SPARTex: A Vertex-Centric Framework for RDF Data Analytics

Ibrahim Abdelaziz* Razen Harbi* Semih Salihoglu‡ Panos Kalnis* Nikos Mamoulis†

* KAUST, Saudi Arabia
{first.last}@kaust.edu.sa

‡ Stanford University, USA
semih@stanford.edu

† University of Ioannina, Greece
nikos@cs.uoi.gr

ABSTRACT

A growing number of applications require combining SPARQL queries with generic graph search on RDF data. However, the lack of procedural capabilities in SPARQL makes it inappropriate for graph analytics. Moreover, RDF engines focus on SPARQL query evaluation whereas graph management frameworks perform only generic graph computations. In this work, we bridge the gap by introducing SPARTex, an RDF analytics framework based on the vertex-centric computation model. In SPARTex, user-defined vertex centric programs can be invoked from SPARQL as stored procedures. SPARTex allows the execution of a pipeline of graph algorithms without the need for multiple reads/writes of input data and intermediate results. We use a cost-based optimizer for minimizing the communication cost. SPARTex evaluates queries that combine SPARQL and generic graph computations orders of magnitude faster than existing RDF engines. We demonstrate a real system prototype of SPARTex running on a local cluster using real and synthetic datasets. SPARTex has a real-time graphical user interface that allows the participants to write regular SPARQL queries, use our proposed SPARQL extension to declaratively invoke graph algorithms or combine/pipeline both SPARQL querying and generic graph analytics.

1. INTRODUCTION

SPARQL is the standard RDF query language; it allows users to express subgraph pattern matching queries. On the other hand, there is an emerging new type of RDF analytics [9, 10] that require the combination of generic graph search operations with SPARQL patterns. Examples for such operations are reachability queries, centrality analysis and community detection. For example, Qu et al. [9] filter the results of SPARQL queries by a set of graph centrality algorithms to identify the key biological entities within the resulting RDF subgraphs. Rietveld et al. [10] represent the RDF data as a directed unlabeled graph, analyze it by degree centrality and PageRank operations using a generic graph engine, re-write the result back into RDF format, and run on it SPARQL queries. In this case, SPARQL queries are evaluated against a mutated RDF graph, enriched by centrality and PageRank information for each node.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 41st International Conference on Very Large Data Bases, August 31st - September 4th 2015, Kohala Coast, Hawaii.
Proceedings of the VLDB Endowment, Vol. 8, No. 12
Copyright 2015 VLDB Endowment 2150-8097/15/08.

Putting aside the aforementioned attempts which are application-specific, state-of-the-art centralized [6, 13] and distributed [7, 2] RDF data management systems fail to support graph search operations in a principled way and they only target SPARQL queries. SPARQL itself lacks procedural capabilities; therefore, expressing graph operations using SPARQL [12] results in verbose and complex queries that are (*i*) expensive to evaluate; and (*ii*) hard to formulate, read and understand by users. This is evident in some recent works [12, 8, 5], which try to express generic graph operations using SPARQL; these approaches are limited to a small set of graph operations like clustering and graph diffusion. On the other hand, a deluge of vertex-centric graph management systems have been proposed for efficient graph analytics, like Pregel [4] and GRACE [1]. However, these systems lack the capability of evaluating ad-hoc SPARQL queries, which means that a vertex-centric program has to be written for each SPARQL query. Such an approach is tedious and requires prior knowledge about the data, in order to select the optimal query evaluation plan.

In this paper, we introduce SPARTex, a framework that supports efficient and scalable evaluation of SPARQL and graph analytics, while enabling expressing queries in an easy and natural manner. SPARTex capitalizes on two key design principles:

A unified framework. SPARTex is based on the vertex-centric computation model; hence, it inherits the scalability and abstraction of vertex-centric systems. SPARTex can be implemented on top of any system that supports computation at the vertex granularity and exchanging messages between vertices. We propose an efficient and scalable SPARQL operator on top of SPARTex. Since different operators in SPARTex require different data access views, SPARTex has a unified in-memory data store that provides different views of the same data. For example, while PageRank needs to access all outgoing edges of a vertex, SPARQL queries usually select only edges that match a specific pattern. Consequently, SPARTex supports both vertex-centric graph analytics and efficient and scalable SPARQL query evaluation.

SPARQL and User Defined Procedures. To mitigate SPARQL limitations, we propose a SPARQL extension that allows the invocation of user defined vertex-centric programs from SPARQL; such programs are modeled as defined stored procedure in SPARTex. The data store allows graph algorithms to dynamically maintain their computation results in-memory as vertex attributes. Accordingly, the output of such programs can be supplied as input for SPARQL and vice-versa in a pipelined fashion.

SPARTex introduces a new and rich type of RDF analytics that were not feasible before. (*i*) SPARQL can be used for triggering and querying generic graph algorithms. For example, a user can declaratively run PageRank and return the top-k rank values. (*ii*) Original RDF data and vertex-computed results can be combined

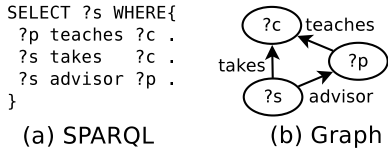

Figure 1: Q_s retrieves students taking courses taught by their advisors.

as a subgraph pattern in SPARQL. In other words, triple patterns of SPARQL queries can be RDF data or vertex-computed derived data. (*iii*) Generic graph algorithms and SPARQL query patterns can be pipelined so the output of one operator is the input to another. For example, the Single Source Shortest Path (SSSP) algorithm can start from the vertices that match a specific SPARQL pattern.

2. RDF ANALYTICS FRAMEWORK

RDF data is a set of ⟨`subject`, $predicate$, `object`⟩ triples that form a directed labeled graph. SPARQL is the standard query language for RDF. Queries consist of a set of RDF triple patterns, where some of the columns are variables. For example, Q_s in Figure 1(a) retrieves all students who take courses taught by their advisors. The query corresponds to the graph in Figure 1(b). The answer is the set of bindings of $?s$, $?c$ and $?p$ that render the query graph isomorphic to subgraphs in the data. SPARQL is restricted to pattern-matching and does not support advanced graph analytics (such as shortest path search, PageRank and centrality computation). Accordingly, current RDF data management systems are not optimized for graph analytics beyond SPARQL.

2.1 SPARQL Extension

In SPARTex, we introduce a SPARQL extension that allows the invocation of user defined stored procedures. Assume that a user is capable of writing and storing his own procedures in a given system; e.g. Pregel programs. For a procedure called *proc* which is located at *path*, the user can call *proc* in SPARTex as:

```
PREFIX  prefix: path
CALL prefix:proc(list[parms]) AS list[properties]
```

$list[parms]$ is a list of parameters that the procedure expects while $list[properties]$ is the list of vertex properties that $proc$ will add to the RDF data. For example, the PageRank algorithm expects the maximum number of computation iterations and introduces a $pRank$ property per vertex. PageRank can be invoked as follows:

```
PREFIX  algo: <file://path_to_algorithms>
PREFIX  sptx: <http://www.spartex.com/analytics/>
CALL algo:PageRank(max_iter) AS sptx:pRank
```

PageRank in the previous example runs on the entire RDF graph. However, we may want to apply them to only a subset of the graph. For this, we introduce constructs that filter the RDF graph based on vertices and edges. Invoked procedures are optionally associated with one or more filters.

```
FILTER_VERTEX AS filter_name WHERE { BGP }
FILTER_EDGE AS filter_name WHERE { BGP }
```

All triple patterns of the basic graph pattern BGP in the WHERE clause must have a common vertex. In other words, BGP is a star query around a specific vertex. For FILTER_VERTEX, vertices that do not match BGP are filtered out. Similarly, all edges that do not satisfy the BGP pattern of FILTER_EDGE are filtered out. For example, we can exclude objects of triples with $rdf : type$ predicates from the PageRank algorithm as follows:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-sy
 ntax-ns#>
FILTER_EDGE AS no_type WHERE {
  ?s ?p ?o .
  FILTER(!sameterm(?p, rdf:type))
}
CALL algo:PageRank(max_iter) USING no_type AS
      sptx:pRank
```

So far, we have seen vertex properties set or deleted by stored procedures. However, users may want to deliberately set or delete some vertex properties. Therefore, we introduce two constructs for adding and deleting vertex properties.

```
ADD PROPERTY {list[property patterns]} WHERE {BGP}
DROP PROPERTY {list[property patterns]} WHERE {BGP}
```

BGP in the optional WHERE clause can be an arbitrary pattern; e.g., the following drops pRank properties smaller than a threshold:

```
DROP PROPERTY {?x sptx:pRank ?val} WHERE{
        ?x sptx:pRank ?rank .
        FILTER(?rank < threshold)
}
```

2.2 Use Cases

Combining the expressiveness of SPARQL with generic graph computations opens opportunities for deeper RDF data analysis. We now discuss use cases that are not feasible without this extension.

2.2.1 Combining SPARQL and graph properties

Consider Q_s in Figure 1 and assume we want to filter students based on whether their advisors are popular and teach core courses. Assume that PageRank and centrality properties are used to measure popularity of professors and importance of courses, respectively. If both PageRank and centrality algorithms are available as user-defined stored procedures, Q_s can be rewritten as:

```
PREFIX  sptx: <http://www.spartex.com/analytics/>
CALL algo:centrality() AS sptx:centrality
CALL algo:PageRank(max_iter) AS sptx:pRank
SELECT ?s WHERE {
    ?p    teaches   ?c .
    ?s    takes     ?c .
    ?s    advisor   ?p .
    ?p    sptx:pRank         ?rank .
    ?c    sptx:centrality    ?cent .
    FILTER (?rank > val1 && ?cent > val2)
}
```

Note that the query has two types of triple patterns; the first one comes from the structure of the input graph, while the second is derived from the vertex-computed values. Without the proposed extension, such a query would not be easy to express.

2.2.2 Combining SPARQL and analytics

We now demonstrate a use case where the results of SPARQL are used in graph analytics. Consider Q_s in the previous example and suppose we want to find the shortest path between popular professors that match the pattern and every other vertex in the graph. This can be done by executing the Single Source Shortest Path (SSSP) algorithm starting from these professors as follows.

```
PREFIX  sptx: <http://www.spartex.com/analytics/>
CALL algo:centrality() AS sptx:centrality
CALL algo:PageRank(max_iter) AS sptx:pRank
ADD PROPERTY {?p sptx:popular "1" . } WHERE {
    ?p    teaches   ?c .
    ?s    takes     ?c .
    ?s    advisor   ?p .
    ?p    sptx:pRank         ?rank .
    ?c    sptx:centrality    ?cent .
    FILTER (?rank > val1 && ?cent > val2)
}
```

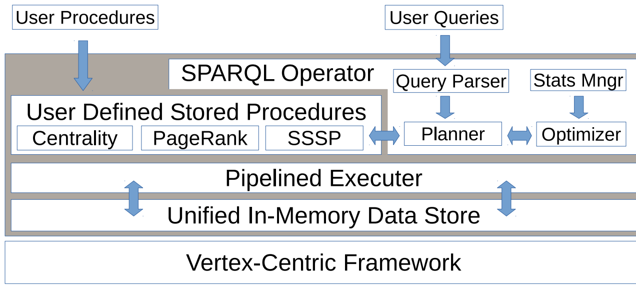

Figure 2: SPARTex Architecture.

```
FILTER_VERTEX AS start WHERE {
  ?p sptx:popular "1" .
}
CALL algo:SSSP() USING start AS sptx:sssp
```

Using the ADD PROPERTY construct, we identify the popular professors by setting their $sptx : popular$ property as 1. Then, we create a vertex filter to exclude all vertices without this property. Finally, the filter is associated with the SSSP procedure call so it only starts from vertices defined in the filter.

2.3 SPARTex Architecture

SPARTex is an RDF analytics framework that supports declarative SPARQL queries as well as procedural graph algorithms, based on the discussed SPARQL extension. SPARTex can call user-defined stored procedures written as vertex-centric programs. These procedures can be executed in a pipelined fashion and can dynamically maintain their computation results in memory as vertex attributes. Subsequent operators can use these results if needed. An overview of SPARTex is depicted in Figure 2. In the rest of this section, we detail each of the different components of our framework.

2.3.1 *Vertex-centric Framework*

Vertex-centric computation frameworks are built on top of distributed file systems for data persistence, such as the input graph, check-pointing files, and computation outputs. The user defines a generic function, which is executed on each vertex independently. Graph vertices interact with each other through message passing. In SPARTex, any vertex-centric program is considered as a user defined stored procedure.

2.3.2 *Unified In-Memory Data Store*

SPARTex stores $G = (V, E, F_E, L_E, F_V, L_V, \Psi)$. G a directed, labeled and vertex-attributed graph. V and E are the set of vertices and edges in G, respectively. $e(u, v) \in E$ denotes a directed edge from u to v. L_E is the set of possible edge labels while $F_E : E \rightarrow L_E$ is a labeling function that assigns a label to each edge. L_V is the set of possible vertex labels and $F_V : V \rightarrow L_V$ assigns a unique label for each vertex. Ψ defines a set of properties that can be associated to a vertex. Each vertex $v \in V$ is associated with a set of m properties $(p_1(v), p_2(v), ...p_m(v))$ such that $p_i(v)$ denotes the value of the property p_i for v.

Our graph model is generic and can be easily adopted for RDF data. We use the labels of subjects, predicates, and objects as their unique identifiers. [1] Therefore, V is the the set of subjects and objects, while E is the set of edges defined by the input triples. L_E is the set of unique predicate labels. Ψ is initially empty; however, during execution a vertex can be given one or more properties.

[1]The labels of subjects are uniform resource identifiers (URIs) or blank nodes, predicates are represented by URIs, while objects can be URIs, blank nodes or literals.

While SPARQL queries access incoming and outgoing edges of a vertex using predicate labels, graph algorithms like PageRank may disregard labels and access all outgoing edges. Therefore, rich RDF analytics requires modeling the data in a uniform way, while providing different data access methods. Specifically, our framework supports: (*i*) *label-based* data access used for SPARQL query evaluation; (*ii*) *label-oblivious* data access used by graph algorithms; and (*iii*) dynamically adding, deleting, or updating *vertex properties*. SPARTex also supports data filtering constraints. Computations in vertex-centric frameworks are done at the vertex granularity; thus, our unified in-memory data store supports the aforementioned types of data accesses via the following indices:

1. Miniature RDF Data Index: For each vertex, we maintain: (*i*) a Predicate-Object (PO) Index which, given an edge predicate p, returns all outgoing neighbors (objects) from edges labeled p; and (*ii*) a Predicate-Subject (PS) Index which, given an edge predicate p, returns all incoming neighbors (subjects) from edges labeled p.

2. Miniature In-Memory Store: each vertex maintains an in-memory key-value store where algorithms can delete, add or update a vertex property.

These indices are accessed through a set of API calls. Filtering constraints, that are associated with procedures function calls, are registered in the unified data store; only data that satisfy the filtering constraints are retrieved or modified.

2.3.3 *SPARQL Operator*

The SPARQL operator evaluates RDF rich data analysis tasks. The parser is responsible for preprocessing incoming queries. It segregates the analytics constructs from the pattern matching of SPARQL. The subgraph pattern is then converted into a graph representation. Afterwards, both the analytics constructs and the pattern graph are passed to the query planner. The query planner checks the existence of the called procedures and the consistency of their parameters. Then, it sends the pattern graph to the query optimizer to compile the query execution plan. The query optimizer utilizes global statistics maintained in the statistics manager. The planner consolidates the generic analytics part and the optimized pattern matching query plan into a global pipelined execution plan. Finally, the plan is evaluated by the pipelined executer.

3. DEMONSTRATION

We implemented SPARTex on top of GPS [11], an open-source Pregel clone. SPARTex is deployed on a cluster of 12 machines each with 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs. The machines run 64-bit 3.2.0-38 Linux Kernel and connected by a 10Gbps Ethernet switch. We plan to conduct the demonstration with remote access to this cluster which will be accessible from the conference site. As the amount of data communicated between the GUI and the cluster is minimal (queries and final results), we are not expecting significant delays during the demonstration. Using the synthetic LUBM[2] benchmark, we have generated LUBM-4000 dataset which contains 534 million triples. Besides, we also used YAGO2[3], a real dataset consisting of 295 million triples.

3.1 Audience Interaction

Figure 3 shows the GUI interface of SPARTex. Using the GUI, we offer two different interaction scenarios:

SPARQL Query Evaluation: Participants can select a dataset against which they can execute either manually written SPARQL

[2]http://swat.cse.lehigh.edu/projects/lubm/
[3]www.mpi-inf.mpg.de/yago/

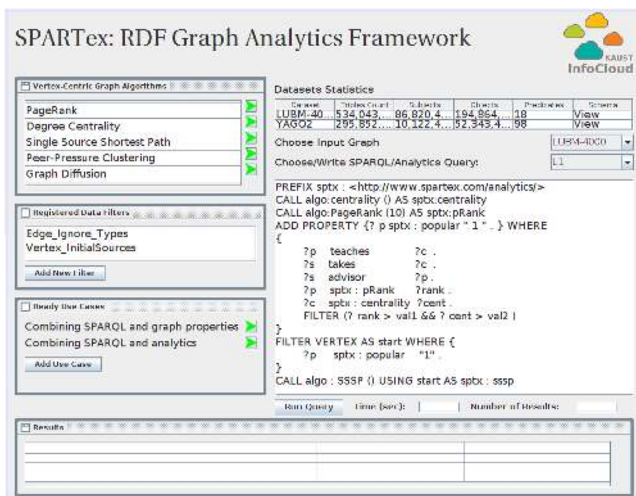

Figure 3: SPARTex Graphical Interface

queries or choose one from the predefined benchmark queries. Predefined queries for YAGO2 dataset are those defined in [7] whereas LUBM queries are the same queries used in [2]. The predefined queries provide a mixture of queries with varying structural characteristics and selectivity classes. Once the query and the dataset are specified, SPARTex evaluates the query and displays the results as well as the total query response time.

RDF Graph Analytics: Participants can run vertex-centric graph algorithms on a dataset of their choice. We provide a wide range of graph algorithms from which participants can choose. In particular, the following algorithms are available: PageRank, Degree Centrality, SSSP, Peer-Pressure clustering and Graph Diffusion.

Using the set of supported algorithms and our extension, participants are allowed to run different algorithms on the dataset they choose. Moreover, they can pipeline SPARQL queries and graph analytics as needed. For example, on YAGO2 dataset, participant can evaluate many possible use cases, like: (*i*) using our filtering construct, a participant can define a filter that limit PageRank evaluation to only vertices that have the property $hasPage$. (*ii*) Write a SPARQL query that retrieves the vertices with the highest PageRank values. (*iii*) Cluster the entire YAGO2 graph and retrieve the clusters information. (*iv*) Managing (add/delete) vertex properties. (*v*) Run SSSP from vertices that belong to certain cluster or has certain property value. All these use cases are defined and participants can visualize its query syntax and and modify it if needed. Moreover, they can also define and evaluate other use cases.

3.2 Evaluation Results

We evaluate the two use cases described in Section 2.2 using SPARTex and LUBM-4000 dataset. Since no other system can fully support these use cases, we use combinations of SPARQL engines and analytics systems. We use H2RDF+ [7] as SPARQL engine with two different analytics systems. The first combination is H2RDF+ with PEGASUS [3], a graph mining library on top of MapReduce. The second combination uses H2RDF+ with GPS [11]. In both cases, data need to be moved between multiple systems and formatted accordingly. Figure 4 shows the execution time for both use cases. In the first case, graph analytics is performed prior to query evaluation. SPARTex performs better than H2RDF+GPS because it maintains the computation results in its in-memory store. Therefore, no data formatting or re-indexing is required. GPS on the other hand needs to output the results to

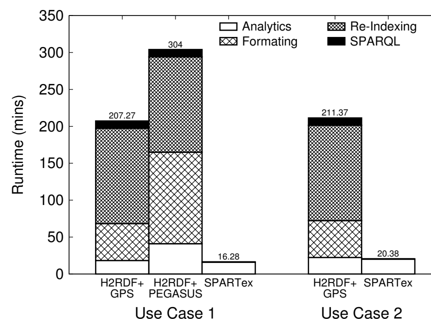

Figure 4: Rich RDF Analytics.

HDFS for further processing in H2RDF+. PEGASUS, performs worse than GPS confirming that MapReduce approaches do not perform well for graph analytics. Both H2RDF+PEGASUS and H2RDF+GPS require data formatting and re-indexing by H2RDF+ before evaluating SPARQL queries. The cost of data formatting and indexing is very substantial accounting for more than 80% of the processing time. Finally, when evaluating SPARQL queries, SPARTex performs significantly better than H2RDF+. The same applies on the second use case; however, since the SSSP algorithm is not available in PEGASUS, we only compare to H2RDF+GPS.

4. REFERENCES

[1] G. Wang, W. Xie, A. Demers and J. Gehrke. Asynchronous Large-Scale Graph Processing Made Easy. In *CIDR*, 2013.

[2] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. TriAD: A Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *SIGMOD*, 2014.

[3] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM*, 2009.

[4] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a System for Large-scale Graph Processing. In *SIGMOD*, 2010.

[5] D. Mizell, K. J. Maschhoff, and S. P. Reinhardt. Extending SPARQL with graph functions. In *IEEE Big Data*, 2014.

[6] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1), 2010.

[7] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras, and N. Koziris. H2rdf+: High-performance distributed joins over large-scale rdf graphs. In *IEEE BigData*, 2013.

[8] L. Qi, H. Lin, and V. Honavar. Clustering remote RDF data using SPARQL update queries. In *ICDEW*. IEEE, 2013.

[9] X. Qu, R. Gudivada, A. Jegga, E. Neumann, and B. Aronow. Inferring novel disease indications for known drugs by semantically linking drug action and disease mechanism relationships. *BMC bioinformatics*, 10(Suppl 5):S4, 2009.

[10] L. Rietveld, R. Hoekstra, S. Schlobach, and C. Guéret. Structural Properties as Proxy for Semantic Relevance in RDF Graph Sampling. In *ISWC*, 2014.

[11] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *SSDBM*, 2013.

[12] R. W. Techentin, B. K. Gilbert, A. Lugowski, K. Deweese, J. R. Gilbert, E. Dull, M. Hinchey, and S. P. Reinhardt. Implementing Iterative Algorithms with SPARQL. In *EDBT/ICDT Workshops*, 2014.

[13] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. TripleBit: A Fast and Compact System for Large Scale RDF Data. *PVLDB*, 6(7), 2013.