Towards Making Distributed RDF Processing FLINKer

Amr Azzam*, Sabrina Kirrane*, Axel Polleres*†
*Vienna University of Economics and Business, Vienna, Austria
{amr.azzam, sabrina.kirrane, axel.polleres}@wu.ac.at
†Complexity Science Hub Vienna, Vienna, Austria

***Abstract*—In the last decade, the Resource Description Framework (RDF) has become the de-facto standard for publishing semantic data on the Web. This steady adoption has led to a significant increase in the number and volume of available RDF datasets, exceeding the capabilities of traditional RDF stores. This scenario has introduced severe big semantic data challenges when it comes to managing and querying RDF data at Web scale. Despite the existence of various off-the-shelf Big Data platforms, processing RDF in a distributed environment remains a significant challenge. In this position paper, based on an in-depth analysis of the state of the art, we propose to manage large RDF datasets in Flink, a well-known scalable distributed Big Data processing framework. Our approach, which we refer to as FLINKer extends the native graph abstraction of Flink, called Gelly, with RDF graph and SPARQL query processing capabilities.**

***Index Terms*—RDF, SPARQL, Flink, Big Semantic Data**

I. Introduction

The *Linked Open Data (LOD) cloud* [7] is a collective, open effort to interconnect data scattered on the Web into a standard format that can be leveraged by diverse applications. This interlinked graph data is based on RDF [35], a common graph-based model, used to describe and link data at various degrees of granularity. Nowadays, RDF is the de-facto standard for representing heterogeneous knowledge on the Web, organized around the emerging notion of knowledge graphs [39].

The latest edition of DBpedia (2016-10), a partial conversion of Wikipedia to RDF, consists of more than 13 billion triples (i.e., RDF statements). While LOD-a-lot [14], a dataset integrating a partial crawl of the LOD cloud, includes more than 28 billion triples from heterogeneous sources. It is not surprising that Linked Data suffers from scalability challenges when it comes to storing, indexing, processing and querying, large collections of RDF triples [30]—typically through the SPARQL [19] query language.

In recent years several approaches to efficiently process and query RDF for such big semantic data scenarios have been proposed. Besides efficient, compressed representations [15], scalable RDF stores [8] and distributed indexes [37], recent trends seek to leverage the potential of existing big data tools, such as Apache Spark[1] [41] or Apache Accumulo [2], and distributed computation, with a particular focus on vertex-centric query resolution [4], [5], [31].

Despite initial efforts towards efficient distributed SPARQL query processing, there are still several open research questions. Empirical evaluations [3], [13], [31] show that the noticeable differences in performance among existing query engines are heavily dependent on: (i) the skewed structured of RDF graphs, which can slow down graph processing, (ii) the nature of the SPARQL queries, as some engines lack support for certain SPARQL operators, (iii) the graph partitioning strategy, as some approaches are only optimized to process the full graph (which is rarely the case in selective SPARQL queries) and; (iv) the non-optimized query plans, which can generate multiple inefficient iterations.

In a step towards addressing these open challenges, in this position paper we present *FLINKer*[2], a proposal to use Flink[3] [9], a Big Data platform, to process RDF data and resolve SPARQL queries in a distributed and scalable manner.

Flink provides an open-source stream processing framework for distributed and high-performing data processing that can fit the high demands of large, ever-growing RDF datasets. In addition, Flink provides a graph processing library called Gelly, which we propose as a basis to represent and process RDF graphs. Finally, we leverage the optimized iterative graph processing capabilities of Flink to serve and resolve full SPARQL queries at web scale. We expect that the combination of Flink, RDF, and SPARQL will on the one hand enable semantic practitioners to perform semantic processing and novel analysis over large RDF datasets, and on the other hand enable Flink practitioners to leverage the very expressive SPARQL language to process and analyze large knowledge graphs.

The remainder of the paper is organized as follows. Section II provides background information on RDF, SPARQL, Flink and vertex-centric processing. In Section III we review the current state of the art in RDF and SPARQL processing, with a particular focus on existing Big Data frameworks. Our FLINKer proposal is detailed in Section IV, where we provide our initial insights on representing RDF in Gelly and using Flinks capabilities to efficiently resolve SPARQL queries. Finally, we discuss the approach in Section V and conclude and provide future work in Section VI.

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

[2]In German, *FLINKer* stands for "nimble, speedy, agile".

[3]http://flink.apache.org/

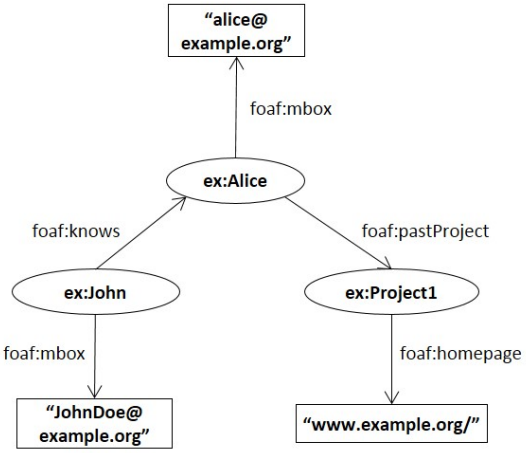

Fig. 1: An example of an RDF graph.

II. Background

In this section, we provide the necessary background information on the RDF data model and the standard query language SPARQL, the vertex-centric distributed processing paradigm and the Flink stream processing framework.

A. RDF and SPARQL

The RDF data model, is typically formalized as follows (cf. see [18]): Assume infinite, mutually disjoint sets I (RDF IRIs references), B (Blank nodes), and L (RDF literals).

Definition 1 (RDF triple): A tuple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an RDF triple, in which s is the subject, p the predicate and o the object.

Definition 2 (RDF graph): An RDF graph G is a set of RDF triples. Thus, (s, p, o) can be represented as a direct edge-labeled graph $s \xrightarrow{p} o$.

RDF graphs can be grouped and managed together as an *RDF dataset* [35], i.e. a collection of RDF graphs.

Figure 1 represents an RDF graph with five triples representing two persons, John and Alice, their emails and Alice's project. RDF is typically queried trough the standard SPARQL [19] query language. SPARQL is an expressive declarative language based on graph-pattern matching with a SQL-like syntax. For instance, the query in Listing 1 retrieves the homepages of the projects of those people known by John.

Listing 1: Example of a SPARQL query

```
PREFIX ex: <http://example.org>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?projectHomePage
WHERE {
    ex:John foaf:known ?people .
    ?people foaf:pastProject ?project .
    ?project foaf:homepage ?projectHomePage .
}
```

SPARQL is based on graph pattern matching, where the core component is the concept of a triple pattern, *i.e.*, triples where subjects, predicates and objects may be variables. This is formalized in Definition 3, assuming a set V of variables that are disjoint from the aforementioned I, B and L.

Definition 3 (SPARQL triple pattern): A tuple from $(I \cup B \cup V) \times (I \cup V) \times (I \cup B \cup L \cup V)$ is a triple pattern.

The previous example showed three triple patterns, called a Basic Graph Pattern (BGP). In general terms, BGPs are sets of triple patterns in which all triple patterns must match. They can be seen as inner-joins in SQL.

Definition 4 (SPARQL Basic Graph pattern (BGP)): A SPARQL Basic Graph Pattern (BGP) is defined as a set of triple patterns. SPARQL FILTERs can restrict a BGP. If B_1 is a BGP and R is a SPARQL *built-in* condition, then (B_1 `FILTER` R) is also a BGP.

Finally, it is worth mentioning that several constructions can be applied over BGPs, such as grouping, alternative groups with the `UNION` keyword, optional graph patterns with an `OPTIONAL` keyword and restrictions by means of a `FILTER` clause.

B. Vertex-Centric Model

Interest in distributed SPARQL query processing has increased in recent years. In the following, we briefly describe the vertex-centric model, one of the most prominent paradigms for distributed processing.

The vertex-centric model, also known as "Pregel" or "think like a vertex" [27] is a computational model proposed by Google and inspired by the Bulk Synchronous Parallel (BSP) model [38] to address the challenge of the efficient processing of large scale graphs with billions of vertices. This computation model is highly adaptable, while at the same time providing guarantees with respect to performance, scalability and fault-tolerance.

The computation in Pregel can be modelled as a directed graph, where each processing entity represents a vertex of the graph that can send messages to its connected neighbours. Pregel based programs are based on a sequence of iterations, called supersteps. In each superstep the framework calls a user-defined function that determines the behavior at a single vertex in the iteration.

The framework introduces a message passing model for communication between vertices during the iterations. Each vertex can read the messages sent from the previous superstep and modify the current state of the vertex as well as the outgoing edges. The framework subsequently sends messages to the other vertices so that the updated values can be used in the next superstep.

C. The Flink architecture

Apache Flink [9] is an open source distributed dataflow processing framework that supports a single environment for batch and stream processing. When it comes to parallelization, Flink supports a generalization of the MapReduce programming model commonly known as the parallelization contract or PACT programming model.

1) Flink Overview: Flink affords large scale data applications, high throughput, low latency and fault tolerant processing. Flink is a true stream processing engine. Thus, batch and stream processing are supported on the basis of two types of

streams: unbounded streams, which are used to develop stream applications, and bounded streams, used for batch processing.

Flink offers different level of abstractions to develop stream and batch processing applications, as shown in the Figure 2. The lowest abstraction level provided by Flink is called ***Stateful Stream Processing***, which is then encapsulated in a higher abstraction level that provides the *core APIs* of Flink, ***DataStream*** and ***DataSet***. Both can be seen as immutable, distributed data collections, with a finite (*DataSet*) or unbounded (*DataStream*) number of elements. These core APIs provide data transformation operators, such as filter, map, join, grouping and aggregation, that require user-defined functions (UDFs) as arguments in order to construct a new data collection from either a data source or another data collection. In addition, Flink provides a declarative domain-specific language through the ***Table API***, which offers a relational-like model abstraction. Finally, the highest level of abstraction proposed by Flink is the ***SQL*** library, which allows users to implement Flink programs as SQL query expressions. Finally, our ***FLINKer*** approach to support RDF processing via Flink, based on the *Gelly* library, is represented at the same level of abstraction as the *SQL* library. The specifics of the *FLINKer* approach will be presented in Section IV.

The Flink dataflow program typically starts by constructing data collections such as *DataSet* or *DataStream* from different data sources, e.g HDFS files, Kafka topics[4], etc. Then, a chain of data transformations are applied on these distributed collections. Finally, the results are retrieved via data sinks that, for example, write the results to distributed files.

Flink provides support for the data distribution, process optimization and parallel processing over clusters of machines. Furthermore, several libraries for diverse analytical tasks are implemented on top of Apache Flink. For instance, *FlinkCEP* and *FlinkML* are two well-known libraries to deal with endless complex event processing and scalable machine learning algorithms, respectively. Moreover, Flink introduced the *Gelly* library, which contains a set of methods for graph analysis as well as graph algorithms that support graph processing.

2) Flink and the PACT Model: The parallelization contract programming model (PACT) [6] was proposed to generalize two MapReduce concepts [11], namely second-order functions and contracts. The main purpose of PACT is to simplify the process of parallel Web-scale dataflow processing. A PACT operator receives one or more input datasets that contain a predefined data type in addition to a user-defined function that will be applied to the input dataset on several cluster nodes. Then, the PACT operator will produce one or more outputs.

The PACT model consists of two main components, the *input contract* and the *output contract*. The input contract defines how the user-defined functions will be evaluated in parallel, i.e., the contract determines how the input data will be divided into disjoint fragments that can be processed independently in a parallel dataflow configuration. In turn, the output contract provides the PACT compiler with additional information that allows the optimizer to generate more efficient program execution strategies.

In practice, a PACT program can be represented as a sequence of PACT operators where the output of one operator can be consumed as an input of other operators. Thus, the PACT compiler first transforms the dataflow into a Directed Acyclic Graph (DAG) of operators. Then, the DAG is used to generated multiple execution strategies. The most efficient strategy is selected based on a cost function that estimates the least amount of data movements between clusters nodes [21].

The Apache Flink framework currently provides 22 data transformation operators[5] (e.g. join, reduce, filter, aggregate, union, etc.), which follow the PACT programming model. In addition, Flink uses a PACT-based optimizer in order to produce an optimized Flink execution plan, following the same procedure mentioned previously. It is worth mentioning that the Flink optimizer computes the cost function based on valuable resources utilization such as data shipping over the cluster network, memory utilization and hard disk I/O costs

III. State of the Art

Generally speaking, RDF triple stores support SPARQL queries based on: i) a relational schema; ii) a native index; or iii) a NOSQL solution. As an example of the former case, the relational-based Virtuoso [8] represents RDF data in a column-based store, with two full indexes over the RDF datasets and 3 projections to support and speed up SPARQL queries. As for native indexes, RDF stores often speed up queries by indexing different combinations of the subject, predicate and object elements in RDF [20], [24]. The well-known Apache Jena TDB[6] stores RDF datasets using 6 B+Trees indexes in order SPOG, POSG, OSPG, GSPO, GPOS and GOSP, where S, P and O represent the subject, predicate and object respectively, and G represent the graph in which the triple holds. A recent approach, RDF-4X [2] implements a cloud-based NOSQL solution using Apache Accumulo[7]. Blazegraph[8] (formerly BigData) follows a similar NOSQL using six indexes.

A. Big Data tools based RDF Engines

The increasing availability of new RDF datasets has led to a need for distributed RDF engines in order to deal with the impracticability of executing complex queries using centralized engines. Over the years, several distributed RDF processing engines (cf. the Hadoop-based query engine proposed by [33] or the Spark-based query engine presented in [36]), have been proposed in order to bridge the gap between data volumes and performance demands. Although distributed RDF systems provide access to more system resources such as memory sizes and greater processing capacity, they face the problem of intermediate data shipping between clusters nodes. In the following, we provide an overview of existing distributed RDF

[4]https://kafka.apache.org/

[5]See https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/stream/operators/index.html#datastream-transformations.

[6]http://jena.apache.org/documentation/tdb/index.html

[7]ttps://accumulo.apace.org/

[8]https://www.blazegraph.com/

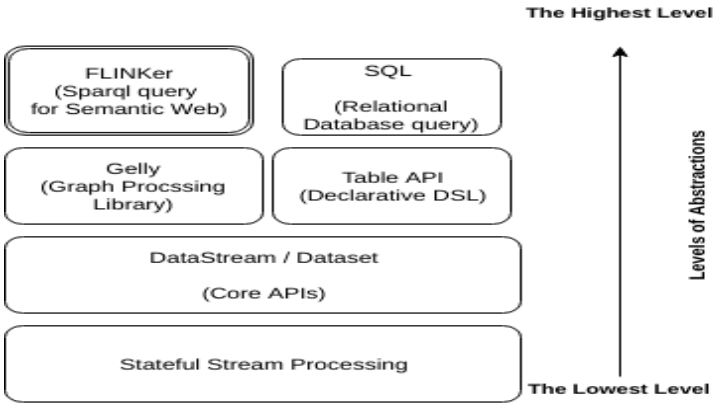

Fig. 2: FLINKer Abstraction Level Representation.

processing approaches that aim to enable RDF querying with SPARQL via big data tools.

1) Hadoop based RDF processing engines: SHARD [29] is a hadoop-based triple store that makes use of MapReduce jobs in order to evaluate SPARQL queries on RDF datasets. In SHARD, the entire dataset is stored in a single file on HDFS (i.e. without any partitioning scheme), where each line groups all triples corresponding to one subject. SHARD follows an approach called *Clause Iteration* for query processing, which performs a sequence of MapReduce iterations. Each iteration is responsible for executing a subquery clause, while the result of every subquery is incrementally joined with the following iteration. The final iteration removes duplicates and filters bounded variables according to what was requested in the SELECT query.

HadoopRDF [12] supports SPARQL queries on top of Hadoop by leveraging the HDFS file partitioning and distribution capabilities. In contrast to SHARD, HadoopRDF splits the data into multiple files based on the predicate of the RDF triples, which is traditionally referred to as *Vertical Partitioning* (VP) [1]. After the partitioning phase, HadoopRDF creates a query plan that minimizes the MapReduce tasks as well as the intermediate results.

CliqueSquare[9] [16] focuses on minimizing the number of MapReduce jobs and the data movements between the server nodes. To achieve the aforementioned goals, the CliqueSquare engine exploits the default replication mechanism in HDFS and. Additionally, CliqueSquare partitions triples based on hashing the subjects, predicates and objects. Moreover, another partitioning is performed on each node by grouping the triples that have the same predicate in one file.

H2RDF+ [28] is a distributed RDF data store based on MapReduce processing and Hbase[10] indexes. H2RDF+ utilizes Hbase in order to create six indexes on the RDF triples, so that all triple patterns can be answered in a single table scan. H2RDF+ employs merge joins along with a greedy algorithm to determine the least costly joins.

Finally note that several distributed SPARQL query frameworks translate SPARQL queries to Hadoop SQL-Like queries, such as PigSPARQL [32] based on Apache Pig[11], and Apache Sempala [33] that relies on Apache Impala[12].

2) Spark based RDF processing engines: S2RDF [34] is a Spark-based SPARQL query processing engine that is based on translating SPARQL queries to SparkSQL. In addition, S2RDF introduces ExtVP, a novel partitioning strategy that extends the aforementioned idea of vertical partitioning. ExtVP precomputes a semi-join reduction of vertical partitioning tables. This reduction produces semi-join tables that are smaller than the base table. Therefore, ExtVP reduces the unnecessary input/output operations and avoids excessive memory consumption during join operations.

S2X [31] is a SPARQL query engine that makes intensive use of GraphX, the graph-parallel library of Spark, as well as data-parallel operators provided by Spark. S2X defines a property graph representation of RDF for Graphx, and applies a vertex-centric model for BGP matching.

In turn, the SPARKRDF [40] engine partitions the RDF graph into multi-layer elastic subgraphs, creating five indexes with different granularities in order to speed up SPARQL query resolution. Note that, in order to support faster joins, all intermediate results reside in memory. The query plan of SPARKRDF is generated based on a greedy algorithm that aims to avoid data shuffling and the reduction of intermediate results.

SPARQLGX [17] executes SPARQL queries over Spark by translating SPARQL queries into Scala code. SPARQLGX also uses vertical partitioning based on the predicates of the RDF triples, and computes statistics to execute plans with less intermediate results.

Finally, SANSA [25] is an initial effort to leverage the currently existing big data frameworks such as Apache Spark

[9]https://team.inria.fr/oak/projects/cliquesquare/
[10]https://hbase.apache.org/
[11]https://pig.apache.org/
[12]https://impala.apache.org/

Flink Operator	Description
Map	**Map** transformation applies a user-defined map function to each element of the input DataSet. Map insures that the function takes one element as an input and produces one and only one element, so that map transformation assures one-to-one relation between input and output DataSets.
FlatMap	**FlatMap** transformation takes a user-defined function as an argument and applies the function to each element of the input DataSet. When the map function is applied to each element of the stream, it produces a stream of new elements so the output could be zero, one or more elements.
Filter	**Filter** transformation takes a user-defined predicate that allows a predefined filtering criteria to be applied to each element in the input DataSet, so that it could decide which items should be kept or eliminated from the output DataSet.
Project	**Project** transformation enables selecting a subset of DataSet fields from user-defined tuples so that the output DataSet returns the selected subset.
FlatJoin	**FlatJoin** transformation joins two DataSets by generating all pairs of elements that hold the same values for the specified keys. The Flink join operator provides the option to execute a JoinFunction which produces exactly one output element or a FlatJoinFunction which turns pairs of elements into zero or more elements.
ReduceGroup	**ReduceGroup** groups DataSet elements based on a specific key. On each group of elements, a user-defined function is applied thereby it reduces the count of output elements.
Iteration Operators	Flink provides two types of iterations: BulkIteration and DeltaIteration. Iteration Operators are useful for implementing loops by performing part of the program repeatedly and then forwarding the results to the next iteration. • **BulkIteration**: a normal loop which iterates over the entire input whether it is the result from the earlier iteration or the initial DataSet; passes the result as a partial solution; the loop terminates by reaching the maximum number of iterations. • **DeltaIteration**: supports incremental loops that modify a subset of the elements in the solution rather than recomputing it.

TABLE I: Subset of Flink operators used in FLINKer

and Apache Flink to manipulate, store and analyze RDF data. The proposed framework architecture aims at reading/writing RDF to HDFS, supporting SPARQL, RDF inference, OWL and machine learning algorithms on top of RDF data. In spite of the initial engine based on converting SPARQL to SQL, the evaluation of the SPARQL engine at large scale is still pending.

To the best of our knowledge, our work is the first approach proposing to resolve SPARQL using a vertex-centric approach on top of the Flink/Gelly graph-processing infrastructure. A similar paradigm has been recently followed [22] to implement the well-known Neo4J *Cypher* query language[13] on top of Flink and Gelly.

IV. SPARQL query processing in Flink

Big data tools such as Spark and Flink are distributed data-parallel frameworks that are designed to support efficient processing of data-centric algorithms by taking care of the parallelization aspects, thus taking this burden off the programmers. Recently, these frameworks provide graph processing APIs (such as GraphX in Spark and Gelly in Flink) so that graph processing systems could benefit from the scalable processing infrastructures. In addition, unlike MapReduce frameworks, the distributed dataflow systems present a wide range of operators which can be executed efficiently in main memory.

In the following, we present FLINKer, our proposal to manage large RDF datasets and resolve SPARQL queries on top of Flink/Gelly. First, we provide an overall description of our architecture. Then, we introduce a suitable representation for RDF graphs in Gelly. Finally, we outline how Flink and the distributed dataflow operators can be used to support a vertex-centric resolution of SPARQL.

[13]See https://neo4j.com/developer/cypher-query-language/

A. FLINKer Overview

FLINKer is an RDF graph processing system on top of the distributed Apache Flink framework. FLINKer combines the data-parallel operators provided by the Flink core APIs and the graph-parallel abstraction offered by the Gelly library (shown in Figure 2), in order to provide a SPARQL query engine compatible with the existing Flink ecosystem. That is, FLINKer is aimed at positioning SPARQL as the candidate high level query language for graphs in Flink, complementing the existing Flink SQL libraries (and other extensions such as the aforementioned support for the Cypher query language [22]).

In practice, FLINKer makes use of Gelly to provide the vertex-centric view on graph processing, and the DatasSet API operators to support each of the transformations required to resolve SPARQL queries. Table I shows a subset of Flink DataSet transformations that are used for implementing SPARQL query operators. For example, the join operator is performed using a combination of Flink operators such as Filter transformation and Iteration operators.

In the following, we will concisely discuss the proposed components of the FLINKer engine, which is depicted in Figure 3.

B. RDF Graph Loader in Flink

The FLINKer RDF graph loader is the component responsible for ingesting RDF data into Flink. Thus, we read triples in the RDF N-Triples format (i.e. one triple per line) and convert them into the corresponding Gelly graph representation that is suitable to be loaded in Flink. Note that the design of the graph data structure plays a key role for the overall efficiency of the query engine in a distributed dataflow scenario, given the importance of efficient read/write access to the data as well as efficient data shuffling.

Listing 2 represents the code to load the following RDF triple from Figure 1, ⟨ex:John, foaf:knows, ex:Alice⟩, based

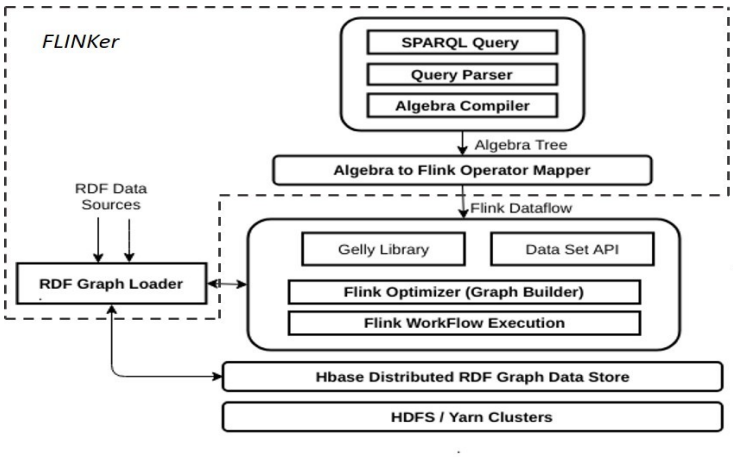

Fig. 3: FLINKer Architecture Overview

on Gelly data structures. First, the ***Vertex*** data type represents the subject (*ex:John*) and the object (*ex:Alice*) values. Note that each vertex consists of the concrete value and a given identifier (of type long in our example). While, the RDF predicates are represented by the ***Edge*** data type. An edge is defined by a source vertex identifier, a target vertex identifier and a predicate value ((*foaf:knows* in our case). Finally, the lists of all vertices and edges are used to create the final Gelly graph data structure.

Listing 2: Loading RDF data in Gelly

```
// Create Flink environment
ExecutionEnvironment env = ExecutionEnvironment.
    getExecutionEnvironment();

// E.g. add the triple, (ex:John, foaf:knows, ex:Alice)
Vertex<Long,String> vertex1 = new Vertex<Long, String>(1L,
    "ex:John");
Vertex<Long,String> vertex2 = new Vertex<Long, String>(2L,
    "ex:Alice");
Edge<Long, String> edge1 = new Edge<Long, String>(1L, 2L, "
    foaf:knows");

// Load all vertices (RDF nodes)
List<Vertex<Long,String>> vertexList = new ArrayList<Vertex
    <Long,String>>() {{
    add(vertex1);
    add(vertex2);
}};

// Load all edges (triples)
List<Edge<Long, String>> edgeList = new ArrayList<Edge<Long
    , String>>(){{
    add(edge1);
}};

// Create the graph
Graph<Long, Long, String> graph = Graph.fromCollection(
    vertexList, edgeList, env);
```

C. Query Processing and Vertex-centric BGP Matching

The FLINKer query processing is based on translating SPARQL queries into relational operations[14], which are then resolved through the existing operators in the DataSet API of Flink.

In particular, FLINKer performs basic graph pattern (BGP) matching of SPARQL queries based on the vertex-centric model. The computation is performed through a sequence of iterations steps called supersteps. First the triple patterns of the BGP are broadcast to all graph vertices. After that, each vertex matches its edge labels with the triple's predicate. In the next iterations, the match candidates are iteratively validated between neighbor vertices using message exchanging. Finally the partial results are incrementally unioned. An example of this process for a given BGP is outlined in Algorithm 1 (cf. see [31] for a similar approach in Spark). In a first superstep (lines 3 -12), for each triple pattern in the BGP (line 4) we iterate over each edge in the graph (line 6), i.e. each triple, and we check if it can match the corresponding triple pattern (line 7). In such case, we add the subject and object as a potential candidate vertex (lines 8-9). If a triple pattern is not matched, we can automatically assure that no solution in the graph can be found, and the execution is finished (lines 11-12). Otherwise, in a series of supersteps (lines 12-20), we check the validity of each candidate vertex with respect to the full BGP (line 15). If the match is still valid (line 16), we add the neighbors of the vertex as potential candidates for further solutions, and we send a message to the neighbors for validation in the next supersttep (lines 17-18). Otherwise, if the vertex is no longer valid, we remove it from the candidates (line 20). The final candidate vertices are then valid and returned as the final solution (line 21).

Gelly supports this vertex-centric model via two user-defined functions: a vertex computation function that validates the match candidates and a function that implements the message which is sent to other vertices.

Gelly implements iterative algorithms by defining a step function and passing it to the iteration operators. As shown in Table I, Flink has a particular graph processing feature, as it provides two versions of the iteration operators: *Iterate* and *Delta Iterate*[15]. The *Delta Iterate* operator leads to more

[14]In practice, FLINKer uses the Jena ARQ [10] SPARQL processor to parse a given SPARQL query and generate a parsing tree of operations

[15]See https://flink.apache.org/news/2015/08/24/introducing-flink-gelly.html

Algorithm 1: BGPMATCHING

```
Input: G:(V,E), BGP:Set[(s,p,o)]
Output: The vertices matching the given pattern
candidateVertex ← ()
candidateBGP ← ()
// Superstep 1
foreach tp ∈ BGP do
    foundCandidate ← false
    foreach (v_s, predicate, v_o) ∈ E do
        if match(tp, v_s, predicate, v_o) then
            candidateVertex ← candidateVertex ∪ v_s
            candidateVertex ← candidateVertex ∪ v_o
            foundCandidate ← true
    if foundCandidate == false then
        return ∅
// Supersteps 2..n
foreach v ∈ candidateVertex do
    valid ← validateMatch(v, BGP)
    if valid == true then
        candidateVertex ←
          candidateVertex ∪ neighbors(v)
        sendToNeighbors(v)
    else
        candidateVertex ← candidateVertex \ v
return candidateVertex
```

efficient incremental algorithms through identifying two input sets: *WorkSet* and *SolutionSet*. WorkSet contains the graph elements that require re-computation in the next iteration, while the *SolutionSet* represents the current solution state of the input.

The vertex-centric model provided by Gelly is internally mapped to a *Delta Iteration*. Thus, the *SolutionSet* is the vertex set of the input RDF graph while the *WorkSet* holds the active vertices (RDF subjects and objects), i.e. those vertices that have received messages from the previous supersteps. In each superstep, the messages are sent to the vertex neighbors by using the *coGroup* operator to co-group the active vertices (i.e. the *WorkSet*) considering the different edges. Next, the vertex-update function is applied by co-grouping the messaging function results with the current vertex values (i.e. the *SolutionSet*). Finally the output of the *coGroup* operator is used to update the *SolutionSet* in addition to preparing the *WorkSet* for the next iterations.

Note that the main advantage of using Gelly as a backend is that Flink has native iterative support, i.e. the iterations do not require new job scheduling overheads to be performed. In addition, the Flink optimizer is able to detect if there is loop invariant data so it will be pushed out of the loop to enhance the performance. Moreover, Gelly maintains the state as an index so that the updates are very efficient.

V. DISCUSSION

Providing efficient and scalable distributed SPARQL query resolution on top of existing Big Data engines is a relevant and emerging area of interest, unfortunately, the few current approaches are compromised by performance drawbacks at Web scale [3], [13], [31]. Our current work is focused on the efficient implementation and optimization of distributed SPARQL query resolution in Flink, based on the inherent capabilities of Gelly, its graph processing framework. Our main efforts concern two complementary challenges, *data distribution* and *auxiliary indexing*.

a) Data distribution: The logical idea of data distribution is to distribute related data to identical storage nodes to minimize the connections between nodes. In the graph-based scenario of RDF data, this means to provide a graph partitioning strategy to keep the related vertices in identical nodes [23]. In spite of existing RDF graph partitioning approaches [26], no strategy is optimal for all SPARQL queries and RDF structures [3]. Investigating which partition strategy to use in which situation remains an open challenge.

b) Auxiliary indexes: In our current proposal, we exploit the native Gelly graph processing mechanisms, with particular attention to the iterative model to support efficient vertex-centric SPARQL query resolution. In addition, different auxiliary indexes could be built to boost the current performance. In particular, it is worth mentioning that, in the current approach, each vertex and edge carries both the textual representation of the corresponding term (i.e. the subject, predicate or object textual value), as well as an identifier to denote the such vertex or edge respectively. Several RDF engines, such as HDT [15], split this information and consider an RDF dictionary to map all RDF terms to identifiers, while the graph structure is then represented as a graph of identifiers. The same philosophy can be used in Gelly, where a dictionary can be indexed in a Flink dataset, which can be distributed among the nodes, and the graph of identifiers can be represented in Gelly. In addition, in-memory auxiliary indexes, e.g. represented in the aforementioned compressed HDT structure, can be built to improve the performance of certain queries. For instance, highly connected vertex (i.e. nodes with a high out-in degree) can be grouped and represented in HDT indexes in order to avoid the well-known overhead of the vertex-centric approach in such scenarios [31].

c) Cost Based Optimization: The order of Flink operators has a significant impact on the optimization of the query execution. Apache Flink offers a dataflow optimizer that determines the suitable join strategy for the execution plan. Flink optimizer selects between partitioning as opposed to broadcasting in addition to hash join versus sort-merge join. In addition, FLink optimizer benefits from reusing partitioning and sort orders across operators and super steps. However, the Flink optimizer lacks optimization capabilities that we are considering to address in FLINKer. First, the optimizer does not use statistics about data characteristics which is crucial for the query performance, hence we plan to consider pre-computed statistics about the dataset. Moreover, Flink does not optimize the operator reordering. In FLINKer, we plan to exploit the operators reordering for better query performance. Finally, The data sources are always fully scanned. However, FLINKer aims at supporting smart data scanning with pre-select and project operations.

VI. Conclusions and Future Work

Herein we have introduced FLINKer, an approach to distribute RDF processing and querying via Flink's native streaming processing framework. We have reviewed current vertex-centric distributed graph processing approaches and discussed how the Flink architecture, and its graph processor (Gelly), can be extended to support SPARQL queries over RDF data. We also highlighted challenges and potential optimizations, such as efficient graph partitioning and query planning.

In future work, we plan to optimize SPARQL query plans to take advantage of the Flink native features, exploring different partitioning strategies. Finally we aim to perform a large scale evaluation to compare FLINKer against the existing state-of-the-art RDF engines providing distributed SPARQL query resolution.

Acknowledgements

Supported by the EUs Horizon 2020 research and innovation programme: grant 731601 (SPECIAL) and the Austrian Research Promotion Agency's (FFG) program "ICT of the Future": grant 861213 (CitySpin).

References

[1] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proc. of VLDB*, pages 411–422. VLDB Endowment, 2007.

[2] S. Abbassi and R. Faiz. RDF-4X: A Scalable Solution for RDF Quads Store in the Cloud. In *Proc. of MEDES*, pages 231–236, 2016.

[3] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis. A survey and experimental comparison of distributed sparql engines for very large rdf data. *Proc. of the VLDB Endowment*, 10(13):2049–2060, 2017.

[4] I. Abdelaziz, R. Harbi, S. Salihoglu, and P. Kalnis. Combining Vertex-centric Graph Processing with SPARQL for Large-scale RDF Data Analytics. *Proc. of TPDS*, 28(12):3374–3388, 2017.

[5] I. Abdelaziz, R. Harbi, S. Salihoglu, P. Kalnis, and N. Mamoulis. SPARTex: a vertex-centric framework for RDF data analytics. *Proc. of the VLDB Endowment*, 8(12):1880–1883, 2015.

[6] 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 *Proc. of SoCC*, pages 119–130, 2010.

[7] C. Bizer, T. Heath, and T. Berners-Lee. Linked Data - The Story So Far. *IJSWIS*, 5(3):1–22, 2009.

[8] P. Boncz, O. Erling, and M.-D. Pham. Advances in large-scale RDF data management. In *Linked Open Data–Creating Knowledge Out of Interlinked Data*, pages 21–44. 2014.

[9] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[10] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *Proceedings of the WWW*, pages 74–83. ACM, 2004.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[12] J.-H. Du, H. Wang, Y. Ni, and Y. Yu. HadoopRDF: A Scalable Semantic Data Analytical Engine. In *Proc. of ICIC*, 2012.

[13] J. Feng, C. Meng, J. Song, X. Zhang, Z. Feng, and L. Zou. SPARQL Query Parallel Processing: A Survey. In *Proc. of BigData*, pages 444–451. IEEE, 2017.

[14] J. D. Fernández, W. Beek, M. A. Martínez-Prieto, and M. Arias. LOD-a-lot: A queryable dump of the LOD cloud. In *Proc. of ISWC*, pages 75–83, 2017.

[15] J. D. Fernández, M. A. Martínez-Prieto, C. Gutiérrez, A. Polleres, and M. Arias. Binary RDF Representation for Publication and Exchange (HDT). *JWS*, 19:22–41, 2013.

[16] F. Goasdoué, Z. Kaoudi, I. Manolescu, J.-A. Quiané-Ruiz, and S. Zampetakis. Cliquesquare: Flat plans for massively parallel RDF queries. In *Proc. of ICDE*, pages 771–782. IEEE, 2015.

[17] D. Graux, L. Jachiet, P. Genevès, and N. Layaïda. SPARQLGX: Efficient Distributed Evaluation of SPARQL with Apache Spark. In *Proc. of ISWC*, 2016.

[18] C. Gutiérrez, C. Hurtado, A. O. Mendelzon, and J. Perez. Foundations of Semantic Web Databases. *JCSS*, 77:520–541, 2011.

[19] S. Harris and A. Seaborne. SPARQL 1.1 Query Language. W3C Recommendation, Mar. 2013.

[20] A. Harth and S. Decker. Optimized index structures for querying rdf from the web. In *Proc. of LA-WEB*, pages 10–pp. IEEE, 2005.

[21] 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):1256–1267, July 2012.

[22] M. Junghanns, M. Kiessling, A. Averbuch, A. Petermann, and E. Rahm. Cypher-based Graph Pattern Matching in Gradoop. In *Proc. of GRADES*, pages 3:1–3:8. ACM, 2017.

[23] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[24] J. Leeka and S. Bedathur. RQ-RDF-3X: going beyond triplestores. In *Proc. of ICDEW*, pages 263–268. IEEE, 2014.

[25] J. Lehmann, G. Sejdiu, L. Bühmann, P. Westphal, C. Stadler, I. Ermilov, S. Bin, N. Chakraborty, M. Saleem, A.-C. N. Ngomo, et al. Distributed semantic analytics using the sansa stack. In *Proc. of ISWC*, pages 147–155, 2017.

[26] Y. Leng, Z. Chen, H. Wang, and F. Zhong. A Partitioning and Index Algorithm for RDF Data of Cloud-based Robotic Systems. *IEEE Access*, 2018.

[27] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proc. of SIGMOD*, pages 135–146. ACM, 2010.

[28] N. Papailiou, D. Tsoumakos, I. Konstantinou, P. Karras, and N. Koziris. H2rdf+: an efficient data management system for big rdf graphs. In *Proc. of SIGMOD*, pages 909–912. ACM, 2014.

[29] K. Rohloff and R. E. Schantz. High-performance, massively scalable distributed systems using the mapreduce software framework: The shard triple-store. In *Proc. of PSI EtA*, pages 4:1–4:5. ACM, 2010.

[30] S. Sakr, M. Wylot, R. Mutharaju, D. Le Phuoc, and I. Fundulaki. *Linked Data: Storing, Querying, and Reasoning*. Springer, 2018.

[31] A. Schätzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen. S2X: graph-parallel querying of RDF with GraphX. In *Proc. of Big-O(Q) and DMAH*, volume 9579 of *LNCS*, pages 155–168, 2016.

[32] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In *Proc. of SWIM*, pages 4:1–4:8. ACM, 2011.

[33] A. Schätzle, M. Przyjaciel-Zablocki, A. Neu, and G. Lausen. Sempala: Interactive SPARQL Query Processing on Hadoop. In *Proc of ISWC*, pages 164–179. Springer, 2014.

[34] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen. S2RDF: RDF Querying with SPARQL on Spark. *Proc. VLDB Endow.*, 9(10):804–815, June 2016.

[35] G. Schreiber and Y. Raimond. *RDF 1.1 Primer*. W3C Working Group Note, 2014.

[36] A. Schtzle, M. Przyjaciel-Zablocki, T. Berberich, and G. Lausen. S2X: Graph-Parallel Querying of RDF with GraphX. In *Big-O(Q)/DMAH@VLDB*, volume 9579 of *LNCS*, pages 155–168. Springer, 2015.

[37] J. Urbani, J. Maassen, and H. Bal. Massive Semantic Web data compression with MapReduce. In *Proc. of HPDC*, pages 795–802. ACM, 2010.

[38] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[39] X. Wilcke, P. Bloem, and V. De Boer. The knowledge graph as the default data model for learning on heterogeneous knowledge. *Data Science*, 1(1-2):39–57, 2017.

[40] Z. Xu, W. Chen, L. Gai, and T. Wang. Sparkrdf: In-memory distributed rdf management framework for large-scale social data. In *WAIM*, 2015.

[41] 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):56–65, Oct. 2016.