SPARQL2FLINK: Evaluation of SPARQL queries on Apache Flink

Oscar Ceballos [a], Carlos Ramirez [b], María-Constanza Pabón [b], Andres M. Castillo [a,*] and Oscar Corcho [c]

[a] *Escuela de Ingeniería de Sistemas y Computación, Universidad del Valle, Cali, Colombia*
E-mails: oscar.ceballos@correounivalle.edu.co, andres-m.castillo@correounivalle.edu.co
[b] *Departamento de Electrónica y Ciencias de la Computación, Pontificia Universidad Javeriana Cali, Cali, Colombia*
E-mails: carlosalbertoramirez@javerianacali.edu.co, mpabon@javerianacali.edu.co
[c] *Ontology Engineering Group, Universidad Politécnica de Madrid, Madrid, Spain*
E-mail: ocorcho@fi.upm.es

Abstract. Increasingly larger RDF datasets are being made available on the Web of Data, either as Linked Data, via SPARQL endpoints or both. Existing SPARQL query engines and triple stores are continuously improving to handle larger datasets. However, there is an opportunity to explore the use of Big Data technologies for SPARQL query evaluation. Several approaches have been developed in this context, proposing the storage and querying of RDF data in a distributed fashion, mainly using the MapReduce Programming Model and Hadoop-based Ecosystems. New trends in Big Data Technologies have also emerged (e.g., Apache Spark, Apache Flink); they use distributed in-memory processing and promise to deliver higher performance data processing. In this paper we present an approach for transforming a given SPARQL query into an Apache Flink program for querying massive static RDF data. An implementation of this approach is presented and a preliminary evaluation with an Apache Flink cluster is documented. This a first step towards this main goal, but efforts to ensure optimization and scalability of the system need to be done.

Keywords: SPARQL, Massive RDF data, Apache Flink

1. Introduction

The amount and size of datasets represented in the Resource Description Framework (RDF) [1] have dramatically increased during the recent years, requiring more efficient query evaluation techniques. Several proposals have been documented in the state of the art on the use of Big Data technologies for storing and querying RDF data [2–7]. Some of these proposals have focused on executing SPARQL queries on the MapReduce Programming Model [8] and its implementation, Hadoop [9].

However, more recent trends in Big Data technologies have emerged (e.g., Apache Spark [10], Apache Flink [11], Google DataFlow [12]). They use distributed in-memory processing and promise to deliver higher performance data processing than traditional MapReduce platforms [13]. These technologies are actually widely used in research projects and large companies (e.g., Google, Twitter, and Netflix).

To analyze whether these technologies can be used to provide support for query evaluation over large RDF dataset, in particular, we will work with Apache Flink, an open source platform for distributed stream and batch data processing. Developers can write their own programs by using fluent APIs such as the DataSet API, DataStream API, and Table API & SQL. Flink Programs are regular programs written in Java or Scala, which implement multiple transformations (e.g., filter, map, join, group) on distributed collections which are initially created from sources (e.g., by reading from files). Results are returned via sinks, which may, for example, write the data to (distributed) files,

[*] Corresponding author. E-mail: andres-m.castillo@correounivalle.edu.co.

or to the standard output (e.g., to the command line terminal).

Even though the PACT Programming Model and execution framework is described in [14], the set of initial transformations of the language (i.e., map, reduce, cross, cogroup, match) are formally described from the point of view of distributed data processing. However, in our approach we describe it from a point of view of how operations transform the dataset.

Hence, the main challenge that we need to address is **how to transform SPARQL queries into programs that use the Apache Flink DataSet API**. In this paper, we present our approach for SPARQL query evaluation over massive static RDF datasets, called SPARQL2FLINK, which is a tool at the application level, based on Apache Flink. To summarize, the main contributions of this paper are the following:

1. A formal definition of the Apache Flink's set transformations.
2. A formal definition of the semantic correspondence between SPARQL Algebra operators and Apache Flink's set transformations.
3. An open source implementation of the latter, available on Github under the MIT license, which transform a SPARQL query into an Apache Flink program. We assume that writing a SPARQL query is easier than writing program using the Apache Flink DataSet API to deal with an RDF dataset.

This is a preliminary work towards making such scalable queries processable in a framework like Apache Flink, and further work is needed in optimizing the resulting Flink programs so as to ensure that queries can be run over large RDF datasets as described as part of our motivation. So far we have focused on providing the formal descriptions and correspondences, and on a preliminary implementation which is shown to be correct in an experimental setup. It is out of our scope to produce a formal proof of correctness.

The remainder of the paper is organized as follows: In Section 2 we present a brief overview of RDF, SPARQL, PACT Programming Model, and Apache Flink. In Section 3 we describe our approach for transforming a SPARQL query into a Apache Flink program. In Section 4 we present an implementation of the transformations described in Section 3 as a Java library. In Section 5 we present and analyze the results of our evaluation using an adaptation of the Berlin SPARQL Benchmark [15]. In Section 6 we present related work on SPARQL query processing over Big Data Technologies. Finally, Section 7 presents conclusions and interesting issues for future work.

2. Background

2.1. RDF

The Resource Description Framework (RDF) [1] is a W3C recommendation for the representation of data on the Semantic Web. There are different serialization formats for RDF documents (e.g., RDF/XML, N-Triples, N3, Turtle). In the following, some essential elements of the RDF terminology are defined in an analogous way as it is done by Perez et al. in [16] and [1].

Definition 1 (**RDF Terms and Triples**). *Assume there are pairwise disjoint infinite sets I, B, and L (IRIs, Blank nodes, and literals). A tuple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an* RDF triple. *In this tuple, s is called the subject, p the predicate, and o the object. We denote T (the set of RDF terms) as the union of IRIs, blank nodes and literals, i.e. $T = I \cup B \cup L$. IRI (Internationalized Resource Identifier) is a generalization of URI (Uniform Resource Identifier). URIs represent common global identifiers for resources across the Web.*

Definition 2 (**RDF Dataset**). *An* RDF dataset $\mathcal{DS}$ *is a set:*

$$\mathcal{DS} = \{g_0, (\mu_1, g_1), (\mu_2, g_2) \ldots, (\mu_n, g_n)\}$$

where g_0 and g_i are RDF graphs, and each corresponding u_i is a distinct IRI. g_0 is called the default graph, *while the others are called* name graph.

2.2. SPARQL

The SPARQL Protocol and RDF Query Language (SPARQL) [17] is the W3C recommendation to query RDF. There are four query types: SELECT, ASK, DESCRIBE, and CONSTRUCT. In this paper we focus on SELECT queries. The basic SELECT query consists of three parts separated by the keywords PREFIX, SE-

LECT and WHERE. The PREFIX part allows to define shorten IRIs; The SELECT part identifies the variables to appear in the query result; The WHERE part provides the Basic Graph Pattern (BGP) to match against the input data graph. A definition of terminology comprising the concepts of *Triple and Basic Graph Pattern, Mappings, Basic Graph Patterns and Mappings, Subgraph Matching, Value Constraint, Built-in condition, Graph pattern expression, Graph Pattern Evaluation* and *SELECT Result Form* is done by Perez et al. in [16] and [18]. We encourage the reader to refer to these papers before going ahead.

2.3. The PACT Programming Model

The PACT Programming Model [14] is considered as a generalization of MapReduce. The PACT Programming Model operates on a key/value data model and is based on so-called *Parallelization Contracts* (PACTs). A PACT consists of a system-provided second-order function (called *Input Contract*) and a user-defined first-order function (UDF) which processes custom data types. The PACT Programming Model provides a initial set of five *Input Contract* that include two *Single-Input Contracts*: map and reduce as known from MapReduce which apply to user-defined function with a single input, and three additional *Multi-Input Contracts*: cross, cogroup, and match which apply to user-defined function for multiple inputs. As in the previous subsection, we encourage the reader to refer to the work of Battre at al. [14] for a complete review of the definitions concerning to the concepts of *Simple-Input Contract, mapping function, map, reduce, Multi-Input Contract, cross, cogroup* and *match*.

2.4. Apache Flink

Apache Flink [11] is an open source framework for distributed stream and batch data processing which had its origins in the Stratosphere Project [19]. The main components of Apache Flink architecture are: the Core, the APIs (e.g., DataSet, DataStream, Table & SQL), and the Libraries (e.g., FlinkML, Gelly). The Apache Flink core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance, for distributed computations over data streams. Among the APIs, the DataSet API allows processing finite datasets (batch processing), the DataStream API processes potentially unbounded data streams (stream processing), the Table & SQL API allows the composition of queries from relational operators. The SQL support is based on Apache Calcite [20], which implements the SQL standard. The libraries are built upon those APIs. The Gelly library provides methods and utilities for the development of graph analysis applications, the FlinkML library provides a set of scalable machine library algorithms (e.g., for supervised learning, unsupervised learning, data preprocessing, recommendation), and the FlinkCEP library allows complex events processing in streams. Another feature of Apache Flink is the optimizer, called Nephele [21], which transform a PACT program into a *Job Graph* [22]. Nephele optimizer is based on the PACT Programming Model [14]. Apache Flink provides several PACT operators for data transformation (e.g., filter, map, join, group).

3. Mapping SPARQL queries to an Apache Flink program

This section presents our first two contributions: the formal description of the Apache Flink's set transformations and the semantic correspondence between SPARQL Algebra operators and Apache Flink's set transformations.

3.1. PACT Data Model

We describe the PACT Data Model in a way similar to the one in [23], which is centered around the concepts of data sets and records. We assume a universal multi-set of records $\mathcal{T}$. In this way, a dataset $\mathcal{T}_1 = \{t_1, \ldots, t_p\}$ is a collection of records. Consequently, each dataset $\mathcal{T}_1$ is a subset of the universal multi-set $\mathcal{T}$, i.e, $\mathcal{T}_1 \subseteq \mathcal{T}$. A record $t = [k_1 : v_1, \ldots, k_n : v_n]$ is an unordered list of key-value pairs. The semantics of the keys and values, including their type is left to the user-defined functions that manipulate them [23]. We employ the record keys to define some PACT operators. It is possible to use numbers as the record keys; in the special case where the keys of a record t is the set $\{1, 2, \ldots, n\}$ for some $n \in \mathbb{N}$, we say t is a tuple. For sake of simplicity, we write $t = [v_1, \ldots, v_n]$ instead of tuple $t = [1 : v_1, \ldots, n : v_n]$. Two records $r_1 = [k_{1,1} : v_{1,1}, \ldots, k_{1,n} : v_{1,n}]$ and $r_2 = [k_{2,1} : v_{2,1}, \ldots, k_{2,m} : v_{2,m}]$ are equal ($r_1 \equiv r_2$) iff $n = m$ and $\forall_{i \in \{1,\ldots,n\}}, \exists_{j \in \{1,\ldots,m\}}. k_{1,i} = k_{2,j} \wedge r_1[k_{1,i}] = r_2[k_{2,j}]$.

Following, we present a precise definition for map, reduce, filter, project, match and outer match PACT operators. Additionally, we present the definition of some auxiliary notions.

Definition 3 (**Map Transformation**). *Let $\mathcal{T}' \subseteq \mathcal{T}$ be a data-set and given a function f ranging over $\mathcal{T}'$, i.e., $f : \mathcal{T}' \to \mathcal{T}$, we define a* map transformation *as follows:*

$$map(\mathcal{T}', f) = \mathcal{T}'' = \{[k'_1 : v'_1, \ldots, k'_m : v'_m] \mid \exists_{(k_1:v_1,\ldots,k_n:v_n)\in\mathcal{T}'} .\ f([k_1 : v_1, \ldots, k_n : v_n]) = [k'_1 : v'_1, \ldots, k'_m : v'_m]\}$$

Correspondingly, the map transformation takes each record $t = [k_1 : v_1, \ldots, k_n : v_n]$ of a dataset $\mathcal{T}'$ and produces a new record $t' = [k'_1 : v'_1, \ldots, k'_m : v'_m]$ by means of a user function f. Records produced by function f can differ with respect to the original records. First, the number of key-value pairs can be different, i.e., $n \neq m$. Second, the keys $k'_1, \ldots, k'_m$ do not have to match with the keys $k_1, \ldots, k_n$. Last, the data type associated to each value can differ.

In order to define the remaining PACT operators, we first define the record projection and record value projection operations. While the record projection enables us to build a new record, which is made up of the key-value pairs associated to some specific keys, the record value projection allows obtaining the values associated to some specific keys. Record projection can be defined as follows:

3.2. Formalization of Apache Flink Transformations

In this section, we propose a formal interpretation of the PACT operators that will be used to establish a correspondence with the SPARQL Algebra operators. These PACT operators are implemented by the Apache Flink DataSet API. This correspondence is necessary before we establish an scheme to translate SPARQL queries to Flink programs in order to exploit the capabilities of this framework for data processing.

Definition 4 (**Record Projection**). *Let $t = [k_1 : v_1, \ldots, k_n : v_n] \in \mathcal{T}$ be a record, we define the* projection of t over a set of keys $I = \{i_1, \ldots, i_m\}$ *(denoted as $t(i_1, \ldots, i_m)$) as follows:*

$$t(i_1, \ldots, i_m) = \{(i', v) \mid (i', v) \in t \ \wedge \ i' \in \{i_1, \ldots, i_m\}\}$$

In this way, by means of a record projection, a new record is obtained only with the key-value pairs associated to some key in the set $I = \{i_1, \ldots, i_m\}$. For example, consider a record t_1 as follows:

$$t_1 = [\mathsf{name} : \texttt{"Alex"}, \mathsf{occupation} : \texttt{"Actor"}, \mathsf{age} : 35]$$

Then, a record projection as $t_1(\mathsf{name}, \mathsf{age})$ will produce a new record as follows:

$$[\mathsf{name} : \texttt{"Alex"},\ \mathsf{age} : 35]$$

Record value projection can be defined as follows:

Definition 5 (**Record Value Projection**). *Let $t = [k_1 : v_1, \ldots, k_n : v_n] \in \mathcal{T}$ be a record, we define the* value projection of t over a sequence of keys $I = [i_1, \ldots, i_m]$ *(denoted as $t[i_1, \ldots, i_m]$) as follows:*

$$[i_1, \ldots, i_m] = [v_1, v_2, \ldots, v_m]$$
$$where\ \forall_{j\in\{1,\ldots,m\}} .\ (i_j : v_j) \in t.$$

It is worth to precise that the record value projection takes a record and produces a sequence. In this way, in this operation the keys order in sequence $[i_1, \ldots, i_m]$ is considered for the result construction. Likewise, the result of the record value projection can contain repeated elements. For example, consider a record t_2 as follows:

$$t_2 = [\mathsf{name} : \text{“}\texttt{Alex}\text{”}, \mathsf{occupation} : \text{“}\texttt{Actor}\text{”}, \mathsf{age} : 35, \mathsf{birth-city} : \text{“}\texttt{Madrid}\text{”}, \mathsf{residence-city} : \text{“}\texttt{Madrid}\text{”}]$$

Then, a record value projection as $t[\mathsf{name}, \mathsf{birth\text{-}city}, \mathsf{residence\text{-}city}]$ will produce a sequence as follows:

$$[\texttt{"Alex"}, \texttt{"Madrid"}, \texttt{"Madrid"}]$$

Besides, the record value projection $t[\mathsf{birth\text{-}city}, \mathsf{name}, \mathsf{residence\text{-}city}]$ will produce a sequence as the following:

$$[\texttt{"Madrid"}, \texttt{"Alex"}, \texttt{"Madrid"}]$$

Therefore, it is held that the result of operation $t[\mathsf{name}, \mathsf{birth\text{-}city}, \mathsf{residence\text{-}city}]$ is different to the result of operation $t[\mathsf{birth\text{-}city}, \mathsf{name}, \mathsf{residence\text{-}city}]$. Now, we define the reduce PACT transformation as follows:

Definition 6 (**Reduce Transformation**). *Let $\mathcal{T}' \subseteq \mathcal{T}$ be a data-set and given a set of keys K and a function f*

ranging over the power set of $\mathcal{T}'$, i.e., $f : \mathbb{P}(\mathcal{T}') \to \mathcal{T}$, we define a reduce transformation *as follows:*

$$reduce(\mathcal{T}', f, K) = \mathcal{T}'' = \{[k_1 : v_1, \ldots, k_m : v_m] \mid \exists_{t_1,\ldots,t_n \in \mathcal{T}'} . f(\{t_1, \ldots, t_n\}) = [k_1 : v_1, \ldots, k_m : v_m] \ \wedge t_1(K) \equiv \ldots \equiv t_n(K)\}$$

In this way, the reduce transformation takes a dataset $\mathcal{T}'$ and groups records, which have the same values for the keys in set K. It is worth to highlight that a record projection is used. Then, it applies user function f over each group and produces new records $[k_1 : v_1, \ldots, k_m : v_m]$.

Definition 7 (**Filter Transformation**). *Let $\mathcal{T}' \subseteq \mathcal{T}$ be a data set and given a function f ranging over $\mathcal{T}'$ to boolean values, i.e., $f : \mathcal{T}' \to \{\text{true}, \text{false}\}$, we define a* filter transformation *as follows:*

$$filter(\mathcal{T}', f) = \mathcal{T}'' = \{t \mid f(t) = \text{true}\}$$

The filter transformation evaluates predicate f with every record of a dataset $\mathcal{T}'$ and it selects only those records with which f returns true.

Definition 8 (**Project Transformation**). *Let $\mathcal{T}' \subseteq \mathcal{T}$ be a data set and given a set of keys $K = \{k'_1, \ldots, k'_m\}$, we define a* project transformation *as follows:*

$$project(\mathcal{T}', K) = \{t' \mid \exists_{t \in \mathcal{T}'} . \ t' = t(K)\}$$

While filter transformation allows selecting some specific records according some criteria, which are expressed in the semantics of a function f, the project transformation enables us to obtain some specific fields of the records of a dataset $\mathcal{T}'$. For this purpose, it is applied a record projection operation (c.f. Def. 4) to each record in $\mathcal{T}'$ with respect to a set of keys K. It is worth to highlight that the result of a project transformation is a multi-set due to several records can have same values in the keys of set K.

Previous PACT transformations takes as a parameter a dataset $\mathcal{T}'$ and produces as a result a new dataset according to a specific semantics. Nevertheless, there exist a lot of data sources, and eventually it is necessary to process and combine multiple datasets. In consequence, there are some PACT transformations taking as parameters two or more datasets [14]. Following, we present a formal interpretation of the most important multi-datasets transformations including matching, grouping and union.

Definition 9 (**Match Transformation**). *Let $\mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{T}$ be data sets, given a function f ranging over $\mathcal{T}_1$ and $\mathcal{T}_2$, i.e., $f : \mathcal{T}_1 \times \mathcal{T}_2 \to \mathcal{T}$ and given sets of keys K_1 and K_2, we define a* match transformation *as follows:*

$$match(\mathcal{T}_1, \mathcal{T}_2, f, K_1, K_2) = \mathcal{T}' = \{[k_1 : v_1, \ldots, k_n : v_n] \mid \exists_{t_1 \in \mathcal{T}_1, \ t_2 \in \mathcal{T}_2} . \ f(t_1, t_2) = [k_1 : v_1, \ldots, k_n : v_n] \ \wedge t_1[K_1] = t_2[K_2]\}$$

Consequently, match transformation takes each pair of records (t_1, t_2) built from datasets $\mathcal{T}_1$ and $\mathcal{T}_2$ and applies user function f with those pairs for which the values in t_1 with respect to keys in K_1 coincide with the values in t_2 with respect to keys in K_2. For this purpose, it checks this correspondence by means of a record value projection (c.f. Def. 5). Intuitively, the match transformation enables us to group and process pairs of records related for some specific criterion. In some cases, it is necessary to match and process a record in a dataset even if there not exists a correspondent record in the another dataset. The outer match transformation extends the match transformation to enable such a matching. Outer match transformation is defined as follows:

Definition 10 (**Outer Match Transformation**). *Let $\mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{T}$ be data sets, given a function f ranging over $\mathcal{T}_1$ and $\mathcal{T}_2$, i.e., $f : \mathcal{T}_1 \times \mathcal{T}_2 \to \mathcal{T}$ and given sets of keys K_1 and K_2, we define a* outer match transformation *as follows:*

$$outerMatch(\mathcal{T}_1, \mathcal{T}_2, f, K_1, K_2) = \mathcal{T}' = \{[k_1 : v_1, \ldots, k_n : v_n] \mid \exists_{t_1 \in \mathcal{T}_1}, ((\exists_{t_2 \in \mathcal{T}_2} . f(t_1, t_2) = [k_1 : v_1, \ldots, k_n : v_n] \ \wedge t_1[K_1] = t_2[K_2]) \oplus f(t_1, [\,]) = [k_1 : v_1, \ldots, k_n : v_n])\}$$

In this manner, the outer match transformation is similar to the match transformation but it allows to apply the user function f with a record t_1 although there not exists a record t_2 that matches with record t_1 with respect to keys K_1 and K_2 respectively.

In addition to the match and outer match transformations, the cogroup transformation enables us to group records in two datasets. Those records must coincide with respect to a set of keys. Following, it is defined the cogroup transformation:

Definition 11 (**CoGroup Transformation**). *Let $\mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{T}$ be data sets, given a function f ranging over*

$\mathbb{P}(\mathcal{T}) \rightarrow \mathcal{T}$ and given a set of keys K, we define a cogroup transformation *as follows:*

$$\begin{aligned}&cogroup(\mathcal{T}_1, \mathcal{T}_2, f, K) = \mathcal{T}' \\ &= \{[k_1 : v_1, \ldots, k_p : v_p] \mid \exists_{s_1,\ldots,s_n \in \mathcal{T}_1,\ t_1,\ldots t_m \in \mathcal{T}_2}. \\ &f(\{s_1, \ldots, s_n, t_1, \ldots, t_m\}) = [k_1 : v_1, \ldots, k_p : v_p] \wedge \\ &s_1[K] = \ldots = s_n[K] = \quad t_1[K] = \ldots = t_m[K]\}\end{aligned}$$

Intuitively, the cogroup transformation processes conforms groups with the records in datasets $\mathcal{T}_1$ and $\mathcal{T}_2$ for which the values of the keys in K are equal. Then, it applies a user function f on each of those groups. Finally, the union transformation creates a new dataset with every record in two datasets $\mathcal{T}_1$ and $\mathcal{T}_2$. It is defined as follows:

Definition 12 (Union Transformation). *Let $\mathcal{T}_1, \mathcal{T}_2 \subseteq \mathcal{T}$ be data sets, we define a* union transformation *as follows:*

$$\begin{aligned}&union(\mathcal{T}_1, \mathcal{T}_2) = \mathcal{T}' = \{[k_1 : v_1, \ldots, k_n : v_n] \mid \\ &[k_1 : v_1, \ldots, k_n : v_n] \in \mathcal{T}_1 \vee \\ &[k_1 : v_1, \ldots, k_n : v_n] \in \mathcal{T}_2\}\end{aligned}$$

3.3. Correspondence between SPARQL Algebra operators and Apache Flink transformations

In this section, we propose a semantics correspondence between SPARQL algebra operators and the PACT transformations implemented in the Apache Flink DataSet API. We use the formalization of PACT transformations in the previous section to provide an intuitive and correct mapping of semantics elements of SPARQL queries. In this way, this correspondence validates our implementation, which is presented in the next section.

As described in Section 2.1, a RDF dataset is a set of triples. We assume that each field of a record t can be accessed by means of indexes 0, 1 and 2. Likewise, we assume that RDF triple patterns are triples $[s, p, o]$ where s, p, o can be variables or values. Also, the result of the application of each PACT transformation is intended to be a solution mapping, i.e., sets of key-value pairs with keys as RDF variables.

Following, we present the definition of our encoding of SPARQL queries as PACT transformations. First, we define the encoding of the graph pattern evaluation as follows:

Definition 13 (Graph Pattern PACT Encoding). *Let P be a graph pattern and $\mathcal{D}$ be an RDF dataset, the* PACT encoding *of the evaluation of P over $\mathcal{D}$, denoted by $||P||^{\mathcal{D}}$, is defined recursively as follows:*

1. If P is a triple pattern $[s, p, o]$ then:

$$||P||^{\mathcal{D}} = map(filter(\mathcal{D},\ \mathsf{f}_1),\ \mathsf{f}_2)$$

where function f_1 is defined as follows:

$$\begin{aligned}\mathsf{f}_1(t) = \mathsf{pred}(s, t[0]) \ \wedge\ &\mathsf{pred}(p, t[1]) \\ &\wedge \mathsf{pred}(o, t[2])\end{aligned}$$

$$\mathsf{pred}(a, b) = \begin{cases} \mathsf{true} & \texttt{if } a \in Var \\ \\ a = b & \texttt{otherwise}\end{cases}$$

and function f_2 is defined as follows:

$$\mathsf{f}_2(t) = \begin{cases} [s : t[0], p : t[1], o : t[2]] & \texttt{if } c1 \\ [s : t[0],\ p : t[1]] & \texttt{if } c2 \\ [s : t[0],\ o : t[2]] & \texttt{if } c3 \\ [p : t[1],\ o : t[2]] & \texttt{if } c4 \\ [s : t[0]] & \texttt{if } c5 \\ [p : t[1]] & \texttt{if } c6 \\ [o : t[2]] & \texttt{if } c7 \\ [\,] & \texttt{otherwise}\end{cases}$$

$$\begin{aligned}&\textit{where } c1 = s, p, o \in Var \\ &c2 = s, p \in Var \ \wedge\ o \notin Var \\ &c3 = s, o \in Var \ \wedge\ p \notin Var \\ &c4 = p, o \in Var \ \wedge\ s \notin Var \\ &c5 = s \in Var \ \wedge\ p, o \notin Var \\ &c6 = p \in Var \ \wedge\ s, o \notin Var \\ &c7 = o \in Var \ \wedge\ s, p \notin Var\end{aligned}$$

2. If P is $(P_1$ AND $P_2)$ then:

$$||P||^{\mathcal{D}} = match(||P_1||^{\mathcal{D}},\ ||P_2||^{\mathcal{D}},\ \mathsf{f},\ K,\ K)$$

where $K = vars(P_1) \cap vars(P_2)$ and function f is defined as follows:

$$f(t_1, t_2) = t_1 \cup t_2$$

3. If P is $(P_1$ `OPT` $P_2)$ then:

$$||P||^{\mathcal{D}} = outerMatch(||P_1||^{\mathcal{D}},\ ||P_2||^{\mathcal{D}},\ \mathsf{f},\ K,\ K)$$

where $K = vars(P_1) \cap vars(P_2)$ and function f is defined as follows:

$$f(t_1, t_2) = t_1 \cup t_2$$

4. If P is $(P_1$ `UNION` $P_2)$ then:

$$||P||^{\mathcal{D}} = union(||P_1||^{\mathcal{D}},\ ||P_2||^{\mathcal{D}})$$

5. If P is $(P'$ `FILTER` $R)$ then:

$$\mathsf{f}(t) = \mathsf{exp}(R, t)$$

$$\mathsf{exp}(R, t) = \begin{cases} t[x]\ \mathsf{op}\ c & \texttt{if}\ c1 \\ t[x]\ \mathsf{op}\ t[y] & \texttt{if}\ c2 \\ \mathsf{exp}(R_1, t)\ \mathsf{op}\ \mathsf{exp}(R_2, t) & \texttt{if}\ c3 \end{cases}$$

$$\begin{aligned} \textit{where } c1 &= R\ \texttt{is}\ (x\ \mathsf{op}\ c)\ \texttt{for}\ x \in Var, \\ &\qquad \mathsf{op} \in \{=, <, \leqslant, >, \geqslant\} \\ c2 &= R\ \texttt{is}\ (x\ \mathsf{op}\ y)\ \texttt{for}\ x, y \in Var, \\ &\qquad \mathsf{op} \in \{=, <, \leqslant, >, \geqslant\} \\ c3 &= R\ \texttt{is}\ (R_1\ \mathsf{op}\ R_2)\ \texttt{for}\ \mathsf{op} \in \{\wedge, \vee\} \end{aligned}$$

In this way, the graph pattern PACT evaluation is encoded according to the recursive definition of a graph pattern P. More precisely, we have that:

- If P is a triple pattern, then records of dataset $\mathcal{D}$ are filtered (by means of function f_1) to obtain only the records that are compatible with the variables and values in $[s, p, o]$. Then, the filtered records are mapped (by means function f_2) to obtain solutions mappings that relate each variable to each possible value.
  If P is a join (left join) (it uses the SPARQL operators `AND` (`OPT`)), then it performs a *match* (*outermatch*) transformation between the recursive evaluation of subgraphs P_1 and P_2 with respect to a set K conformed by the variables in P_1 and P_2.
- If P is a union graph pattern, then there is a union transformation between the recursive evaluation of subgraphs P_1 and P_2.
- Finally, if P is a filter graph pattern, then it performs a $filter$ transformation over the recursive evaluation of subgraph P' where the user function f is built according to the structure of the filter expression R.

Additionally to the graph pattern evaluation, we present an encoding of the evaluation of SELECT and DISTINCT SELECT queries as well as the ORDER-BY and LIMIT modifiers. The selection encoding is defined as follows:

Definition 14 (Selection PACT Encoding). *Let $\mathcal{D}$ be an RDF dataset, P be a graph pattern, K be a finite set of variables and $Q = \langle P, K\rangle$ be a selection query over $\mathcal{D}$, the PACT Encoding of the evaluation of Q over $\mathcal{D}$ is defined as follows:*

$$||Q||^{\mathcal{D}} = project(||P||^{\mathcal{D}}, K)$$

Correspondingly, the selection query is encoded as a project transformation over the evaluation of the graph pattern P associated to the query with respect to a set of keys K conformed by the variables in the FROM part of the query. We make a subtle variation to define the distinct selection as follows:

Definition 15 (Distinct Selection PACT Encoding). *Let $\mathcal{D}$ be an RDF dataset, P be a graph pattern, K be a finite set of variables and $Q^* = \langle P, K\rangle$ be a distinct selection query over $\mathcal{D}$, the PACT Encoding of the evaluation of Q^* over $\mathcal{D}$ is defined as follows:*

$$||Q^*||^{\mathcal{D}} = reduce(project(||P||^{\mathcal{D}},\ K),\ \mathsf{f},\ K)$$

where function f is defined as follows:

$$\mathsf{f}(\{t_1, \ldots, t_n\}) = t_1$$

Clearly, the definition of the distinct selection PACT encoding is similar to the general selection query encoding. The main difference corresponds to a reduction step (*reduce* transformation) in which, the duplicate records, i.e. records with the same value in the keys of set K (the distinct keys) are reduced to only one occurrence.

The encoding of the evaluation of a order-by query is defined as follows:

Definition 16 (Order By PACT Encoding). *Let $\mathcal{D}$ be an RDF dataset, P be a graph pattern, k be a variable and $Q^* = \langle P, k\rangle$ be a order by query over $\mathcal{D}$, the PACT*

Encoding of the evaluation of Q^* over $\mathcal{D}$ is defined as follows:

$$||Q^*||^{\mathcal{D}} = \mathsf{order}(||P||^{\mathcal{D}},\ k)$$

where function order *is defined as follows:*

$$\mathsf{order}(M, k) = M'$$

where $M = \{t_1, \ldots, t_n\}$ and $M' = \{t'_1, \ldots, t'_n\}$ is a permutation of M such that $t'_i[k] < t'_{i+1}[k]$ for each $i \in \{1, \ldots, n-1\}$.

Thereby, the graph pattern associated to the query is first evaluated according to the encoding of its precise semantics. Then, the resulting solution mapping is ordered by means of a function order. Finally, the encoding ot the evaluation of a limit query is defined as follows:

Definition 17 (**Limit PACT Encoding**). *Let $\mathcal{D}$ be an RDF dataset, P be a graph pattern, m be an integer such that $m \geqslant 1$ and $Q^* = \langle P, m\rangle$ be a limit query over $\mathcal{D}$, the PACT Encoding of the evaluation of Q^* over $\mathcal{D}$ is defined as follows:*

$$||Q^*||^{\mathcal{D}} = \mathsf{limit}(||P||^{\mathcal{D}},\ m)$$

where function limit *is defined as follows:*

$$\mathsf{limit}(M, m) = M'$$

where $M = \{t_1, \ldots, t_n\}$ and $M' = \{t'_1, \ldots, t'_m\}$ such that $t'_i \in M$ and $|M'| = m$.

In this way, once the graph pattern associated to the query is evaluated, the result is shortened to consider only the m records according to the query.

4. Implementation

This section presents the last contribution. We implemented the transformations described in Section 3 as a Java library[1]. This library is composed by two modules, namely: *Mapper* and *Runner* as shown in Figure 1. The *Mapper* module transforms a declarative SPARQL query into a Apache Flink program (Flink program). The *Runner* module allows to execute the Flink program on an Apache Flink standalone. Apache Jena ARQ and Apache Flink libraries are shared among both modules.

According to Apache Flink, a typical Flink program consists of four basic stages: 1) Loading/Creating the initial data. 2) Specifying the transformations of the data. 3) Specifying where to put the results of the computations, and 4) triggering the program execution. The *Mapper* module is composed of three submodules, which are focused in the first three stages of a Flink program.

Load SPARQL Query File submodule loads the declarative SPARQL query from a file with `.rq` extension. Listing 1 shows an example of a SPARQL query in a declarative form that gives the names of all persons with their email, if they have it.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person ?name ?mbox
WHERE {
        ?person foaf:name ?name .
        OPTIONAL { ?person  foaf:mbox  ?mbox }
}
```

Listing 1: SPARQL query example

Translation Query To Logical Query Plan This submodule uses the Jena ARQ library to validate the SPARQL query syntax [17] translate it to a Logical Query Plan (LQP) o Parse Tree which contains the SPARQL Algebra operator. The LQP is represented with and RDF-centric syntax provided by Jena, which is called SPARQL Syntax Expression (SSE) [24]. Listing 2 shows a LQP of the SPARQL query example represented with SSE.

```
(project (?person ?name ?mbox)
 (leftjoin
   (bgp (triple ?person
         <http://xmlns.com/foaf/0.1/name> ?name))
   (bgp (triple ?person
         <http://xmlns.com/foaf/0.1/mbox> ?mbox))
 ))
```

Listing 2: SPARQL Syntax Expression of SPARQL query example

Convert Logical Query Plan To Flink program submodule convert each SPARQL Algebra operator in the query to a transformation from DataSet API of Apache Flink according to the correspondence described in Section 3. For instance, each *triple pattern* within a *Ba-*

[1]The source code is available in https://github.com/oscarceballos/sparql-to-flink

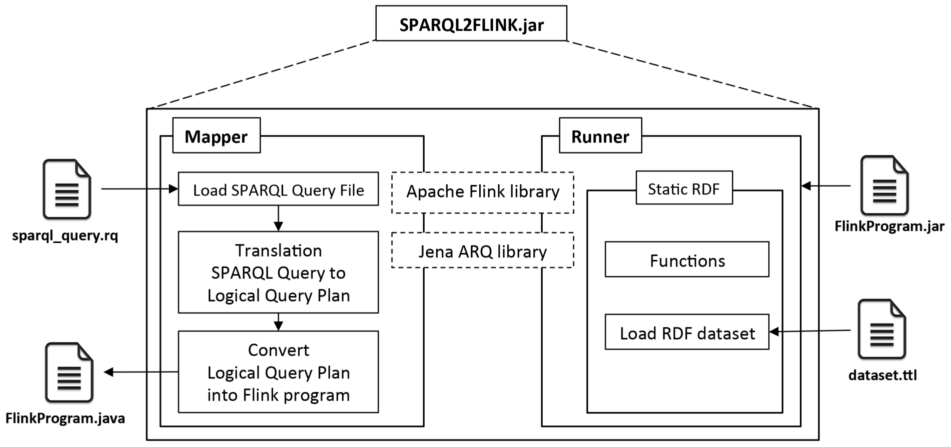

Fig. 1. Sparql2Flink conceptual architecture

sic Graph Pattern (BGP) is encoded as a combination of *filter* and *map* transformations, the *leftjoin* operator is encoded as a *leftOuterJoin* transformation, whereas the *project* operator is expressed as a *map* transformation. Listing 3 shows how to a SPARQL query example is converted into a Java Flink program.

```java
...
public class Query {
 public static void main(String[] a) throws Exception {

  /***** Environment (DataSet) and Source
                (static RDF dataset) *****/
  final ExecutionEnvironment env = ExecutionEnvironment
        .getExecutionEnvironment();
  DataSet<Triple> dataset = LoadTransformTriples
   .loadTriplesFromDataset(env, "dataset.ttl");
  /***** Applying Transformations *****/
  DataSet<SolutionMapping> sm1 = dataset
   .filter(new T2T_FF(null,
            "http://xmlns.com/foaf/0.1/name", null))
   .map(new T2SM_MF("?person", null, "?name"));

  DataSet<SolutionMapping> sm2 = dataset
   .filter(new T2T_FF(null,
            "http://xmlns.com/foaf/0.1/mbox", null))
   .map(new T2SM_MF("?person", null, "?mbox"));

  DataSet<SolutionMapping> sm3 = sm1.leftOuterJoin(sm2)
   .where(new SM_JKS(new String[]{"?person"}))
   .equalTo(new SM_JKS(new String[]{"?person"}))
   .with(new SM_LOJF(new String[]{"?person"}));

  DataSet<SolutionMapping> sm4 = sm3
   .map(new SM2SM_PF(new String[]{"?person", "?name",
                   "?mbox"}));

  //***** Sink  *****
  sm4.print();
 }
}
```

Listing 3: Java Flink Program

The Flink program built by the SPARQL2FLINK library must be deployed using the maven package utility to produce a `jar` file that can be executed in an Apache Flink standalone or server. This module is composed of two submodules, which are focused in the last stage to build a Flink program. *Load RDF Dataset* submodule loads an RDF dataset in Turtle (i.e., a file with `.ttl` extension). *Functions* submodule contains several Java classes that allow to solve the transformations within the Flink program.

5. Evaluation and Result

There are many benchmarks designed for evaluating RDF Systems, for example: LUBM [25], WatDiv [26], SP2Bench [27], DBpedia Bench [28], Berlin SPARQL Benchmark (BSBM) [15], and RBench [29], among others. In our case, we reused BSBM for testing empirically the correctness of the results of each SPARQL query proposed in BSBM and its transformation into a Flink program.

5.1. Test plan

The SPARQL2FLINK library does not implement the SPARQL protocol and cannot be used as a SPARQL endpoint[2]. For this reason we do not use the test drive proposed in BSBM. In contrast, we followed the next steps:

[2]Note that this does not impose a strong limitation to our approach at this stage. This is an engineering task that will be supported in the future

1. Setup the environment on which the test will be carried out, describing their characteristics and configuration.
2. Verify which SPARQL queries template can be run on Apache Flink. If necessary, modify the query template omitting SPARQL operators and expressions that are not yet implemented by the library.
3. Generate a dataset with a number of different products in order to instantiate each SPARQL query template proposed in BSBM.
4. Transform the SPARQL queries template into a Flink program through our library.
5. Run each SPARQL query instantiated in Apache Jena and its corresponding Flink program in an Apache Flink standalone.
6. Compare the result of both executions.

5.2. Setup

We have run the SPARQL2FLINK library on a Mac Sierra operating system, on a Laptop with Intel Core Duo i5 2.8GHz, 8 GB RAM, and 1TB solid state disk. The main applications are Apache Jena Triple Store and Apache Flink 1.3.2 standalone.

As the queries depend on the dataset on which the queries will be run, a dataset with 100 products and 40.177 triples was generated by using the BSBM data generator. The generated dataset describes an e-commerce use case including products, vendors, user ratings with comments and other related items. The explore use case describes the scenario of users searching for products and consists of 12 query templates which comply to SPARQL 1.0.

5.3. Results

Based on the dataset generated, the SPARQL queries templates Q1, Q2, Q3, Q4, Q5, Q7, Q8, Q10 and Q11 were instantiated and transformed into a Flink program[3]. The query template Q6 was not transformed because omitting the *FILTER regex(?label, %word1%)* expression the resulting query not contribute significant challenge. The queries template Q9 and Q12 were not transformed because the *DESCRIBE* and *CONSTRUCT* query types are not supported by the SPARQL2FLINK library. Table 1 shows which queries are supported, partially supported, and not supported. In the case where the query is partially supported we details how the query was modified to be able to transform it into a Flink program.

In order to testing empirically the correctness of the results of the transformation, each SPARQL query instantiated was executed in Apache Jena and its corresponding Flink program was executed in an Apache Flink standalone. Table 2 describes the result of the empirical correctness test.

6. Related Work

Several proposals have been documented on the use of Big Data technologies for storing and querying RDF data [2–7]. The most common way so far to query massive static RDF data has been rewriting SPARQL queries over the MapReduce Programming Model [8] and executing them on Hadoop [9] Ecosystems. A detailed comparison of existing approaches can be found in the survey presented in [4]. This work provide a comprehensive description of RDF data management in large-scale distributed platforms, where storage and query processing are performed in a distributed fashion, but under a centralized control. The survey classifies the systems according to the way in which they implement three fundamental functionalities: data storage, query processing, and reasoning; this determines how the triples are accessed and the number of MapReduce jobs. Additionally, it details the solutions adopted to implement those functionalities.

Another survey is [30] that presents gave high level overview of RDF data management, focusing on several approaches that have been adopted. The discussion focused on centralized RDF data management, distributed RDF systems, and querying over the Linked Open Data. In particular, in the distributed RDF systems identified and discussed four classes of approaches: cloud-based solutions, partitioning-based approaches, federated SPARQL evaluation systems, and partial evaluation-based approach.

In recent years, new trends in Big Data Technologies such as Apache Spark [10], Apache Flink [11] and Google DataFlow [12] have been proposed. They use distributed in-memory processing and promise to deliver higher performance data processing than traditional MapReduce platforms [13]. In particular, Apache Spark implements a programming model similar to MapReduce but extends it with two abstractions: *Resilient Distributed Datasets* (RDDs) [31] and

[3]The dataset generated, the SPARQL queries instantiated, and the Flink program transformations are available in https://github.com/oscarceballos/sparql2flink

Table 1
Sparql queries supported by SPARQL2FLINK library. The labels **S**, **PS**, and **NS** under the Support column means that the query is **S**upported, **P**artially **S**upported, and **N**ot **S**upported, respectively.

Query	Support	Reason	Modification
Q1	S		
Q2	S		
Q3	PS	The *bound* function within the *FILTER* operator is not supported by our library	The *FILTER (!bound(?testVar))* expression is omitted
Q4	PS	The *OFFSET* operator is not supported by our library	The *OFFSET 5* expression is omitted
Q5	PS	The condition within the *FILTER* operator contains addition or subtraction operations which are not supported by our library	The operations within the *FILTER condition* are changed by a constant in order to evaluate && operator
Q6	PS	The *regex* function within the *FILTER* operator is not supported by our library	The *FILTER* expression is omitted
Q7	S		
Q8	PS	The *langMatches* and *lang* functions within the *FILTER operator* are not supported by our library	The *FILTER langMatches(lang(?text), "EN")* expression is omitted
Q9	NS	*DESCRIBE* query type is not supported by our library	
Q10	S		
Q11	S		
Q12	NS	*CONSTRUCT* query type is not supported by our library	

Table 2
Results of the empirical correctness test of SPARQL queries vs Flink programs over Apache Jena and Apache Flink standalone, respectively.

Q1	Q2	Q3	Q4	Q5	Q7	Q8	Q10	Q11
Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Data Frames (DF) [32]. RDDs are a distributed, immutable and fault-tolerant memory abstraction and DF is a compressed and schema-enabled data abstraction.

A few proposals focused on SPARQL query processing and optimization has been built on top of Apache Spark. For example, S2RDF [33] proposes a novel relational schema and relies on a translation of SPARQL queries into SQL for being executed using Spark SQL. The new relational partitioning schema for RDF data is called Extended Vertical Partitioning (ExtVP) [34]. In this schema, the RDF triples are distributed in relations of two columns each one corresponding to an RDF term (one for the the subject and one for the object). The relations are computed at data load-time using semi-joins, akin to the concept of Join Indexes [35] in relational databases, to limit the number of comparisons when joining triple patterns. Each triple pattern of a query is translated into a single SQL query and the query performance is optimized using the set of statistics and additional data structures computed during the data pre-processing step.

The authors in [36] propose and compare five different query processing approaches based on different join execution models (i.e, partitioned join and broadcast join) on Spark components like RDD, DF, and SQL API. Beside, they propose a formalization for evaluating the cost of SPARQL query processing in a distributed setting. The main conclusion about these is that Spark SQL does not (yet) fully exploit the variety of distributed join algorithms and plans that could be executed using the Spark platform and propose some first directions for more efficient implementations.

On the other hand, Google DataFlow [12] is a Programming Model and Cloud Service for batch and stream data processing with a unified API. It is built upon Google technologies, such as MapReduce for batch processing, FlumeJava [37] for programming model definition and MillWheel [38] for stream processing. Google released the Dataflow Software Development Kit (SDK) as an open source Apache project, named Apache Beam [39]. To the best of our knowledge, there are no works reported which use of Google DataFlow to processing SPARQL queries on massive static RDF data.

7. Conclusions and Future Work

We have presented an approach for transforming SPARQL queries into Apache Flink programs for querying massive static RDF data. The main contributions of this paper are the formal definitions of the Apache Flink's set transformations, the definition of the semantic correspondence between Apache Flink's set transformations and the SPARQL Algebra operators, and the implementation of our approach as a library (Maven project), which transforms a SPARQL query into an Apache Flink program.

For the sake of simplicity, we limit to SELECT queries with SPARQL Algebra operators such as Basic Graph Pattern, AND (Join), OPTIONAL (LeftJoin), UNION, PROJECT, DISTINCT, ORDER BY, LIMIT and specific FILTER expressions of the form (*variable operator variable*), (*variable operator constant*), and (*constant operator variable*); the operator within Filter expression are logical connectives (&&, ||), inequality symbols ($<, \leqslant, \geqslant, >, !$), and the equality symbol (=).

Our approach does not support queries containing clauses or operators FROM, FROM NAMED, FILTER, CONSTRUCT, DESCRIBE, ASK, INSERT, DELETE, LOAD, CLEAR, CREATE/DROP GRAPH, MINUS or EXCEPT.

As future work we will focus on two aspects: optimization and RDF stream processing. On the one hand, we will study the possibility to adapt in Apache Flink some optimization techniques inherent in the processing of SPARQL queries like dictionaries, indexing, and multi-way join operators. On the other hand, we will extend the former approach to RDF stream processing. For this, we will formally define the Apache Flink's DataStream API set transformations. Then, we will define the semantic correspondence between Apache Flink's DataStream API set transformations and the RDF Stream Processing Query Language (RSP-QL) [40] operators. Finally, we will to provide an implementation by extending the Sparql2Flink project to transform RSP queries into Apache Flink Programs.

Acknowledgement

Oscar Ceballos is supported by the Colombian Administrative Department of Science, Technology and Innovation (Colciencias) under the PhD scholarship program 617-2013. Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

Declarations of interest: none

Appendix A. SPARQL Algebra definition

In the following, we introduce some SPARQL terminology in an analogous way as it is done by [16].

Definition 18 (**Triple and Basic Graph Pattern**). *A tuple* $t \in (I \cup V) \times (I \cup V) \times (I \cup L \cup V)$ *is a* triple pattern. *A* Basic Graph Pattern *is a finite set of triple patterns. Given a triple pattern t, var(t) is the set of variables occurring in t. Similarly, given a basic graph pattern P,* $var(P) = \bigcup_{t \in P} var(t)$*, i.e. var(P) is the set of variables occurring in P.* Note 1. *In these definitions, triple and basic graph patterns do not contain blank nodes.*

Definition 19 (**Mappings**). *A mapping* μ *from V to T is a partial function* $\mu : V \rightarrow T$*. The domain of* μ*,* $dom(\mu)$*, is the subset of V where* μ *is defined. The empty mapping* $\mu_\emptyset$ *is a mapping such that* $dom(\mu_\emptyset) = \emptyset$ *(i.e.* $\mu_\emptyset = \emptyset$*).*

Definition 20 (**Basic Graph Patterns and Mappings**). *Given a triple pattern t and a mapping* μ *such that* $var(t) \subseteq dom(\mu)$*,* $\mu(t)$ *is the triple obtained by replacing the variables in according to* μ*. Given a basic graph pattern P and a mapping* μ *such that* $var(P) \subseteq dom(\mu)$*, we have that* $\mu(P) = \bigcup_{t \in P}\{\mu(t)\}$*, i.e.* $\mu(P)$ *is the set of triples obtained by replacing the variables in the triples of P according to* μ*.*

Definition 21 (**Subgraph Matching**). *Let G be an RDF graph over T, and P a basic graph pattern. The evaluation of P over G, denoted by* $[\![P]\!]_{\mathcal{D}}$ *is defined as the set of mappings* $[\![P]\!]_{\mathcal{D}} = \{\mu : V \rightarrow T | dom(\mu) = var(P) \text{ and } \mu(P) \subseteq G\}$*. Note 2. For every RDF graph G,* $[\![\emptyset]\!]_G = \{\mu_\emptyset\}$*, i.e. the evaluation of an empty basic graph pattern against any graph always results in the set containing only the empty mapping. For every basic graph pattern* $P \neq \emptyset$*,* $[\![P]\!]_\emptyset = \emptyset$*.*

Definition 22 (**Value Constraint**). *A SPARQL value constraint is defined recursively as follows:*

1. *If $?X, ?Y \in V$ and $u \in I \cup L$, then $?X = u, ?X = ?Y$, $bound(?X)$, $isIRI(?X)$, $isLiteral(?X)$, and $isBlank(?X)$ are atomic value constraints.*
2. *If R_1 and R_2 are value constraints then $\neg R_1$, $R_1 \wedge R_2$, and $R_1 \vee R_2$ are value constraints.*

Note 3. In these definitions, value constraint do not contain black nodes.

Definition 23 (Built-in condition). *A SPARQL* built-in condition *is constructed using elements of the set $I \cup L \cup V$ and constants, logical connectives $(\neg, \wedge, \vee)$, inequality symbols $(<, \leqslant, \geqslant, >, !)$, and the equality symbol $(=)$. unary predicates like bound, isBlank, and isIRI, plus other features. We define inductively a* built-in condition *as follows:*

1. *If $?X, ?Y \in V$, $u \in I \cup L$, and* op $\in \{=, <, \leqslant, \geqslant, >, !\}$ *then $?X$ op $u, ?X$ op $?Y$, are atomic value constraints. $bound(?X)$, $isIRI(?X)$, $isLiteral(?X)$, and $isBlank(?X)$*
2. *If R_1 and R_2 are value constraints then $\neg R_1$, $R_1 \wedge R_2$, and $R_1 \vee R_2$ are value constraints.*

Definition 24 (Graph pattern expression). *A SPARQL graph pattern expression is defined recursively as follows:*

1. *A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a graph pattern (a* triple pattern*).*
2. *A basic graph pattern is a graph pattern.*
3. *If P_1 and P_2 are graph patterns, then expressions (P_1 AND P_2), (P_1 UNION P_2), and (P_1 OPT P_2) are graph patterns (*conjunction graph pattern, optional graph pattern, *and* union graph pattern, *respectively).*
4. *If P is a graph pattern and R is a SPARQL built-in condition, then the expression (P FILTER R) is a graph pattern (a filter graph pattern).*

Definition 25 (Graph Pattern Evaluation). *Let $\mathcal{D}$ be a RDF dataset and P be a graph pattern, the* evaluation of P over $\mathcal{D}$, *denoted by $[\![P]\!]_{\mathcal{D}}$, is defined recursively as follows:*

1. *If P is a triple pattern t, then $[\![P]\!]_{\mathcal{D}} = \{\mu \mid dom(\mu) = var(t)$ and $\mu(t) \in \mathcal{D}\}$.*
2. *If P is (P_1 AND P_2), then $[\![P]\!]_{\mathcal{D}} = [\![P_1]\!]_{\mathcal{D}} \bowtie [\![P_2]\!]_{\mathcal{D}}$*
3. *If P is (P_1 OPT P_2), then $[\![P]\!]_{\mathcal{D}} = [\![P_1]\!]_{\mathcal{D}} ⟕ [\![P_2]\!]_{\mathcal{D}}$*
4. *If P is (P_1 UNION P_2), then $[\![P]\!]_{\mathcal{D}} = [\![P_1]\!]_{\mathcal{D}} \cup [\![P_2]\!]_{\mathcal{D}}$*

Definition 26 (SELECT Result Form). *A SELECT query is a tuple (W, P), where $W \subseteq V$ is a finite set of variables and P is a graph pattern. The answer of (W,P) in a dataset $\mathcal{D}$ is the set of mappings $\{\mu_{|_W} \mid \mu \in [\![P]\!]_{\mathcal{D}}\}$.*

Appendix B. PACT Programming Model definition

In the following, we describe the five PACT operators in an analogous way as it is done by [14].

Simple-Input Contract. The input of a Single-Input Contract is a set of key/value pairs $S = \{s\}$. For a key/value pair $s = (k, v)$ the function $key(s)$ returns the key k. A Single-Input Contract is defined as a mapping function m that assigns each $s \in S$ to one or more parallelization units (PU) u_i. A PU u_i is a set of key/-value pairs for which holds $u_i \subset S$. The set of all PU is denoted with $U = \{u_i\}$.

Definition 27 (mapping function). *The mapping function m is defined as:*

$$m : s \to \{i \mid i \in \mathbb{N} \wedge 1 \leqslant i \leqslant |U|\},$$

generate all $u_i \in U$ such that:

$$u_i = \{s \mid s \in S \wedge i \in m(s)\}$$

Definition 28 (map). *The map contract is used to independently process each key/value pair; consequently, the UDF is called independently for each element of the input*

$$|m(s)| = 1 \wedge m(s_i) \neq m(s_j), \textit{for } i \neq j$$

Definition 29 (reduce). *The reduce contract partitions key/value pairs by their keys. The definition assures that all pairs with the same key are grouped and handed together*

$$|m(s)| = 1 \wedge (m(s_i) = m(s_j)) \Leftrightarrow (key(s_i) = key(s_j))$$

Multi-Input Contract. A Multi-Input Contract is defined over n sets of key/value pairs $S_i = \{s_i\}$, *for* $i \leqslant i \leqslant n$. The function

$$m_i : s_i \to \{j \mid j \in \mathbb{N} \wedge 1 \leqslant j \leqslant |P_i|\},$$

All $p_{(i,j)} \in P_i$ are generated as:

$$P_{(i,j)} = \{s_i \mid s_i \in S_i \wedge j \in m_i(s_i)\}$$

Given a set of subsets P_i for each input set S_i the set of parallelization units U is generated as follows:

$$U = \{u \mid a(u) = true, u \in (P_1 \times P_2 \times \cdots \times P_n)\}$$

where $a(.)$ is an association function that decides whether a parallelization unit u is valid

$$a : u \rightarrow Boolean$$

Definition 30 (cross). *The cross contract is defined as the Cartesian Product over its input sets. The UDF is executed for each element of the Cartesian Product.*

$$m \textit{ for each input like map}$$
$$a(u) = true$$

Definition 31 (cogroup). *The cogroup contract partitions the key/value pairs of all input sets according to their keys. For each input, all pairs with the same key form one*

$$m \textit{ for each input like reduce}$$
$$a(u) = true \textit{ if } key_1(s_1) = \cdots = key_m(s_m)$$
$$\textit{for } u = \{p_1, \ldots, p_m\} \wedge \forall s_1 \in p_1, \ldots, \forall s_m \in p_m$$

Definition 32 (match). *The match contract is a relaxed version of the CoGroup contract. Similar to the Map contract, it maps for all input set each key/value pair into a*

$$m \textit{ for each input like map}$$
$$a \textit{ identical to cogroup}$$

Appendix. References

[1] G. Klyne and J.J. Carroll, Resource Description Framework (RDF): Concepts and Abstract Syntax, 2004, W3C Recommendation. [Online; accessed November 21, 2017]. https://www.w3.org/TR/rdf-sparql-query/.

[2] H. Choi, J. Son, Y. Cho, M.K. Sung and Y.D. Chung, SPIDER: A System for Scalable, Parallel / Distributed Evaluation of Large-scale RDF Data, in: *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, ACM, New York, NY, USA, 2009, pp. 2087–2088. ISBN 978-1-60558-512-3. doi:10.1145/1645953.1646315. http://doi.acm.org/10.1145/1645953.1646315.

[3] S. Harris, N. Lamb, N. Shadbolt and G. Ltd, 4store: The design and implementation of a clustered RDF store (2009), 94–109.

[4] Z. Kaoudi and I. Manolescu, RDF in the Clouds: A Survey, *The VLDB Journal* **24**(1) (2015), 67–91, ISSN 1066-8888. doi:10.1007/s00778-014-0364-z. http://dx.doi.org/10.1007/s00778-014-0364-z.

[5] P. Peng, L. Zou, M.T. Özsu, L. Chen and D. Zhao, Processing SPARQL Queries over Distributed RDF Graphs, *The VLDB Journal* **25**(2) (2016), 243–268, ISSN 1066-8888. doi:10.1007/s00778-015-0415-0. http://dx.doi.org/10.1007/s00778-015-0415-0.

[6] P. Khodke, S. Lawange, A. Bhagat, K. Dongre and C. Ingole, Query Processing over Large RDF Using SPARQL in Big Data, in: *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, ICTCS '16, ACM, New York, NY, USA, 2016, pp. 66–1666. ISBN 978-1-4503-3962-9. doi:10.1145/2905055.2905124. http://doi.acm.org/10.1145/2905055.2905124.

[7] A. Hasan, M. Hammoud, R. Nouri and S. Sakr, DREAM in Action: A Distributed and Adaptive RDF System on the Cloud, in: *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 2016, pp. 191–194. ISBN 978-1-4503-4144-8. doi:10.1145/2872518.2901923. https://doi.org/10.1145/2872518.2901923.

[8] J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, *Commun. ACM* **51**(1) (2008), 107–113, ISSN 0001-0782. doi:10.1145/1327452.1327492. http://doi.acm.org/10.1145/1327452.1327492.

[9] A.S. Fundation, The Apache Hadoop, 2008, Apache Software Fundation. [Online; accessed November 21, 2017]. http://hadoop.apache.org.

[10] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker and I. Stoica, Apache Spark: A Unified Engine for Big Data Processing, *Commun. ACM* **59**(11) (2016), 56–65, ISSN 0001-0782. doi:10.1145/2934664. http://doi.acm.org/10.1145/2934664.

[11] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi and K. Tzoumas, Apache Flink: Stream and Batch Processing in a Single Engine, *IEEE Data Eng. Bull.* **38**(4) (2015), 28–38. doi:http://sites.computer.org/debull/A15dec/p28.pdf.

[12] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R.J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt and S. Whittle, The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing, *Proc. VLDB Endow.* **8**(12) (2015), 1792–1803, ISSN 2150-8097. doi:10.14778/2824032.2824076. http://dx.doi.org/10.14778/2824032.2824076.

[13] A. Alexandrov, S. Ewen, M. Heimel, F. Hueske, O. Kao, V. Markl, E. Nijkamp and D. Warneke, MapReduce and PACT - Comparing Data Parallel Programming Models (2011), 25–44.

[14] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl and D. Warneke, Nephele/PACTs: A Programming Model and Execution Framework for Web-scale Analytical Processing, in: *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, ACM, New York, NY, USA, 2010, pp. 119–130. ISBN 978-1-4503-0036-0. doi:10.1145/1807128.1807148. http://doi.acm.org/10.1145/1807128.1807148.

[15] C. Bizer and A. Schultz, The Berlin SPARQL Benchmark, *Int. J. Semantic Web Inf. Syst.* **5** (2009), 1–24.

[16] J. Pérez, M. Arenas and C. Gutierrez, Semantics and Complexity of SPARQL, *ACM Trans. Database Syst.* **34**(3) (2009), 16–11645, ISSN 0362-5915. doi:10.1145/1567274.1567278. http://doi.acm.org/10.1145/1567274.1567278.

[17] E. Prud'hommeaux and A. Seaborne, SPARQL Query Language for RDF, 2008, W3C Recommendation. [Online; accessed November 21, 2017]. https://www.w3.org/TR/rdf-sparql-query/.

[18] J. Pérez, M. Arenas and C. Gutierrez, Semantic of SPARQL, Technical Report, TR/DCC-2006-17, Department of Computer Science, Universidad de Chile, Santiago de Chile, Chile, 2006.

[19] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M.J. Sax, S. Schelter, M. Höger, K. Tzoumas and D. Warneke, The Stratosphere Platform for Big Data Analytics, *The VLDB Journal* **23**(6) (2014), 939–964, ISSN 1066-8888. doi:10.1007/s00778-014-0357-y. http://dx.doi.org/10.1007/s00778-014-0357-y.

[20] A.S. Fundation, The Apache Calcite, 2014, Apache Software Fundation. [Online; accessed November 21, 2017]. https://calcite.apache.org.

[21] D. Warneke and O. Kao, Nephele: Efficient Parallel Data Processing in the Cloud, in: *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, ACM, New York, NY, USA, 2009, pp. 8–1810. ISBN 978-1-60558-714-1. doi:10.1145/1646468.1646476. http://doi.acm.org/10.1145/1646468.1646476.

[22] K. Tzoumas, J.-C. Freytag, V. Markl, F. Hueske, M. Peters, M. Ringwald and A. Krettek, Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs, in: *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 1292–1295. ISBN 978-1-4673-4909-3. doi:10.1109/ICDE.2013.6544927. http://dx.doi.org/10.1109/ICDE.2013.6544927.

[23] F. Hueske, M. Peters, M.J. Sax, A. Rheinländer, R. Bergmann, A. Krettek and K. Tzoumas, Opening the Black Boxes in Data Flow Optimization, *Proc. VLDB Endow.* **5**(11) (2012), 1256–1267, ISSN 2150-8097. doi:10.14778/2350229.2350244. http://dx.doi.org/10.14778/2350229.2350244.

[24] A. Jena, SPARQL Syntax Expression, 2011, Apache Software Fundation. [Online; accessed November 21, 2017]. https://jena.apache.org/documentation/notes/sse.html.

[25] Y. Guo, Z. Pan and J. Heflin, LUBM: A Benchmark for OWL Knowledge Base Systems, *Web Semant.* **3**(2–3) (2005), 158–182, ISSN 1570-8268. doi:10.1016/j.websem.2005.06.005. http://dx.doi.org/10.1016/j.websem.2005.06.005.

[26] G. Aluç, O. Hartig, M.T. Özsu and K. Daudjee, Diversified Stress Testing of RDF Data Management Systems, in: *Proceedings of the 13th International Semantic Web Conference - Part I*, ISWC '14, Springer-Verlag New York, Inc., New York, NY, USA, 2014, pp. 197–212. ISBN 978-3-319-11963-2. doi:10.1007/978-3-319-11964-9_13. http://dx.doi.org/10.1007/978-3-319-11964-9_13.

[27] M. Schmidt, T. Hornung, M. Meier, C. Pinkel and G. Lausen, SP2Bench: A SPARQL Performance Benchmark, in: *Semantic Web Information Management: A Model-Based Perspective*, R. de Virgilio, F. Giunchiglia and L. Tanca, eds, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 371–393. ISBN 978-3-642-04329-1. doi:10.1007/978-3-642-04329-1_-16. https://doi.org/10.1007/978-3-642-04329-1_16.

[28] M. Morsey, J. Lehmann, S. Auer and A.-C.N. Ngomo, DBpedia SPARQL Benchmark: Performance Assessment with Real Queries on Real Data, in: *Proceedings of the 10th International Conference on The Semantic Web - Volume Part I*, ISWC'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 454–469. ISBN 978-3-642-25072-9. http://dl.acm.org/citation.cfm?id=2063016.2063046.

[29] S. Qiao and Z.M. Özsoyoğlu, RBench: Application-Specific RDF Benchmarking, in: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, ACM, New York, NY, USA, 2015, pp. 1825–1838. ISBN 978-1-4503-2758-9. doi:10.1145/2723372.2746479. http://doi.acm.org/10.1145/2723372.2746479.

[30] M.T. Özsu, A Survey of RDF Data Management Systems, *Front. Comput. Sci.* **10**(3) (2016), 418–432, ISSN 2095-2228. doi:10.1007/s11704-016-5554-y. http://dx.doi.org/10.1007/s11704-016-5554-y.

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker and I. Stoica, Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing, in: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, USENIX Association, Berkeley, CA, USA, 2012, pp. 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301.

[32] M. Armbrust, R.S. Xin, C. Lian, Y. Huai, D. Liu, J.K. Bradley, X. Meng, T. Kaftan, M.J. Franklin, A. Ghodsi and M. Zaharia, Spark SQL: Relational Data Processing in Spark, in: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, ACM, New York, NY, USA, 2015, pp. 1383–1394. ISBN 978-1-4503-2758-9. doi:10.1145/2723372.2742797. http://doi.acm.org/10.1145/2723372.2742797.

[33] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic and G. Lausen, S2RDF: RDF Querying with SPARQL on Spark, *Proc. VLDB Endow.* **9**(10) (2016), 804–815, ISSN 2150-8097. doi:10.14778/2977797.2977806. http://dx.doi.org/10.14778/2977797.2977806.

[34] D.J. Abadi, A. Marcus, S.R. Madden and K. Hollenbach, Scalable Semantic Web Data Management Using Vertical Partitioning, in: *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, VLDB Endowment, 2007, pp. 411–422. ISBN 978-1-59593-649-3. http://dl.acm.org/citation.cfm?id=1325851.1325900.

[35] P. Valduriez, Join Indices, *ACM Trans. Database Syst.* **12**(2) (1987), 218–246, ISSN 0362-5915. doi:10.1145/22952.22955. http://doi.acm.org/10.1145/22952.22955.

[36] H. Naacke, B. Amann and O. Curé, SPARQL Graph Pattern Processing with Apache Spark, in: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, GRADES'17, ACM, New York, NY, USA, 2017, pp. 1–117. ISBN 978-1-4503-5038-9. doi:10.1145/3078447.3078448. http://doi.acm.org/10.1145/3078447.3078448.

[37] C. Chambers, A. Raniwala, F. Perry, S. Adams, R.R. Henry, R. Bradshaw and N. Weizenbaum, FlumeJava: Easy, Efficient Data-parallel Pipelines, *SIGPLAN Not.* **45**(6) (2010), 363–375, ISSN 0362-1340. doi:10.1145/1809028.1806638. http://doi.acm.org/10.1145/1809028.1806638.

[38] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom and S. Whittle, MillWheel: Fault-tolerant Stream Processing at Internet Scale, *Proc. VLDB Endow.* **6**(11) (2013), 1033–1044, ISSN 2150-8097. doi:10.14778/2536222.2536229. http://dx.doi.org/10.14778/2536222.2536229.

[39] A.S. Fundation, The Apache Hadoop, 2017, Apache Software Fundation. [Online; accessed November 21, 2017]. https://beam.apache.org.

[40] D. Dell'Aglio, E. Della Valle, J.-P. Calbimonte and O. Corcho, RSP-QL Semantics: A Unifying Query Model to Explain Heterogeneity of RDF Stream Processing Systems, *Int. J. Semant. Web Inf. Syst.* **10**(4) (2014), 17–44, ISSN 1552-6283. doi:10.4018/ijswis.2014100102. http://dx.doi.org/10.4018/ijswis.2014100102.