S2X: Graph-Parallel Querying of RDF with GraphX

Alexander Schätzle, Martin Przyjaciel-Zablocki,
Thorsten Berberich, and Georg Lausen

Department of Computer Science, University of Freiburg
Georges-Köhler-Allee 051, 79110 Freiburg, Germany
{schaetzle,zablocki,lausen}@informatik.uni-freiburg.de

Abstract. RDF has constantly gained attention for data publishing due to its flexible data model, raising the need for distributed querying. However, existing approaches using general-purpose cluster frameworks employ a record-oriented perception of RDF ignoring its inherent graph-like structure. Recently, GraphX was published as a graph abstraction on top of Spark, an in-memory cluster computing system. It allows to seamlessly combine graph-parallel and data-parallel computation in a single system, an unique feature not available in other systems. In this paper we introduce S2X, a SPARQL query processor for Hadoop where we leverage this unified abstraction by implementing basic graph pattern matching of SPARQL as a graph-parallel task while other operators are implemented in a data-parallel manner. To the best of our knowledge, this is the first approach to combine graph-parallel and data-parallel computation for SPARQL querying of RDF data based on Hadoop.

Keywords: RDF, SPARQL, Hadoop, Spark, GraphX

1 Introduction

Driven by initiatives like *Schema.org*, one can expect the amount of RDF [8] data to grow steadily towards massive scale, requiring distributed solutions to store and query it. Likewise, the Hadoop ecosystem has become the de facto standard in the area of large-scale distributed processing and has been applied to many diverse application fields. The recent development and integration of in-memory frameworks (e.g. *Spark*, *Impala*) facilitates even further application fields covering not only classical ETL-like but also more interactive workloads.

In addition, storing RDF data in a common data pool (e.g. *HDFS*) among various Hadoop-based applications enables manifold synergy benefits compared with dedicated infrastructures, where interoperability between different systems comes along with high integration costs. Consequently, there already exists some work on querying RDF with Hadoop, e.g. [6, 7, 9, 11, 12], but they all follow a relational-style model to manage RDF data. However, the RDF data model can also be interpreted as a graph and query processing (in terms of SPARQL) as a task to identify subgraphs that match a query pattern. Implementing such graph

computations using *data-parallel* frameworks like MapReduce or Spark can be challenging and inefficient as they do not provide any graph abstraction and do not exploit their inherent graph structure. Specialized parallel graph processing systems like *Pregel*, *Giraph* and *GraphLab* (see [5] for comparison) are tailored towards such iterative graph algorithms but they are either not integrated in the Hadoop ecosystem or come with their own programming abstraction that cannot be combined with existing data-parallel frameworks. Thus, in order to compose graph-parallel and data-parallel operations, the data has to be moved or copied from one system to the other.

Recently, an abstraction for graph-parallel computation was added to Spark, called *GraphX* [3]. It is implemented on top of Spark but complements it with graph-specific optimizations similar to those in specialized graph processing systems. This bridges the gap between the record-centric view of data-parallel frameworks and the graph-parallel computation of specialized systems, paving the way for new applications that seamlessly combine the best of both worlds. Moreover, it fits to our concept of having a common data pool as it uses HDFS for permanent storage.

In this paper we introduce S2X (SPARQL on Spark with GraphX). It combines graph-parallel abstraction of GraphX to implement the graph pattern matching part of SPARQL with data-parallel computation of Spark to build the results of other SPARQL operators. There is not much other work yet on querying RDF with SPARQL using a parallel graph processing framework. In [4] the authors present an implementation on top of GraphLab while Trinity.RDF [15] is an RDF implementation on top of the Trinity graph engine which uses a distributed key-value store. However, both approaches use their own dedicated data storage which conceptually differs from our approach to use a common data pool for synergy reasons. To the best of our knowledge, there is currently no other work on SPARQL querying on Hadoop using a native graph abstraction for RDF due to the lack of suitable graph processing frameworks for Hadoop. We think that GraphX is a promising step to bridge this gap.

Our major contributions are as follows: (1) We define a mapping from RDF to the property graph model of GraphX. (2) Based on this model we introduce S2X, a SPARQL implementation on top of GraphX and Spark. (3) Finally, we provide some preliminary experiments to compare S2X with a state of the art SPARQL engine that uses MapReduce as execution layer.

2 Graph-Parallel Computation with GraphX

Spark [13] is a general-purpose in-memory cluster computing system that can run on Hadoop. The central data structure is a so-called *Resilient Distributed Dataset* (RDD) [14] which is a fault-tolerant collection of elements that can be operated on in parallel. Spark attempts to keep an RDD in memory and partitions it across all machines in the cluster. Conceptually, Spark adopts a *data-parallel* computation model that builds upon a record-centric view of data,

similar to *MapReduce* and *Apache Tez*. A job is modeled as a directed acyclic graph (DAG) of tasks where each task runs on a horizontal partition of the data.

Spark also comes with a rich stack of high-level tools, including an API for graphs and *graph-parallel* computation called *GraphX* [3]. It adds an abstraction to the API of Spark to ease the usage of graph data and provides a set of typical graph operators. It is meant to bridge the gap between data-parallel and graph-parallel computation in a single system such that data can be seamlessly viewed both as a graph and as collections of items without data movement or duplication. Graph-parallel abstraction builds upon a vertex-centric view of graphs where parallelism is achieved by graph partitioning and computation is expressed in the form of user-defined vertex programs that are instantiated concurrently for each vertex and can interact with adjacent vertex programs. Like many other graph-parallel systems, GraphX adopts the *bulk-synchronous parallel* (BSP) execution model where all vertex programs run concurrently in a sequence of so-called *supersteps*. In every superstep a vertex program performs its local transformations and can exchange information with adjacent vertices which is then available to them in the following superstep.

GraphX uses a *vertex-cut* partitioning strategy that evenly assigns edges to machines and allows vertices to span multiple machines in a way that the number of machines spanned by each vertex is minimized. Internally, a graph is represented by two separate collections for edges (EdgeRDD) and vertices (VertexRDD). The graph operators of GraphX are likewise expressed as a combination of data-parallel operations on these collections in Spark with additional graph-specific optimizations inspired from specialized graph processing systems. For example, as many graph computations need to join the edge and vertex collection, GraphX maintains a routing table co-partitioned with the vertex collection such that each vertex is sent only to the edge partitions that contain adjacent edges. This way, GraphX achieves performance parity with specialized graph processing systems (e.g. *Giraph* and *GraphLab*) [3].

3 RDF Property Graph Model in S2X

The property graph data model of GraphX combines the graph structure with vertex and edge properties. Formally, a *property graph* is a directed multigraph and can be defined as $PG(P) = (V, E, P)$ where V is a set of vertices and $E = \{(i, j) \mid i, j \in V\}$ a set of directed edges from i (*source*) to j (*target*). Every vertex $i \in V$ is represented by an unique identifier. $P_V(i)$ denotes the properties of vertex $i \in V$ and $P_E(i, j)$ the properties of edge $(i, j) \in E$. $P = (P_V, P_E)$ is the collection of all properties. In the following we use the notation $i.x = P_V(i).x$ for property x of vertex $i \in V$ and $(i, j).y = P_E(i, j).y$ for property y of edge $(i, j) \in E$, respectively. The separation of structure and properties is an important design aspect of GraphX as many applications preserve the structure of the graph while changing its properties, as we also do in S2X.

In RDF [8] the basic notion of data modeling is a so-called *triple* $t = (s, p, o)$ where s is called *subject*, p *predicate* and o *object*, respectively. It can be inter-

preted as an edge from s to o labeled with p, $s \xrightarrow{p} o$. An RDF dataset is a set of triples and hence forms a directed labeled graph. The simplicity and flexibility of RDF allows to model any kind of knowledge about arbitrary resources, represented by global identifiers (*IRIs*), e.g. the IRI for Leonardo da Vinci in DBpedia is `http://dbpedia.org/resource/Leonardo_da_Vinci`.

We now define how to represent an RDF graph $G = \{t_1, \ldots, t_n\}$ in the property graph data model of GraphX. Let $S(G) = \{s \mid \exists p, o : t = (s, p, o) \in G\}$ be the set of all subjects in G. The sets of all predicates $P(G)$ and all objects $O(G)$ are defined accordingly. Then $PG(P) = (V, E, P)$ is the corresponding property graph for G with $V = S(G) \cup O(G)$, $E = \{(s, o) \mid \exists t = (s, p, o) \in G\}$, $P_V.label : V \to S(G) \cup O(G)$ and $P_E.label : E \to P(G)$. IRIs cannot be used as vertex identifiers in GraphX as it requires 64-bit integers for efficiency reasons. We use the provided `zipWithUniqueID` function of Spark to derive unique integer IDs for all subjects and objects in G and preserve the original terms in the *label* property. Hence, every triple $t = (s, p, o) \in G$ is represented by two vertices $v_s, v_o \in V$, an edge $(v_s, v_o) \in E$ and properties $v_s.label = s$, $v_o.label = o$, $(v_s, v_o).label = p$.

For example, the corresponding property graph representation of an RDF graph $G_1 = \{(userA, knows, userB), (userA, likes, userB), (userA, likes, userC), (userB, knows, userC)\}$ is illustrated in Figure 1. For brevity, we use a simplified notation of RDF without IRIs.

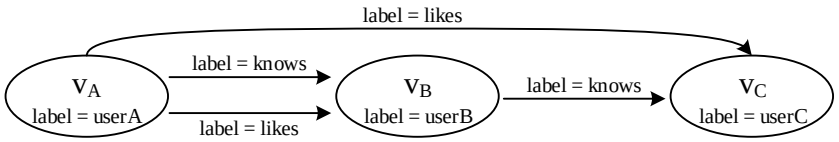

Fig. 1. Property graph representation of RDF graph G_1

4 Query Processing in S2X

SPARQL [10] is the W3C recommended query language for RDF. It combines graph pattern matching with additional more relational-style operators like OPTIONAL and FILTER to produce a set of so-called *solution mappings*, i.e. it does not produce triples and hence is not closed. For example, the SPARQL query Q in Listing 1.1 retrieves a single result (or solution mapping) for the RDF graph G_1 in Section 3: $\{(?A \to userA, ?B \to userB, ?C \to userC)\}$

Listing 1.1. SPARQL query Q

```
SELECT * WHERE {
  ?A knows ?B . ?A likes ?B . ?B knows ?C
}
```

More formally, the basic notion in SPARQL is a so-called *triple pattern* $tp = (s', p', o')$ with $s' = \{s, ?s\}$, $p' = \{p, ?p\}$ and $o' = \{o, ?o\}$, i.e. a triple where every part is either an RDF term (called *bound*) or a variable (indicated by ? and called *unbound*). A set of triple patterns forms a *basic graph pattern* (BGP). Consequently, the query in Listing 1.1 contains a single BGP $bgp_Q = \{tp_1, tp_2, tp_3\}$ with $tp_1 = (?A, knows, ?B)$, $tp_2 = (?A, likes, ?B)$ and $tp_3 = (?B, knows, ?C)$.

Let V be the infinite set of query variables and T be the set of valid RDF terms. A *(solution) mapping* μ is a partial function $\mu : V \rightarrow T$. We call $\mu(?v)$ the variable binding of μ for $?v$ and $vars(tp)$ the set of variables contained in triple pattern tp. Abusing notation, for a triple pattern tp we call $\mu(tp)$ the triple that is obtained by substituting the variables in tp according to μ. The *domain* of μ, $dom(\mu)$, is the subset of V where μ is defined. Two mappings μ_1, μ_2 are called *compatible*, $\mu_1 \sim \mu_2$, iff for every variable $?v \in dom(\mu_1) \cap dom(\mu_2)$ it holds that $\mu_1(?v) = \mu_2(?v)$. It follows that mappings with disjoint domains are always compatible and the set-union (merge) of two compatible mappings, $\mu_1 \cup \mu_2$, is also a mapping. The answer to a triple pattern tp for an RDF graph G is a bag of mappings $\Omega_{tp} = \{\mu \mid dom(\mu) = vars(tp), \mu(tp) \in G\}$. The merge of two bags of mappings, $\Omega_1 \bowtie \Omega_2$, is defined as the merge of all compatible mappings in Ω_1 and Ω_2, $\Omega_1 \bowtie \Omega_2 = \{(\mu_1 \cup \mu_2) \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, \mu_1 \sim \mu_2\}$. Finally, the answer to a basic graph pattern $bgp = \{tp_1, \ldots, tp_m\}$ is then defined as the merge of all bags of mappings for $tp_1, \ldots, tp_m$, $\Omega_{bgp} = \Omega_{tp_1} \bowtie \ldots \bowtie \Omega_{tp_m}$.

All other SPARQL operators use a bag of mappings as input, i.e. they are not evaluated on the underlying RDF graph directly but on the result of one or more BGPs or other operators. Thus, the actual graph pattern matching part of SPARQL is defined by BGPs whereas the other operators are defined in a more relational fashion. In S2X we take advantage of the fact that GraphX is not a standalone specialized graph processing system but a graph abstraction library on top of Spark. BGP matching in S2X is realized in a graph-parallel manner using the GraphX API and the other operators and modifiers (OPTIONAL, FILTER, ORDER BY, PROJECTION, LIMIT and OFFSET) are realized in a more data-parallel manner using the API of Spark. Relying on Spark as the back-end of S2X, graph-parallel and data-parallel computation can be combined smoothly without the need for data movement or duplication.

Fard et al. [2] present an approach for distributed vertex-centric pattern matching on directed graphs with labeled vertices. In contrast to RDF, there are no edge labels and at most one edge between two vertices. In S2X we follow the idea of the *dual simulation* algorithm from [2] but adapt it to our property graph representation of RDF (cf. Section 3) and SPARQL BGP matching.

4.1 Match Candidates & Match Sets

The basic idea of our BGP matching algorithm is that every vertex in the graph stores the variables of a query where it is a possible candidate for. We start with matching all triple patterns of a BGP independently and then exchange messages between adjacent vertices to validate the match candidates until they do not change anymore.

A *match candidate* of vertex v is a 4-tuple $matchC = (v, ?var, \mu, tp)$. $?var \in vars(tp)$ is the variable of tp where v is a candidate for and μ is the corresponding mapping for tp with $dom(\mu) = vars(tp)$ and $\mu(?var) = v.label$. The set of all candidate matches of a vertex v is called *match set* of v, $matchS(v)$.

The match set of a vertex is also called its *local* match set whereas the match sets of adjacent vertices are called *remote* match sets. We store the local match set of a vertex v as a property of this vertex, $v.matchS$. Conceptually, the whole process can be summarized as follows:

1. All possibly relevant vertices (i.e. *match candidates*) are determined by matching each edge with all triple patterns from the BGP (Superstep 1 in Algorithm 1).
2. Match candidates are validated using local (Algorithm 2) and remote match sets (Algorithm 3) and invalid ones get discarded.
3. Locally changed match sets are sent to their neighbors in the graph for validation in next superstep.
4. Process continues with step 2 until no changes occur.
5. The determined subgraph(s) are collected and merged to produce the final SPARQL compliant output (Algorithm 4).

4.2 BGP Matching

Let $G = \{t_1, \ldots, t_n\}$ be an RDF graph, $PG(P)$ the corresponding property graph and $bgp = \{tp_1, \ldots, tp_m\}$ a BGP. The algorithm for BGP matching is depicted in Algorithm 1.

Initially (Superstep 1), we iterate over all edges from PG and check whether it matches any of the triple patterns from bgp, i.e. $\exists t_i \in G, tp_j \in bgp, \mu : \mu(tp_j) = t_i$. Recap that every edge of PG corresponds to a triple in G. If there is a match, we generate a *match candidate* for every variable in subject or object position of the triple pattern and store it in the corresponding vertex of PG. For example, consider a triple $t = (userA, knows, userB)$ and a triple pattern $tp = (?A, knows, ?B)$. t matches tp and hence we derive the following two match candidates:

$$matchC_1 = (v_A, ?A, (?A \rightarrow userA, ?B \rightarrow userB), tp)$$
$$matchC_2 = (v_B, ?B, (?A \rightarrow userA, ?B \rightarrow userB), tp)$$

$matchC_1$ is stored in the match set of v_A (line 7) and $matchC_2$ in the match set of v_B (line 10), respectively. Recap that v_A is the vertex for $userA$ in the corresponding property graph and v_B the vertex for $userB$, respectively (cf. Figure 1). If there is a triple pattern from bgp that does not find any match, the algorithm terminates as bgp cannot be fulfilled (line 12). After this initialization, the match sets of all vertices yield an overestimation of the result for bgp as they do not take into account that the generated mappings of the match candidates have to be compatible. For example, consider again G_1 from Section 3 and bgp_Q from query Q in Listing 1.1. Table 1 lists the match sets of vertices v_A, v_B and v_C after the first superstep of Algorithm 1.

Algorithm 1: BGPMATCHING

input: $PG(P) : (V, E, P)$
$BGP : Set\langle TriplePattern : (s, p, o)\rangle$
output: $matchVertices : Set\langle v : V\rangle$

$matchTp : Set\langle TriplePattern\rangle \leftarrow \emptyset$, $matchVertices \leftarrow \emptyset$
// Superstep 1
foreach $(v_s, v_o) \in E$ **do**
  foreach $tp : TriplePattern \in BGP$ **do**
    if $\exists \mu : \mu(tp) = (v_s.label, (v_s, v_o).label, v_o.label)$ **then**
      $matchTP \leftarrow matchTP \cup tp$
      if *isVar(tp.s)* **then**
        $v_s.matchS \leftarrow v_s.matchS \cup (v_s, tp.s, \mu, tp)$
        $matchVertices \leftarrow matchVertices \cup v_s$
      if *isVar(tp.o)* **then**
        $v_o.matchS \leftarrow v_o.matchS \cup (v_o, tp.o, \mu, tp)$
        $matchVertices \leftarrow matchVertices \cup v_o$
if $matchTP \neq BGP$ **then return** $\emptyset$
else $activeVertices \leftarrow matchVertices$
// Superstep 2...n
while $activeVertives \neq \emptyset$ **do**
  $tmpVertices \leftarrow \emptyset$
  foreach $v : V \in activeVertices$ **do**
    $old \leftarrow v.matchS$
    if $\#superstep > 2$ **then**
      `validateRemoteMatchSet(`v, BGP`)`
    `validateLocalMatchSet(`v, BGP`)`
    if $v.matchS = \emptyset$ **then**
      $matchVertices \leftarrow matchVertices \setminus \{v\}$
    if $v.matchS \neq old \parallel \#superstep = 2$ **then**
      $tmpVertices \leftarrow tmpVertices \cup v.neighbors$
      `sendToAllNeighbors(`v, $v.matchS$`)`
  $activeVertices \leftarrow tmpVertices$
return $matchVertices$

Algorithm 2: VALIDATELOCALMATCHSET

input: $v : V$, $BGP : Set\langle TriplePattern : (s, p, o)\rangle$

foreach $matchC_1 : (v, ?var, \mu, tp) \in v.matchS$ **do**
  foreach $tp \in BGP \neq matchC_1.tp$ **do**
    if $matchC_1.?var \in vars(tp)$ **then**
      if $\nexists\ matchC_2 : (v, ?var, \mu, tp) \in v.matchS :$
        $tp = matchC_2.tp\ \ \&$
        $matchC_1.?var = matchC_2.?var\ \ \&$
        $matchC_1.\mu \sim matchC_2.\mu$ **then**
        $v.matchS \leftarrow v.matchS \setminus \{matchC_1\}$
        break

Algorithm 3: validateRemoteMatchSet

input: $v : V$, $remoteMS : Set\langle matchC : (v, ?var, \mu, tp)\rangle$

```
1 foreach matchC_1 : (v, ?var, μ, tp) ∈ v.matchS do
2     ?var2 ← vars(matchC_1.tp) \ {matchC_1.?var}
3     if ?var2 ≠ ∅ then
4         if ∄ matchC_2 : (v, ?var, μ, tp) ∈ remoteMS :
5           ?var2 = matchC_2.?var  &
6           matchC_1.tp = matchC_2.tp  &
7           matchC_1.μ ∼ matchC_2.μ then
8             v.matchS ← v.matchS \ {matchC_1}
```

Table 1. Match candidates & sets in every Superstep of BGP matching (G_1, bgp_Q) with $tp_1 = (?A, knows, ?B)$, $tp_2 = (?A, likes, ?B)$, and $tp_3 = (?B, knows, ?C)$

	Superstep 1			**Superstep 2**		
v	$?var$	μ	tp	$?var$	μ	tp
v_A	$?A$	$(?A \to userA, ?B \to userB)$	tp_1	$?A$	$(?A \to userA, ?B \to userB)$	tp_1
	$?B$	$(?B \to userA, ?C \to userB)$	tp_3		—	
	$?A$	$(?A \to userA, ?B \to userB)$	tp_2	$?A$	$(?A \to userA, ?B \to userB)$	tp_2
	$?A$	$(?A \to userA, ?B \to userC)$	tp_2		—	
v_B	$?B$	$(?A \to userA, ?B \to userB)$	tp_1	$?B$	$(?A \to userA, ?B \to userB)$	tp_1
	$?C$	$(?B \to userA, ?C \to userB)$	tp_3	$?C$	$(?B \to userA, ?C \to userB)$	tp_3
	$?B$	$(?A \to userA, ?B \to userB)$	tp_2	$?B$	$(?A \to userA, ?B \to userB)$	tp_2
	$?A$	$(?A \to userB, ?B \to userC)$	tp_1		—	
	$?B$	$(?B \to userB, ?C \to userC)$	tp_3	$?B$	$(?B \to userB, ?C \to userC)$	tp_3
v_C	$?B$	$(?A \to userA, ?B \to userC)$	tp_2		—	
	$?B$	$(?A \to userB, ?B \to userC)$	tp_1		—	
	$?C$	$(?B \to userB, ?C \to userC)$	tp_3	$?C$	$(?B \to userB, ?C \to userC)$	tp_3
	Superstep 3			**Superstep 4**		
v	$?var$	μ	tp	$?var$	μ	tp
v_A	$?A$	$(?A \to userA, ?B \to userB)$	tp_1	$?A$	$(?A \to userA, ?B \to userB)$	tp_1
		—			—	
	$?A$	$(?A \to userA, ?B \to userB)$	tp_2	$?A$	$(?A \to userA, ?B \to userB)$	tp_2
		—			—	
v_B	$?B$	$(?A \to userA, ?B \to userB)$	tp_1	$?B$	$(?A \to userA, ?B \to userB)$	tp_1
		—			—	
	$?B$	$(?A \to userA, ?B \to userB)$	tp_2	$?B$	$(?A \to userA, ?B \to userB)$	tp_2
		—			—	
	$?B$	$(?B \to userB, ?C \to userC)$	tp_3	$?B$	$(?B \to userB, ?C \to userC)$	tp_3
v_C		—			—	
		—			—	
	$?C$	$(?B \to userB, ?C \to userC)$	tp_3	$?C$	$(?B \to userB, ?C \to userC)$	tp_3

In every following superstep $(2 \dots n)$, each vertex checks the validity of its local match set and sends it to its neighbors in the graph (both edge directions), if there are any changes. That is, in superstep i a vertex has access to the remote match sets of its neighbors from superstep $i-1$. The validation process can be grouped in two subtasks:

(1) If a vertex v is a candidate for variable $?var$, witnessed by a match candidate $matchC$ for $?var$, it must have a match candidate regarding $?var$ for all triple patterns $tp \in bgp$ that contain $?var$, i.e. $?var \in vars(tp)$. Furthermore, the mappings of all these match candidates must be compatible. Otherwise, $matchC$ is removed from the match set of v. This validation task only needs to consider the local match set of v and thus can already be executed in superstep 2 where remote match sets are not yet available. The validation of local match sets is depicted in Algorithm 2.

(2) Consider match candidate $matchC_1 = (v, ?var, \mu, tp)$ for vertex v where triple pattern tp contains another variable $?var2 \neq ?var$. Then there must be a neighbor of v connected by $tp.p$ which is a candidate for $?var2$. That is, there must exist a corresponding match candidate $matchC_2$ for $?var2$ in the remote match sets of v where the mappings of $matchC_1$ and $matchC_2$ are compatible. If not, $matchC_1$ is removed from the match set of v. The validation of remote match sets is depicted in Algorithm 3.

Continuing the example, in superstep 2 vertices v_A, v_B and v_C validate their local match sets without knowing the remote match sets of their neighbors, respectively. For example, v_A removes $(v_A, ?B, (?B \rightarrow userA, ?C \rightarrow userB), tp_3)$ from its local match set because in order to be a candidate for $?B$ it must also have match candidates for $?B$ with triple patterns tp_1 and tp_2. These do not exist as v_A has no incoming *knows* and *likes* edges. In addition, v_A also removes $(v_A, ?A, (?A \rightarrow userA, ?B \rightarrow userC), tp_2)$ as there is no match candidate for $?A$ with triple pattern tp_1 which is compatible to it. After superstep 2, v_A is not a candidate for $?B$ anymore as its match set contains no match candidate for $?B$ (cf. Superstep 2 in Table 1).

In superstep 3 v_B removes $(v_B, ?C, (?B \rightarrow userA, ?C \rightarrow userB), tp_3)$ from its local match set as v_A removed the corresponding match candidate for $?B$ in superstep 2 and hence it is not contained in the remote match sets at v_B in superstep 3. In superstep 4 the match sets do not change anymore and hence the algorithm terminates yielding v_A as a match for $?A$, v_B for $?B$ and v_C for $?C$, respectively.

4.3 From Graph-Parallel to Data-Parallel

The BGP matching algorithm determines the subgraph(s) of G (resp. PG) that match *bgp*, represented by the set of vertices that match some part of *bgp* ($matchVertices$) and the mappings for triple patterns stored in the match sets of each of these vertices. However, in SPARQL the result of a BGP is defined as a bag of mappings. To produce this final bag of mappings we have to merge the partial mappings of all matching vertices. This is depicted in Algorithm 4. For every triple pattern from *bgp* we collect the corresponding mappings

from the match sets of all vertices (line 6), which gives us m bags of mappings $(\Omega_{tp_1}, \ldots, \Omega_{tp_m})$. These bags are incrementally merged to produce the final result (line 7). A merge of two bags of mappings, $\Omega_1 \bowtie \Omega_2$, can be implemented as a join between the mappings from Ω_1 and Ω_2 where the join attributes are the variables that occur in both sides, i.e. $dom(\Omega_1) \cap dom(\Omega_2)$. This is a data-parallel operation using the API of Spark which result is a collection (RDD) of mappings for *bgp*. Concluding the example, Algorithm 4 outputs a single solution mapping for bgp_Q: $\{(?A \rightarrow userA, ?B \rightarrow userB, ?C \rightarrow userC)\}$.

Algorithm 4: GENERATESOLUTIONMAPPINGS

input: $matchVertices : Set\langle v : V\rangle$,
$BGP : Set\langle TriplePattern : (s, p, o)\rangle$
output: $\Omega_{bgp} : Set\langle \mu : SolutionMapping\rangle$

```
1 Ω_bgp ← ∅
2 foreach tp : TriplePattern ∈ BGP do
3 |   Ω_tp ← ∅
4 |   foreach v : V ∈ matchVertices do
5 |   |   if ∃ matchC : (v, ?var, μ, tp) ∈ v.matchS : matchC.tp = tp then
6 |   |   |   Ω_tp ← Ω_tp ∪ matchC.μ
7 |   Ω_bgp ← Ω_bgp ⋈ Ω_tp
8 return Ω_bgp
```

The remaining SPARQL operators are implemented using data-parallel operators from the Spark API, e.g. OPTIONAL is implemented by a left-outer join and FILTER as a filter function applied to a collection (RDD) of mappings.

5 Experiments

The experiments were performed on a small cluster with ten machines, each equipped with a six core Xeon E5-2420 1.9 GHz CPU, 2×2 TB disks and 32 GB RAM. We used the Hadoop distribution of Cloudera CDH 5.3.0 with Spark 1.2.0 and Pig 0.12. The machines were connected via Gigabit network. This is actually a low-end configuration as typical Hadoop nodes have 256 GB RAM, 12 disks or more and are connected via 10 Gigabit network or faster. We assigned 24 GB RAM for Spark where 60% was used to cache the graph in memory. Spark has manifold configuration parameters that can have severe impact on performance. In our tests, adjusting these parameters improved the execution time of certain queries up to an order of magnitude while revealing a significant slowdown for others. Hence, we decided to use the default configuration of Spark and GraphX for comparability reasons, although some queries could be significantly improved this way. GraphX is in a very early stage of development and not yet recommended for production usage, so one can expect substantial progress in this area with future versions.

The main goal of this preliminary evaluation was to confirm the overall feasibility of our approach and identify possible flaws that we should address for future revisions of S2X. To this end, we compare the current version of S2X with PigSPARQL [11], a SPARQL query engine on top of *Apache Pig*, which has proven to be a competitive baseline for SPARQL query processing on Hadoop. We decided to use the Waterloo SPARQL Diversity Test Suite (WatDiv) [1] as, in contrast to other existing benchmarks, its focus is to benchmark systems against varying query shapes to identify their strengths and weaknesses. The WatDiv data model combines an e-commerce scenario with a kind of social network. We generated datasets with scale factors 10, 100 and 1000 which corresponds to 10K, 100K and 1M users, respectively. In total, the graph of the largest dataset (SF1000) contained more than 10M vertices. The generation of the GraphX property graph for this dataset took 12 minutes. The graph is loaded in the memory cache of Spark once and used for all queries. Match sets of vertices are cleared after every query execution. For PigSPARQL, the generation of its vertical partitioned data model took 90 seconds.

Table 2. Runtimes (in s) for S2X (separated by BGP matching and generation of solution mappings) and PigSPARQL, GM = geometric mean

	SF10		**SF100**		**SF1000**	
Q	S2X	PS	S2X	PS	S2X	PS
S1	3.56, 1.78	42	7.87, 3.39	43	40.1, 23.1	57
S2	1.85, 0.63	41	3.47, 1.55	41	23.2, 5.34	46
S3	1.54, 0.42	41	3.05, 0.97	42	19.7, 2.77	44
S4	1.50, 0.54	41	3.17, 1.65	42	23.6, 10.4	46
S5	1.37, 0.38	41	2.68, 0.81	42	21.3, 2.47	44
S6	2.43, 0.94	41	3.68, 2.23	42	23.5, 9.09	46
S7	1.97, 0.92	41	3.87, 1.76	41	25.4, 9.03	46
L1	2.13, 1.16	77	5.12, 3.41	80	80.1, 122.4	87
L2	1.75, 0.73	78	4.46, 3.21	80	119.5, 215.4	83
L3	1.39, 0.35	41	2.85, 0.74	42	19.3, 2.64	46
L4	1.42, 0.23	41	2.17, 0.45	42	13.4, 0.99	44
L5	1.68, 0.75	79	4.18, 3.28	81	124.1, 224.4	82
F1	2.34, 3.04	110	4.53, 8.22	110	33.0, 19.9	115
F2	2.53, 1.16	80	5.02, 2.45	81	27.0, 9.73	83
F3	10.2, 13.5	80	57.1, 101	80	4571, 10837	92
F4	3.27, 2.70	80	7.83, 5.31	81	56.9, 58	85
F5	5.85, 4.66	80	11.9, 13	81	75.4, 97.5	99
C1	15.4, 51.5	157	34.95, 118	158	199.9, 389.6	170
C2	28.9, 49.8	237	124.7, 222	241	1217, 2652	272
C3	5.7, 1.27	42	11.6, 3.7	51	82.6, 14.9	64
GM	3.01, 1.57	64	6.73, 4.30	64.4	41.0, 18.6	71

WatDiv comes with a set of 20 predefined query templates which can be grouped in four categories according to their shape: *star* (S), *linear* (L), *snowflake* (F) and *complex* (C). For every template we instantiated five queries, yielding 100 queries per dataset. The average runtime for each query template is listed in Table 2. For S2X, we split the overall runtime in two parts: BGP matching (cf. Algorithm 1) and generation of solution mappings (cf. Algorithm 4). This way, we can separate these two parts as the output format is merely a matter of choice and we could output the matching subgraph(s) just as well. For PigSPARQL, this separation is not applicable as it is purely data-parallel without a graph abstraction.

Overall, S2X outperforms PigSPARQL by up to an order of magnitude, especially for smaller graphs. Figure 2 illustrates the geometric mean per query group, demonstrating that S2X performs best for star and linear queries. The runtimes get more closer with larger graphs which demonstrates the excellent scalability of PigSPARQL due to its underlying MapReduce framework. For star queries, we can observe that S2X computes the BGP matching pretty fast and generation of mappings consumes a considerable amount of the overall runtime, especially for S1. This can be attributed to the fact that the generation of mappings depends on the number of triple patterns in the query and S1 is the largest of the star queries. The more patterns, the more joins are needed to compute the final output. This also applies in similar manner to PigSPARQL where the general idea is to incrementally join the results of every triple pattern. It is interesting to notice that this is not true for the BGP matching part of S2X where the runtime is not primarily attributed to the size of the query but rather to its shape and the size of the matching subgraph. In S2X we match all triple patterns in one initial superstep and subsequent supersteps are used to discard invalid match candidates. This is also the reason why star queries exhibit a good performance in S2X, since in every superstep a vertex sends its match set to its neighbors which adapt their own match sets accordingly. That means, if a query defines a path of length i, it takes i supersteps to propagate this information from one vertex to the other (and another i for propagating any changes back). For star queries the path length is only one, thus less supersteps are needed.

In general, most queries scale well in S2X but for the largest graph with more than 10M vertices the runtimes of some queries increase disproportionately, especially for F3 and C2. This can be attributed to the low cluster configuration and the very large size of intermediate results of these queries which exceeded the available memory forcing Spark to spill to disk. For some other queries (e.g. L2, L5) we have noticed that a lot of time was spent for Java garbage collection which also significantly slow downs execution.

Through this comprehensive pre-evaluation, we identified some shortcomings in S2X that we should address in future work: (1) Match set sizes of vertices can be very skewed, especially for graphs with power-law degree distribution. This is also true for WatDiv where some vertices (users) can have hundreds of edges with same label (predicate) and others only a very few. Currently, we send the whole match set of a vertex to its neighbors which can lead to

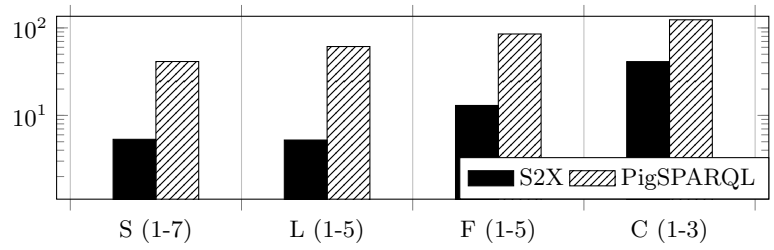

Fig. 2. GM (in s) per query group for S2X (BGP matching + generation of solution mappings) and PigSPARQL (WatDiv SF100, log scale)

very high network I/O. To save network bandwidth, we could cache the match sets of neighbor vertices and only send the changes from one superstep to the next. However, this will increase memory consumption of the property graph. Furthermore, the validation of large match sets (cf. Algorithm 2 and 3) gets expensive and stragglers can slow down the overall progress. More efficient data structures can mitigate this effect. (2) Graph partitioning in GraphX is optimized to distribute the workload for applications that are performed on the whole graph like PageRank. However, a typical SPARQL query defines a small connected subgraph pattern which can lead to highly unbalanced workloads. While this is a common problem in distributed systems, one way to address it is to adjust the partitioning to the actual task. (3) As we match all triple patterns at once in the first superstep, we generate a large overestimation of the final result. For queries that consist of many unselective (unbound subject and object) and one selective triple pattern (bound subject or object), this leads to many and large intermediate match sets where most match candidates are discarded later on. Most WatDiv queries reveal this kind of structure. In these cases, we could order the triple patterns by selectivity estimation and match only one (or some) in a superstep and send the match sets along the edges that are defined in the next triple pattern.

6 Conclusion

In this paper we introduced S2X, an RDF engine for Hadoop that combines graph-parallel with data-parallel computation to answer SPARQL queries. We defined a property graph representation of RDF for GraphX and designed a vertex-centric algorithm for BGP matching. Our experiments using a state of the art benchmark covering various query shapes illustrate that the combination of both graph-parallel and data-parallel abstraction can be beneficial compared to a purely data-parallel execution. For future work, we will address the current shortcomings of S2X that we have identified in our experiments to improve the scalability for graphs with power-law degree distribution and provide an extended evaluation including a comparison with other distributed in-memory frameworks for Hadoop.

References

1. Aluc, G., Hartig, O., Özsu, M., Daudjee, K.: Diversified Stress Testing of RDF Data Management Systems. In: ISWC. pp. 197–212 (2014)
2. Fard, A., Nisar, M., Ramaswamy, L., Miller, J., Saltz, M.: A Distributed Vertex-Centric Approach for Pattern Matching in Massive Graphs. In: IEEE Big Data. pp. 403–411 (2013)
3. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: Graph Processing in a Distributed Dataflow Framework. In: 11th USENIX OSDI '14. pp. 599–613 (2014)
4. Goodman, E.L., Grunwald, D.: Using Vertex-centric Programming Platforms to Implement SPARQL Queries on Large Graphs. In: IA3 (2014)
5. Han, M., Daudjee, K., Ammar, K., Özsu, M.T., Wang, X., Jin, T.: An Experimental Comparison of Pregel-like Graph Processing Systems. PVLDB 7(12), 1047–1058 (2014)
6. Huang, J., Abadi, D.J., Ren, K.: Scalable SPARQL Querying of Large RDF Graphs. PVLDB 4(11), 1123–1134 (2011)
7. Husain, M.F., McGlothlin, J.P., Masud, M.M., Khan, L.R., Thuraisingham, B.M.: Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. IEEE TKDE 23(9) (2011)
8. Manola, F., Miller, E., McBride, B.: RDF Primer. http://www.w3.org/TR/rdf-primer/ (2004)
9. Papailiou, N., Konstantinou, I., Tsoumakos, D., Karras, P., Koziris, N.: H2RDF+: High-performance distributed joins over large-scale RDF graphs. In: IEEE Big Data. pp. 255–263 (2013)
10. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/ (2008)
11. Schätzle, A., Przyjaciel-Zablocki, M., Hornung, T., Lausen, G.: PigSPARQL: A SPARQL Query Processing Baseline for Big Data. In: Proceedings of the ISWC 2013 Posters & Demonstrations Track. pp. 241–244 (2013)
12. Schätzle, A., Przyjaciel-Zablocki, M., Neu, A., Lausen, G.: Sempala: Interactive SPARQL Query Processing on Hadoop. In: The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference (2014)
13. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Fast and Interactive Analytics Over Hadoop Data with Spark. USENIX ;login: 34(4), 45–51 (2012)
14. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In: Proc. of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI. pp. 15–28 (2012)
15. Zeng, K., Yang, J., Wang, H., Shao, B., Wang, Z.: A distributed graph engine for web scale RDF data. In: PVLDB'13. pp. 265–276 (2013)