Stream Processing: The Matrix Revolutions

Pernischova, Romana ; Ruosch, Florian ; Dell'Aglio, Daniele ; Bernstein, Abraham

Stream Processing: The Matrix Revolutions

Romana Pernischova, Florian Ruosch,
Daniele Dell'Aglio, and Abraham Bernstein

University of Zurich, Zurich, Switzerland
{pernischova,dellaglio,bernstein}@ifi.uzh.ch, florian.ruosch@uzh.ch

Abstract. The growth of data velocity sets new requirements to develop solutions able to manage big amounts of dynamic data. The setting becomes even more challenging when such data is heterogeneous in schemata or formats, such as triples, tuples, relations, or matrices. Looking at the state of the art, traditional stream processing systems only accept data in one of these formats.
Semantic technologies enable the processing of streams combining different shapes of data. This article presents a prototype that transforms SPARQL queries to Apache Flink topologies using the Apache Jena parser. With a custom data type and tailored functions, we integrate matrices in Jena and therefore, allow to mix graphs, relational, and linear algebra in an RDF graph. This provides a proof of concept that queries written for static data can easily be run on streams with the usage of the streaming engine Flink, even if they contain multiple of the aforementioned types.

Keywords: query· continuous queries· streams· RDF· SPARQL· Flink· linear algebra· relational algebra

1 Introduction

The processing of real-time information is getting more and more critical, as the number of data stream sources is rapidly increasing. Often, reactivity is an important requirement when working with this kind of data: the value of the output decreases quickly over time. The state of the art to process unbounded data reactively relies on stream processing engines which set their roots in database and middleware research.

The processing of this type of data is also relevant on the Web, where several use cases can be found in the context of Internet of Things (and the related Web of Things), as well as in social network and social media analytics. An interesting challenge that emerges from the Web setting is the data heterogeneity, as shown in the example below.

A market research company is tasked with developing a system to analyze the behavior of users of an online TV platform. In particular, they want to investigate if certain images on TV programs cause customers to change TV stations and if this behavior is similar among people who know each other. This can result in customer specific programs and tailored advertisements that would induce the user to change the TV station or to stay on. Such an analysis needs to combine data of different formats: the TV program (i.e. a stream of images and sounds), the user activities (i.e. a relational stream), and the program schedules and advertisement descriptions (i.e. a relational or graph database). When performing this kind of analysis, it is common practice to represent the TV program as a sequence of matrices, obtained by applying matrix specific functions like the Fast Fourier Transform (FFT). The FFT computes the frequencies of the different images that appear in the video, and it enables an association which can be used in in-depth analysis that includes the behavior and relationship data. The additional data has a different shape than the images. Such data is usually given through tables or graphs.

To find the results, this data needs to be combined: the stream data has to be integrated to identify images which were last seen before switching stations. The data which contains the time spend watching a specific TV program is also a stream, since it is unbounded.

To the best of our knowledge, today we lack scalable big data platforms able to manage streams of different types in a reactive and continuous way. In this paper, we make a first step in this direction by analyzing the problem of processing three different types of data streams: matrices, relations, and graphs. In other words, we want to investigate *how to build a big data platform to process streams containing matrices, tables, and graphs.*

The combination of the different types of streams requires some common data model or strategy of handling the heterogeneity while processing a query. In addition to this, such a platform should allow users to issue complex queries and enable them to exploit different types of operators depending on the underlying data. A query language is therefore needed to capture the needs of the user, including operators to express complex functions and combination of streams. This language depends on the chosen strategy for the integration of the different streams. Finally, the query has to be processed over the streams in a continuous fashion and should return a sequence of answers which are updated according to the input streams.

Our main contribution is a model to process streams of data in different formats through relational and linear algebra operators. We exploit semantic web technologies to cope with the format heterogeneity, and we adopt distributed stream processing engine models as a basis to build an execution environment. We show the feasibility of our approach through an implementation based on Apache Jena and Flink.

2 Background

Processing data in the context of the Web is challenging, since it often inherits the issues that characterize big data. It suffers from a variety of problems: data from multiple sources has different serializations, formats, and schemas. The Semantic Web has shown to be a proper solution to cope with these kinds of issues: it offers a broad set of technologies to model, exchange, and query data on the Web. RDF [8] is a model to represent data. Conceptually, it organizes data in a graph based structure, where the minimal information unit is the statement, a triple composed by a predicate (the edge), a subject and an object (the vertices). Subjects and predicates are resources, i.e. URIs denoting entities or concepts; objects can be either URIs or literals, that are strings with an optional data type, such as integer or date.

SPARQL [11] is a protocol and RDF query language, used to manipulate and retrieve linked data. It uses sets of triple patterns, called Basic Graph Patterns (BGP), to match subgraphs. The language is similar to SQL and uses keywords like SELECT and WHERE to address the underlying concepts. To create graphs and run queries, the framework Apache Jena [1] can be used.

When data is very dynamic and its processing needs to be reactive, solutions like RDF and SPARQL may not suffice. Recently, several research groups started to investigate how to adapt the Semantic Web stack to cope with velocity. In this context, it is worth mentioning the work of the W3C RDF Stream Processing (RSP) Community Group [2], which collected such efforts and led several initiatives to disseminate the results. Relevant results of this trend are *RDF streams*, as a (potentially unbounded) sequence of time-annotated RDF graphs, and *continuous extensions of SPARQL*, which enable users to define tasks, as well as queries to be evaluated over RDF streams. Windows are introduced to be able to treat the unbounded data, which enables calculations over the data inside the window. Without windowing there is no data completeness and the triggering of executions is problematic.

While the RDF Stream Processing trend introduced several notions to manage streams, only an initial effort has been dedicated to the creation of solutions to cope with the volume of data generated in the Web context. The state of the art in the processing of large amounts of streaming data relies on distributed stream processing engines (DSPE). These platforms emerged as successors of MapReduce frameworks and are developed to be deployed into clusters and to run the processing of streams of data in a distributed fashion. Users are required to design *topologies*, i.e. logical workflows of operations arranged in directed acyclic graphs, which are taken as input by the DSPE and are deployed according to the configuration settings and the hardware availability.

3 Related Work

Several studies investigated different types of data and how to combine them. With regards to the three types of data we are considering, Figure 1 shows some of the query languages we considered as foundations of this study.

Graph stream processing There is not a common definition of graph stream processing. In the survey presented by McGregor [18], the focus is on processing very large graphs: since they cannot be kept in memory, they are streamed

[1] Cf. https://jena.apache.org/
[2] Cf. https://www.w3.org/community/rsp/.

into the system, and typical graph operations are run as on-line algorithms. A different approach is the one taken by the RSP community group, which models streams where data items are composed by graphs. In this case, the processing consists of the execution of relational operators over portions of the stream (such as aggregations), event pattern matching, or deductive inference processing [9]. None of the studies mentioned above investigated the integration of streams of graphs with other types of streams.

Dealing with linear and rational algebras. SQL and SPARQL are two examples of query languages to process tuples and graph-based data through relational algebra. However, these kinds of operators can hardly be used to perform linear algebra operations over matrices, such as transposition and calculating the determinant. SciDB [22, 21] is one example of a system that bridges these two worlds. This database stores arrays rather than tuples, and tasks are defined through an SQL-like language called AQL (Array Query Language). Moreover, Andejev, He, and Risch [3] introduce their prototype that can be accessed with Matlab. It provides storage of arrays in an RDF graph and retrieval of the data and its meta-information using SciSPARQL. SciSPARQL is an extension of SPARQL that incorporates array operations within the query. The authors focus on the integration of the different format rather than on stream processing. They make the processing of large amounts of static data easier.

Another effort in such a direction is LaraDB [12], which proposes Lara that combines relational and linear algebra. It uses a new representation, called associative table, into which relations, scalars, and matrices are recast. They map operators from relational and linear algebra onto their functions and in this way are able to express combinations of those.

Looking at query languages, LARA [14] relies on abstract data types and local optimizations; however, there is no known system that would support such a language. EMMA [2] is a language for parallel data analysis: its goal is to hide the notion of parallelism behind a declarative language, which is realized using monad comprehensions, which are based on set comprehension syntax. EMMA introduces bags as the algebraic data types and enables the use of different algebras by replacing the general union representation in a binary tree.

While there is an ongoing trend in research to combine linear and relational algebra, we are not aware of studies that focus on a streaming setting.

Stream Processing Engines Research on stream processing sets its foundation in the database and the middleware communities [7]. The former proposed models and methods to process streams according to the relational model, like CQL [4], while the latter took a different perspective, developing techniques to identify relevant sequences of events in the input streams [15].

The research on this field got revitalized in the last years, as an evolution of the MapReduce paradigm, which led to the development of distributed stream processing engines (DSPE). Apache Spark Streaming [24] sits on top of the initial Spark architecture, which implements batch processing. It focuses on stateless operations and stateful windows. In contrast, Apache Storm [23] is natively a

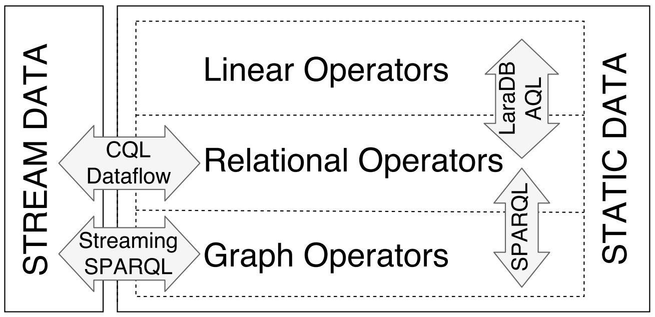


Fig. 1. Existing languages combining different algebras.

stream processing engine and supports query operations such as joins and aggregations. It provides a low-level API which allows for the use of different programming languages. Apache Flink [6] is optimized for cyclic or iterative processes. Unlike Spark, it adopts a native streaming approach and can handle data that does not fit into RAM. The Google Dataflow model [1] and its implementation in Apache Beam [3] present a different approach: they aim to act as a façade, running a Dataflow-compliant topology in a DSPE, such as Apache Spark, Flink, or Google Cloud Dataflow.

All of the above systems support windowing and typical relational algebra operators. Such platforms also offer support to linear algebra operations (through plug-ins and extensions). However, the topologies are specified through programmable APIs rather than a query language. Having such a tool would be useful to let users with limited programming skills express their tasks through a declarative language, without requiring users to code the topologies.

4 The Model

In this section, we describe the model we envision to use for processing queries over heterogeneous streams. Figure 2 shows a logical representation of the model with a highlight on the three main challenges we identified. The first one (denoted by 1 in the picture) relates to the data integration: given a set of streams containing graphs, relations, and matrices, *how can they be integrated in a common data model?* The second one captures the user's needs: *what is a suitable query language to let the user express tasks combining relational and linear algebra operators?* The third one puts the pieces together: *how to execute the queries over the input data?* In the following sections, we discuss the challenges and propose our solution.

[3] Cf. https://beam.apache.org/.

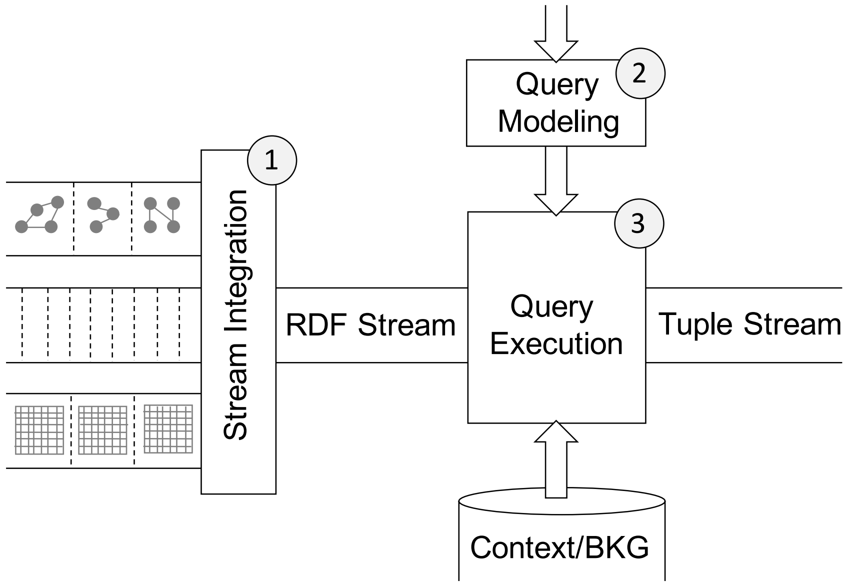


Fig. 2. Model of the proposed system for processing multiple and heterogeneous data sources.

4.1 Stream Integration

The idea of integrating data by exploiting semantic web technologies is well-known and consolidated [19]. This also holds in the streaming context, where recent studies investigated how to integrate streams of relational or graph-based data through RDF streams [5, 13, 17].

How to lift stream of matrices to RDF streams is still unexplored, and requires some considerations. Given a matrix, there are ways to convert it into a graph-based structure and consequently in RDF, e.g., each cell of the matrix can be represented by a node, annotated with its position in the matrix, its value, and properties relating it to adjacent cells. However, the representation of the matrix data has a significant impact on the query language, which may require long and complex descriptions to declare the linear algebra operations. Therefore, an option is to keep the matrix data as is, and only transform it if and when the query execution requires it. On this regard, the authors of LARA [14] point out that the transformation of a matrix to a graph is possible, but the other way around requires an ordering function. This drawback becomes relevant if users want to execute matrix-specific functions on other data formats.

To *append* matrix data to an RDF stream, we defined some properties to annotate the matrix and a custom data type to serialize its content. This allows us to add matrices to streams as literal nodes, bringing advantages to the execution of matrix-specific functions. Listing 1.1 shows an example of an RDF stream encoding matrices. The snippet uses TriG as the serialization format, and

```
:streamItem1 {
:m1 rlg:data "[3 4 8][8 7 2][1 8 2]"^^rlg:matrix ;
rlg:columns    3;
rlg:rows       3 .
} { :streamItem1 prov:generatedAt 15 .}
:streamItem2 {
:m2 rlg:data "[1 0 2][9 6 2][6 4 0]"^^rlg:matrix;
rlg:columns    3;
rlg:rows       3.
:m1 rlg:evolvesTo    :m2.
} { :streamItem1 prov:generatedAt 17 .}
```

Listing 1.1. RDF example including a matrix node

the stream is encoded according to the model proposed in [17]. It contains two stream items (represented as RDF graphs), generated at time instants 15 and 17. Each stream item contains a matrix: *data* is a data type property having literals of type *matrix* as the range; *columns* and *rows* are additional annotations. It is worth noting that the snippet is compliant with the RDF model, making it possible to process it with the usual semantic web related frameworks. Moreover, the object representing the matrix can be annotated with additional properties and can be linked with other resources.

4.2 Query Modeling

The choice of the data model has a significant impact on the design of the query language. As explained above, our data model is compliant with RDF, and carries additional information to account for the streaming nature of the data and the presence of matrices. It follows that SPARQL is the best starting point to design the query language. SPARQL is the W3C recommended query language for RDF with operators to manipulate RDF graphs based on relational algebra, similar to how SQL works on relations.

We need to accommodate matrix-specific functions.Having matrices as nodes makes the retrieval easy because we can refer to them by exploiting variables and accessing their *data* value. When looking at use cases, we are not interested in representing the same data in multiple formats for the sake of achieving high velocity in computation, but enabling the combination of data. With this thinking, we decided on adding the matrix-specific operators to the query language as SPARQL functions. This solution does not lead to a custom version of SPARQL since it is the recommended practice for this type of extensions [11]. An example query is shown in Listing 1.2, where the contents of matrix resources are retrieved (Lines 7–10), their inverses computed (Lines 11–12), added (Line 13) and emitted (Line 3).

Additionally, our query language needs a way to manage streams. Several studies proposed extensions to SPARQL [9], with recent ongoing efforts to unify

```
1  REGISTER STREAM :outStr AS
2  CONSTRUCT RSTREAM {
3        ?m1 :hasInverse [ ?m2 ?addInverse ] .
4  }
5  FROM NAMED WINDOW :win ON :inStr [RANGE 1 STEP 1]
6  WHERE {
7        ?m1 rdf:type rlg:Matrix .
8        ?m2 rdf:type rlg:Matrix .
9        ?m1 rlg:data ?data1 .
10       ?m2 rlg:data ?data2 .
11       BIND (afn:inverse(?data1) AS ?inverse1) .
12       BIND (afn:inverse(?data2) AS ?inverse2) .
13       BIND (afn:add(?inverse1, ?inverse2) AS ?addInverse) .
14 }
```

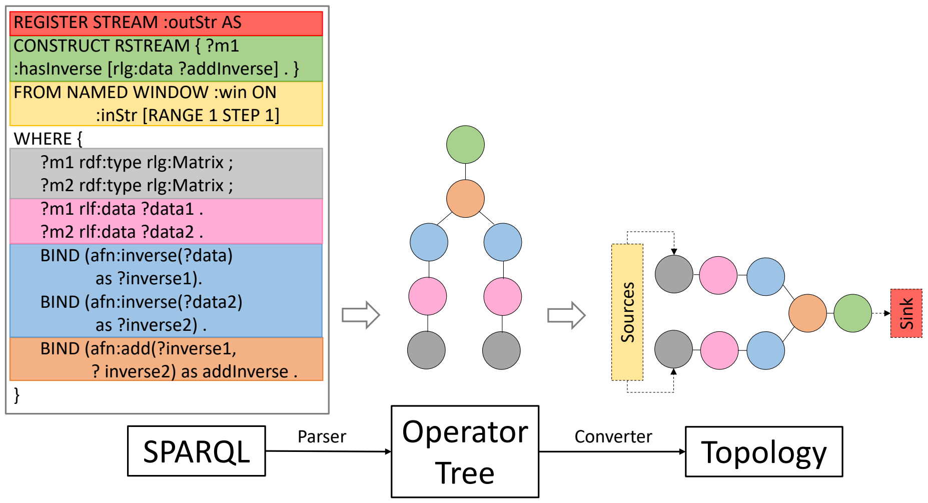
Listing 1.2. Query that computes the inverse matrices (prefixes are omitted for brevity).

them in a common and shared language. The introduction of windows and streams cannot be managed by preserving the original semantics of SPARQL entirely. In particular, the continuous evaluation requires an extension to the original SPARQL semantics: the notion of evaluation time instant needs to be included in the operational semantics to describe when and on which portion of the stream the query should be evaluated [10]. In the example in Listing 1.2, we are adopting the syntax proposed by the W3C RSP community group. An output stream :outStr is declared (Line 1) and its items are defined as graphs containing the matrices and their inverse(Lines 2–4). The window on Line 5 is declared over a stream :inStr as a tumbling window of one stream item, i.e. the query processes one stream item at a time.

4.3 Query Execution

The last step of our model consists in creating a DSPE topology that puts together the data and the query described above. Given a (continuous) SPARQL query, a way to generate a topology is shown in Figure 3. First, a parser creates a logical query plan from the string of the SPARQL query. As usual, the logical plan can be modified and optimized. Being a SPARQL query, the leaves of the tree correspond to the Basic Graph Patterns, which are defined in the WHERE clause. Those operators generate solution mappings, which are further processed by the other operators.

To generate the topology, we exploit the logical plan, as highlighted in Figure 3. In the topology, the BGP operators are on the left, which are fed with portions of the stream selected by the windows. Such BGP operators process the data and push the outputs to the correct operators, which continue the processing, sending the data towards the sinks. A converter traverses the logical query plan and creates a task in the topology for each operator. In this way, it is easier to track what happens during the execution of the query. Moreover, the decision to optimize the logical query plan allows us to exploit well-known techniques from database research. The main drawback is the fact that our converter may not find the best possible topology (regarding time performance). The converter always creates tree-shaped topologies, and it cannot generate other types of DAG.



```
REGISTER STREAM :outStr AS
CONSTRUCT RSTREAM { ?m1
:hasInverse [rlg:data ?addInverse] . }
FROM NAMED WINDOW :win ON
          :inStr [RANGE 1 STEP 1]
WHERE {
   ?m1 rdf:type rlg:Matrix ;
   ?m2 rdf:type rlg:Matrix ;
   ?m1 rlf:data ?data1 .
   ?m2 rlf:data ?data2 .
   BIND (afn:inverse(?data)
          as ?inverse1).
   BIND (afn:inverse(?data2)
          as ?inverse2) .
   BIND (afn:add(?inverse1,
          ? inverse2) as addInverse .
}
```

Fig. 3. Processing a query in a stream setting.

5 Implementation

To verify the feasibility of our model, we built a proof of concept. We started from some existing frameworks: as a DSPE, we opted for Apache Flink [6]; we used Apache Jena [4] to manage the SPARQL query; and we used JAMA [16] as a library providing matrix-related functions. In the following, we highlight some parts of our experience.

5.1 Query language

We used Apache Jena since it already provides tools to manage SPARQL, such as a parser and implementations of its operators. Moreover, Jena offers a well-documented API to extend its functions.

As explained in Section 4.2, our data model manages matrices through individuals of a Matrix class, a new literal data type that serializes its content.

[4] Cf. https://jena.apache.org/.

Whenever a literal is specified as a matrix, the string is parsed into a matrix data structure. Functions, that are specific for matrices, can be executed and the result can then be returned to the query. We implemented such functions according to the SPARQL specification, listed in Table 1. We exploited the JAMA Library from MathWorks and NIST [16] for the matrix data structures as well as for the functions manipulating the matrices.

Function	Returns
plus	result of an addition
minus	result of a subtraction
times	result of an element-wise multiplication
times	result of a multiplication by a scalar
times	result of a matrix-matrix multiplication
divide	result of an element-wise division
between	partial matrix specified by indices
join	result of joining two matrices with an operator
merge	result of merging two matrices
transpose	result of a transposition
rank	calculated rank of the matrix
determinant	calculated determinant of the matrix
inverse	inverse of the input matrix
condition	ratio of largest to smallest singular value
trace	result of the sum of the diagonal

Table 1. Functions added to SPARQL using the Library JAMA [16]

In our current implementation, the query language does not support the SPARQL extensions for streams, which is on the schedule for our future work. At the moment, such information is provided as a set of parameters. It is worth noting that this is not a limitation, since there are several prototypes that are already showing the feasibility of these features [9].

5.2 Topology Creation and Execution

We decided to use Apache Flink as the basis for the execution environment, since it offers a flexible and well-documented API. However, our approach can be ported to other DSPEs, since the notion of topology is shared among them.

When defining a Flink topology, it is necessary to declare the type of data that tasks exchange. Flink offers a set of native data types, among which Tuple is the most prominent. It is a list of values, indexed by their position number. We use Tuple for most of the data exchanges between nodes.

Given a query (partially defined through SPARQL, partially defined through extra parameters), the conversion process derives a topology. For each SPARQL operator, the process creates a task. At the moment the projection, FILTER, BIND, LIMIT, and BGP operators are supported. Furthermore, our prototype supports several window operators (since they are natively supported by Flink), and the matrix-related operations in Table 1. Besides, the process extracts the variable names, which are stored in a dedicated data structure. Tasks use this structure to manage the solution mappings as Tuple objects, inferring the position of the variable content during the query execution.

Streams among tasks exchange Tuple objects; the only exceptions are the tasks implementing BGP operators. The input of a BGP operator is a finite sequence of stream items, expressed as a set of RDF graphs. They are merged into a new RDF graph, which represents the window content, and the BGP is evaluated over it. The resulting solution mappings are converted into Tuple objects and are sent to the other tasks of the topology.

The conversion process returns a snippet of code with the topology description. This code can be fed into Flink, which instantiates the topology and executes it over the input streams. The code of the project can be found at https://gitlab.ifi.uzh.ch/DDIS-Public/ralagra/.

5.3 Limitations

While our prototype shows the feasibility of our model, it has several limitations. The current implementation does not carry the system integration component, i.e. the system expects to get as input one RDF stream compliant with the data model described in Section 4.1. Our system is not able to receive multiple streams and therefore, can not combine them on the fly. This is subject to future work. As explained above, several studies show the feasibility of this component, and we are going to implement it for the next version.

Moreover, we aim at automating the submission of the topology to Flink. When the conversion process creates the topology from the input query, the code snippet should automatically be injected into Flink. Techniques like Java reflection[5] or template engines may help in tackling this problem.

We are also working to extend our system to other SPARQL operators. At the moment, it supports the most common SPARQL features, but it is important to extend the coverage to a wider set of operators.

Finally, serializing matrices as plain strings is not the best solution in terms of space and time to process them. In future works, we plan to explore other serialization formats for matrices (and RDF streams carrying them), such as Protocol Buffer and Apache Thrift.

6 Conclusions & Future Work

In this study, we proposed a model to handle data streams carrying different types of data – relations, graphs, and matrices. We defined a data model by exploiting RDF, where streams are modeled as sequences of time-annotated RDF graphs and matrices are represented as literals. We also described a query language to manage such streams and to perform relational and linear algebra

[5] Cf. https://docs.oracle.com/javase/tutorial/reflect/.

operations over their items. We developed a proof of concept implementing the most unique parts of the model.

Over the course of the next months, we will work to consolidate the prototype and to add the other parts, to have a full RDF stream processing engine. We also aim at performing an extensive evaluation of the system. We are interested in studying the performance, the overhead introduced by our extensions and to compare our system with other prototypes developed so far. It will also be important to study more in depth to which extent our query language can support the modeling of users needs and tasks.

Finally, our prototype is setting the basis to study the problem of distribution. So far, only a few studies targeted the problem of distributed RDF stream processing engines, such as Strider [20]. The main difference in our setting is the presence of matrices and operators over them, which requires different distribution strategies.

Acknowledgments We thank the Swiss National Science Foundation (SNF) for partial support of this work under contract number #407550_167177.

References

1. Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R.J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., et al.: The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. Proceedings of the VLDB Endowment **8**(12), 1792–1803 (2015)
2. Alexandrov, A., Kunft, A., Katsifodimos, A., Schüler, F., Thamsen, L., Kao, O., Herb, T., Markl, V.: Implicit parallelism through deep language embedding. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. pp. 47–61. ACM (2015)
3. Andrejev, A., He, X., Risch, T.: Scientific data as RDF with arrays: Tight integration of scisparql queries into MATLAB. In: International Semantic Web Conference (Posters & Demos). CEUR Workshop Proceedings, vol. 1272, pp. 221–224. CEUR-WS.org (2014)
4. Arasu, A., Babu, S., Widom, J.: CQL: A Language for Continuous Queries over Streams and Relations. In: DBPL. Lecture Notes in Computer Science, vol. 2921, pp. 1–19. Springer (2003)
5. Calbimonte, J.P., Jeung, H., Corcho, ., Aberer, K.: Enabling Query Technologies for the Semantic Sensor Web. Int. J. Semantic Web Inf. Syst. **8**(1), 43–63 (2012)
6. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **36**(4) (2015)
7. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. **44**(3), 15:1–15:62 (2012)
8. Cyganiak, R., Wood, D., Lanthaler, M.: RDF 1.1 Concepts and Abstract Syntax. W3c Recommendation, W3C (2014), https://www.w3.org/TR/rdf11-concepts/
9. Dell' Aglio, D., Della Valle, E., van Harmelen, F., Bernstein, A.: Stream reasoning: A survey and outlook. Data Science **1**(1–2), 59–83 (2017)

10. Dell'Aglio, D., Della Valle, E., Calbimonte, J.P., Corcho, .: RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems. Int. J. Semantic Web Inf. Syst. **10**(4), 17–44 (2014)
11. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3c Recommendation, W3C (2013), https://www.w3.org/TR/sparql11-query/
12. Hutchison, D., Howe, B., Suciu, D.: Laradb: A minimalist kernel for linear and relational algebra computation. BeyondMR@SIGMOD pp. 2:1–2:10 (2017)
13. Kharlamov, E., Hovland, D., Jimnez-Ruiz, E., Lanti, D., Lie, H., Pinkel, C., Rezk, M., eveland, M.G.S., Thorstensen, E., Xiao, G., Zheleznyakov, D., Horrocks, I.: Ontology Based Access to Exploration Data at Statoil. In: International Semantic Web Conference (2). Lecture Notes in Computer Science, vol. 9367, pp. 93–112. Springer (2015)
14. Kunft, A., Alexandrov, A., Katsifodimos, A., Markl, V.: Bridging the gap: towards optimization across linear and relational algebra. In: Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond. p. 1. ACM (2016)
15. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. In: RuleML. Lecture Notes in Computer Science, vol. 5321, p. 3. Springer (2008)
16. MathWorks, T., NIST: Jama: Java matrix package. http://math.nist.gov/javanumerics/jama/ (2012), accessed: 2017-12-04
17. Mauri, A., Calbimonte, J.P., Dell'Aglio, D., Balduini, M., Brambilla, M., Della Valle, E., Aberer, K.: TripleWave: Spreading RDF Streams on the Web. In: International Semantic Web Conference (2). Lecture Notes in Computer Science, vol. 9982, pp. 140–149. Springer (2016)
18. McGregor, A.: Graph stream algorithms: a survey. ACM SIGMOD Record **43**(1), 9–20 (2014)
19. Noy, N.F.: Semantic Integration: A Survey Of Ontology-Based Approaches. SIGMOD Record **33**(4), 65–70 (2004)
20. Ren, X., Cur, O.: Strider: A Hybrid Adaptive Distributed RDF Stream Processing Engine. In: International Semantic Web Conference (1). Lecture Notes in Computer Science, vol. 10587, pp. 559–576. Springer (2017)
21. Rogers, J., Simakov, R., Soroush, E., Velikhov, P., Balazinska, M., DeWitt, D., Heath, B., Maier, D., Madden, S., Patel, J., et al.: Overview of scidb. In: 2010 International Conference on Management of Data, SIGMOD'10 (2010)
22. Stonebraker, M., Brown, P., Zhang, D., Becla, J.: Scidb: A database management system for applications with complex analytics. Computing in Science & Engineering **15**(3), 54–62 (2013)
23. Toshniwal, A., Taneja, S., Shukla, A., Ramasamy, K., Patel, J.M., Kulkarni, S., Jackson, J., Gade, K., Fu, M., Donham, J., et al.: Storm@ twitter. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data. pp. 147–156. ACM (2014)
24. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., et al.: Apache spark: A unified engine for big data processing. Communications of the ACM **59**(11), 56–65 (2016)