engine is run and then streaming of the captured data is started:

```
$ java -jar sr_data_generator/csparql_shim/target/CsparqlShim-0.0.1.jar \
9999 sr_data_generator/queries/csparql/query11.rq \
results/csparql/query11.txt data_portland.ttl
$ python sr_data_generator/feeder/replay_feeder.py \
capture_portland.triple | nc localhost 9999
```

After this evaluation is done, the same is done for the CQELS engine:

```
$ java -jar sr_data_generator/csparql_shim/target/cqels_shim-1.0-w-deps.jar \
~/cqels 9999 sr_data_generator/queries/cqels/query11.rq \
results/cqels/query11.txt data_portland.ttl
$ python sr_data_generator/feeder/replay_feeder.py \
capture_portland.triple | nc localhost 9999
```

And for the Spark Reasoner:

```
$ spark-submit --class "Reasoner" --master 'local[4]' \
target/scala-2.10/gtfs-reasoner-assembly.jar localhost \
Query11 results/spark/ gtfs_portland/stop_times.txt
$ python sr_data_generator/feeder/replay_feeder.py \
capture_portland.triple | kafka-console-producer.sh \
--broker-list localhost:9092 --topic gtfs
```

To provide the baseline for the comparison, windows are simulated using the `triple_to_asp.py` script and clingo is run on the data and query:

```
$ python sr_data_generator/triple_to_asp/triple_to_asp.py replay 1 \
capture_portland.triple capture_asp/
$ for f in capture_asp/*; \
do clingo 1 --outf=1 sr_data_generator/queries/asp/query11.lp \
data_portland.lp $f >> results/asp/query11.txt; done
```

Finally, the `output_formatter.py` script is run to convert the outputs to a canonical format:

```
$ python sr_data_generator/output_formatter/output_formatter.py \
-s results/
```

Now the results of the engines can easily be compared against the baseline:

```
$ diff results/formatted/asp/query11.txt \
results/formatted/csparql/query11.txt
```

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. It probably couldn't keep up with the fast stream and dropped some triples.

References

[1] Apache Kafka. https://kafka.apache.org/. [Online; accessed: 2015-11-10].

[2] Apache Spark. https://spark.apache.org/. [Online; accessed: 2015-11-10].

[3] C-SPARQL maven repository. http://streamreasoning.org/maven. [Online; accessed: 2015-11-10].

[4] CQELS Google Code page. https://code.google.com/p/cqels/. [Online; accessed: 2015-11-10].

[5] General Transit Feed Specification. https://developers.google.com/transit/. [Online; accessed: 2015-11-10].

[6] GoogleTransitDataFeed project site. https://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds. [Online; accessed: 2015-11-10].

[7] GTFS Data Exchange. http://www.gtfs-data-exchange.com/. [Online; accessed: 2015-11-10].

[8] GTFS reference. https://developers.google.com/transit/gtfs/reference. [Online; accessed: 2015-11-10].

[9] JSON Data Interchange Format. https://tools.ietf.org/html/rfc7159. [Online; accessed: 2015-11-13].

[10] LinkedSensorData. http://wiki.knoesis.org/index.php/LinkedSensorData. [Online; accessed: 2015-11-10].

[11] netcat. http://nc110.sourceforge.net/. [Online; accessed: 2015-11-10].

[12] Protocol Buffers. https://developers.google.com/protocol-buffers/. [Online; accessed: 2015-11-13].

[13] RDFLib. https://github.com/RDFLib/rdflib. [Online; accessed: 2015-11-10].

[14] sr_data_generator github repository. https://github.com/mosimos/sr_data_generator/. [Online; accessed: 2015-11-10].

[15] transitfeed library. https://github.com/google/transitfeed. [Online; accessed: 2015-11-10].

[16] W3C Recommendation of the SPARQL 1.1 Query Language. http://www.w3.org/TR/sparql11-query/. [Online; accessed: 2015-11-13].

[17] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 480–491. Morgan Kaufmann, 2004.

[18] 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.

[19] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.

[20] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.

[21] Vladimir Lifschitz. What is answer set programming? In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1594–1597. AAAI Press, 2008.

[22] Danh Le Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, pages 370–388. Springer, 2011.

[23] Danh Le Phuoc, Minh Dao-Tran, Minh-Duc Pham, Peter A. Boncz, Thomas Eiter, and Michael Fink. Linked stream data processing engines: Facts and figures. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA,*

November 11-15, 2012, Proceedings, Part II, volume 7650 of *Lecture Notes in Computer Science*, pages 300–312. Springer, 2012.

[24] Ying Zhang, Minh-Duc Pham, Óscar Corcho, and Jean-Paul Calbimonte. Srbench: A streaming RDF/SPARQL benchmark. In Philippe Cudré-Mauroux, Jeff Heflin, Evren Sirin, Tania Tudorache, Jérôme Euzenat, Manfred Hauswirth, Josiane Xavier Parreira, Jim Hendler, Guus Schreiber, Abraham Bernstein, and Eva Blomqvist, editors, *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, volume 7649 of *Lecture Notes in Computer Science*, pages 641–657. Springer, 2012.