Combining Vertex-centric Graph Processing with SPARQL for Large-scale RDF Data Analytics

Ibrahim Abdelaziz, Razen Harbi, Semih Salihoglu and Panos Kalnis

Abstract—Modern applications require sophisticated analytics on RDF graphs that combine structural queries with generic graph computations. Existing systems support either declarative SPARQL queries, or generic graph processing, but not both. We bridge the gap by introducing Spartex, a versatile framework for complex RDF analytics. Spartex extends SPARQL to combine seamlessly generic graph algorithms (e.g., PageRank, Shortest Paths, etc.) with SPARQL queries. Spartex builds on existing vertex-centric graph processing frameworks, such as Graphlab or Pregel. It implements a generic SPARQL operator as a vertex-centric program that interprets SPARQL queries and executes them efficiently using a built-in optimizer. In addition, any graph algorithm implemented in the underlying vertex-centric framework, can be executed in Spartex. We present various scenarios where our framework simplifies significantly the implementation of complex RDF data analytics programs. We demonstrate that Spartex scales to datasets with billions of edges, and show that our core SPARQL engine is at least as fast as the state-of-the-art specialized RDF engines. For complex analytical tasks that combine generic graph processing with SPARQL, Spartex is at least an order of magnitude faster than existing alternatives.

Index Terms—RDF data, Graph Analytics, SPARQL, Vertex-centric.

1 Introduction

RDF data are collections of ⟨`subject`, *predicate*, `object`⟩ triples that form directed labeled graphs; an example is shown in Figure 1. Due to its versatility, many communities adopt the RDF model for publishing their data. For example, knowledge bases like PubChemRDF[1] and Bio2RDF[2] contain billions of facts in RDF. Traditionally, RDF data are searched for subgraphs that match specific patterns. Such queries are expressed in SPARQL[3] and consist of a set of RDF triple patterns, where some of the columns are variables. For example, query Q_a in Figure 2(a) retrieves all the advisors of students who take the Databases course. The query corresponds to the graph pattern in Figure 2(b). The answer is the set of bindings of $?stud$ and $?prof$ that render the query graph isomorphic to subgraphs in the data. In Figure 1, the answer of Q_a is `(?stud, ?prof)` $\in \{$`(John, Fred)`, `(Ben, Lisa)`$\}$.

The emerging trend for RDF analytics [1], [2], [3], [4] goes beyond simple pattern matching, and requires complex pipelines that combine a variety of graph algorithms with SPARQL. For example, Qu et al. [1] mine biological data to identify new targets for existing drugs. First, they run a set of SPARQL queries against biological databases. On the results they execute graph centrality algorithms to identify key biological entities. Rietveld et al. [2] focus on advanced graph sampling. First, they compute the degree centrality and PageRank of the input graph using a generic graph engine, and generate an RDF dataset enriched by centrality and PageRank information for each vertex. Then, they run SPARQL queries against the intermediate RDF graph.

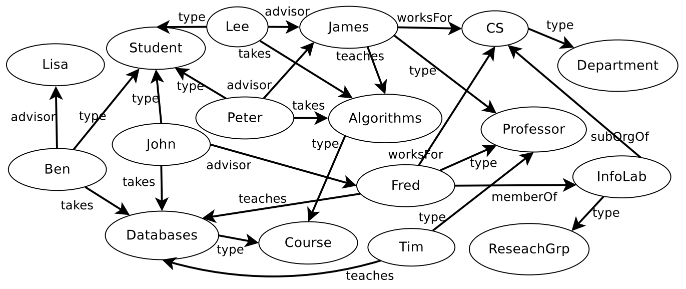
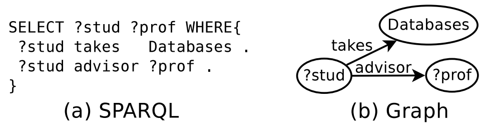


Fig. 1. Example RDF graph consisting of interconnected triples like ⟨`Ben`, *takes*, `Databases`⟩.



```
SELECT ?stud ?prof WHERE{
 ?stud takes   Databases .
 ?stud advisor ?prof .
}
```

Fig. 2. Q_a finds the advisors of the students who take Databases.

The state-of-the-art centralized [5], [6], [7] and distributed [8], [9], [10], [11], [12] RDF data management systems fail to support generic graph processing as they focus only on SPARQL, which lacks procedural capabilities. Expressing graph algorithms using SPARQL results in verbose and complex queries that are hard to formulate and expensive to evaluate [13], [14], [15]. On the other hand, vertex-centric graph processing systems, such as Pregel [16], PowerGraph [17] and GRACE [18], support graph analytics, but lack built-in SPARQL engines, requiring users to hard-

- *I. Abdelaziz and P. Kalnis are with the King Abdullah University of Science and Technology (KAUST), Saudi Arabia.*
  E-mail: {ibrahim.abdelaziz, panos.kalnis}@kaust.edu.sa
- *R. Harbi is with Saudi Aramco, Saudi Arabia.*
  E-mail: {razen.harbi}@aramco.com
- *S. Salihoglu is with the University of Waterloo, Canada.*
  E-mail: semih.salihoglu@uwaterloo.ca

1. http://pubchem.ncbi.nlm.nih.gov/rdf
2. http://bio2rdf.org
3. http://www.w3.org/TR/rdf-sparql-query

code and manually optimize any pattern matching query.

To address the above problems we propose Spartex, a distributed framework for rich RDF data analytics. Spartex allows users to write queries that combine declarative SPARQL with procedural code for generic graph processing. The advantages of Spartex include: (*i*) increased productivity: Our framework simplifies the implementation of complex RDF analysis pipelines, by allowing seamless integration of SPARQL and user defined procedures within the same programming environment, and effortless information exchange between them. (*ii*) Speed: By eliminating the need to move data among independent processing systems, complex analytical tasks are executed at least an order of magnitude faster in our framework. Even for pure SPARQL queries, Spartex is as fast as dedicated RDF engines, due to its sophisticated query optimizer. (*iii*) Versatility: Spartex can be implemented on a variety of vertex-centric graph engines, including Graphlab, Giraph, Pregel, Spark GraphX, and others. (*iv*) Scalability: By taking advantage of the underlying graph engine, Spartex can scale to very large graphs and many compute nodes. (*v*) Community support: There is a thriving community of developers for graph analysis algorithms (e.g., clustering, centrality, mining, machine learning, etc) using the vertex-centric paradigm; the resulting libraries can be used in Spartex.

We realize Spartex in three steps. First, we define an extension of SPARQL. Then, we implement a generic SPARQL query engine as a vertex-centric program. Finally, we extend the in-memory graph representation of the underlying vertex-centric engine. We next give overviews of each step:

SPARQL Extension: Spartex is inspired by the database community, where the coupling of SQL code with a generic programming language (e.g., Java, C++) is common. Spartex defines a Graph Analytics extension of SPARQL that allows generic user-defined procedures (UDPs) to be intermixed with SPARQL. A UDP can be any program implemented in the vertex-centric model (e.g., PageRank, Shortest-Paths, Centrality). UDPs communicate with SPARQL by updating the RDF graph. Spartex also allows filters that limit the scope of the UDP, where the filter condition is a separate SPARQL query.

SPARQL Engine: We implement an efficient SPARQL query engine as a vertex-centric program, allowing UDPs and SPARQL queries to run on top of the same vertex-centric framework. Our SPARQL operator leverages the message-passing nature of the vertex-centric frameworks for join evaluation. Given a SPARQL query Q, our SPARQL operator has two stages: (*i*) a cost-based optimizer picks a *trail* on Q to minimize the number of messages. (*ii*) Vertices exchange messages with their neighbors along the selected trail for several rounds, until the query is solved.

In-memory Data Store: The underlying vertex-centric frameworks typically store the input as a generic graph in memory. Spartex extends this with a per-vertex data store that is tailored to RDF data. Our data store efficiently filters the neighbors of vertices by specific predicates. The data store also allows updates on the RDF graph.

Consequently, Spartex introduces novel ways to implement RDF analytics that were not feasible before: (*i*) Results of generic graph algorithms are represented as new triples; therefore, original RDF data and vertex computed values can be combined in a SPARQL query. (*ii*) Generic graph algorithms and SPARQL queries can be pipelined so that the output of one operator is the input to another; for example, the Single Source Shortest Path algorithm can start from the vertices that match a specific SPARQL pattern. (*iii*) The entire analysis is executed within our framework; there is no need to use different systems or materialize and reformat intermediate results. In summary, our contributions are:

- We propose Spartex, an RDF analytics framework that allows SPARQL queries to be combined with generic graph algorithms. We demonstrate various scenarios where our framework simplifies the implementation of sophisticated RDF analytics programs.
- At the semantic level, Spartex extends SPARQL (Section 2), allowing generic UDPs to be intermixed with SPARQL queries. At the systems level, Spartex implements a complete SPARQL query engine (Section 4) with a cost-based optimizer and an in-memory miniature data store.
- We evaluate Spartex using real and synthetic datasets with billions of triples on a cluster of 288 CPU cores (Section 5). We show that our SPARQL engine is at least as fast as the state-of-the-art dedicated RDF systems. For complex analytical tasks that combine generic graph processing with SPARQL, we show that Spartex is at least 10 times faster that existing alternatives.

2 Graph Analytics Extension

Our SPARQL extension for graph analytics enables two important functionalities: (*i*) users can write their own algorithms in a procedural language and invoke them from within SPARQL as user-defined procedures. (*ii*) Users can mutate the original RDF graph by adding/deleting triples. In particular, the output of UDPs can be materialized as new triples in the RDF graph. These triples can then be used in a SPARQL query or as input to another graph algorithm.

2.1 New Spartex Constructs

Below we introduce the new Spartex constructs that enable the invocation of UDPs and information exchange between UDPs and SPARQL.

2.1.1 Calling UDPs

The first construct allows users to call an already implemented user-defined procedure (UDP), as follows:

```
CALL proc(list[params]) AS list[predicates]
```

The above constraruct calls UDP *proc*. list[params] is the set of input parameters, while list[predicates] is the set of vertex predicates that *proc* will add to the RDF graph; this is effectively the output of the UDP. As an example, the following code computes and stores the PageRank value of each vertex in an RDF dataset:

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

Prefix defines the URI of the stored procedure. The input of PageRank is the maximum number of iterations. For each vertex v, the algorithm adds to the RDF graph a new triple $\langle \texttt{v}, \textit{sptx:pRank}, \texttt{rank} \rangle$ where *sptx:pRank* is the property name and $rank$ is the PageRank value of v. The resulting triples can be used later within a SPARQL query or as input to other algorithms (see Section 2.2).

2.1.2 Data Filters

In the previous example, the PageRank algorithm runs on the entire RDF graph. However, there are cases where an algorithm should run only on a subset of the graph. We introduce filtering constructs based on the vertices and edges of the RDF graph. Invoked procedures are optionally associated with one or more filters:

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

The Basic Graph Pattern (BGP) used with data filters is a star query around a common vertex[4]. For FILTER_VERTEX, vertices that do not match the BGP are filtered out. Similarly, all edges that do not satisfy the BGP pattern of FILTER_EDGE are filtered out. Filters are passed to UDPs through keyword *using*. For example, to exclude the triples with predicate *rdf:type* from the PageRank computation, we define the following filter and use it in PageRank:

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

Users can specify multiple filters to define the subset of the graph the UDP is applied on. Notice that our data filter construct is different from SPARQL filters. The later are used to filter the results of a SPARQL query, whereas our data filter is coupled with the invocation of a graph algorithm to temporary exclude part of the RDF graph before running the algorithm.

2.1.3 Graph Updates

We mentioned how a UDP can update the RDF graph. However, users may want to explicitly insert or delete some triples. This can be done as follows:

```
ADD TRIPLE {list[triple patterns]} WHERE {BGP}
DROP TRIPLE {list[triple patterns]} WHERE {BGP}
```

For example, the following code drops all triples with predicate *sptx:pRank* and object (i.e., rank value) less than a threshold:

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

4. More sophisticated filtering can be achieved by combining FILTER and ADD TRIPLE (see Section 2.2.2)

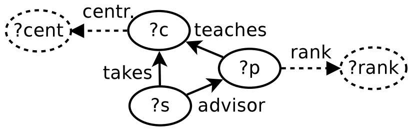


Fig. 3. Example query Q_s. Solid lines are part of the original graph while dotted lines represent triples that needs to be computed first.

2.2 Case Studies of RDF Analytics

We present three case studies that demonstrate how our SPARQL extensions simplify the implementation of complex RDF analytics applications.

2.2.1 Using Graph Analytics Output in SPARQL

Consider query Q_s in Figure 3; it returns the set of students who take courses taught by their advisors. Assume we want to restrict the query results to only popular professors and important courses. For the sake of the example, let PageRank and centrality indicate professor popularity and course importance, respectively. PageRank and centrality values do not exist in the original data, so they will be computed by UDPs; the corresponding triples are depicted with dotted lines in the figure. Q_s can be expressed as:

```
PREFIX sptx: <http://www.spartex.com/analytics/>
CALL com.sptx.algo.centrality() AS sptx:centrality
CALL com.sptx.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)
}
```

Spartex executes the centrality and PageRank algorithms; their output for each vertex is added as new triples to the RDF graph. Then, Spartex executes the subgraph pattern matching part of the query, expressed in SPARQL; only vertices the satisfy the filter constraints are retrieved.

2.2.2 Using SPARQL Output in Graph Analytics

The results of SPARQL can be used in subsequent general graph algorithms. Consider again Q_s and assume we want to find the shortest path between popular professors who teach important courses and every other vertex in the graph. This can be done by executing the Single Source Shortest Path (SSSP) algorithm starting from those professors, as follows:

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

First, we identify the qualifying professors and add in the corresponding vertices $?p$, a Boolean flag T in the form of a triple ⟨`?p`, *sptx:popular*, `"T"`⟩. We create a filter to retrieve those professors who are connected with the Boolean flag, and use the filter in the SSSP procedure call, so the algorithm starts from vertices that match the filter.

2.2.3 Sampling RDF Graphs

SamplD [2] is a pipeline for sampling RDF graphs. Given an RDF graph, SamplD applies a set of graph operations using Apache Giraph [19] and PIG [20]. It transforms the input into a directed unlabeled graph and computes degree centrality and PageRank in the transformed graph. Each triple is assigned a score and triples with the highest scores are selected as the graph sample. The SamplD pipeline requires the movement of the input and intermediate result thought multiple programming platforms. We demonstrate how we implement the same pipeline entirely within Spartex:

```
CALL com.sptx.algo.centrality() AS sptx:centrality
CALL com.sptx.algo.PageRank(max_iter) AS sptx:pRank
CALL com.sptx.algo.SamplDRankTriples()
```

The entire pipeline can be reduced to three simple UDP calls. Our approach avoids data movement and increases productivity by requiring from the user to learn only one programming environment.

3 System Architecture

Our Spartex prototype is built on top of GPS [21], an open-source Pregel clone. However, Spartex supports a variety of distributed vertex-centric bulk synchronous graph processing frameworks, such as Pregel [16], GraphLab [22] and Trinity [23]. In such frameworks, users define a generic compute function to be executed on each vertex independently. Vertices interact with each other through messages. A typical vertex-centric program consists of a number of iterations. In each iteration a vertex can perform computation, change its state and send messages to its neighbors. Typically, vertex-centric frameworks are coupled with a distributed file system, e.g., HDFS.

An overview of Spartex is depicted in Figure 4. Spartex follows a master-slave architecture. Users write vertex-centric programs for generic graph processing, compile and add them to the classpath. Spartex treats these programs as user-defined stored procedures. In addition, Spartex provides an efficient vertex-centric SPARQL operator (Section 4). Having both SPARQL and graph algorithms within the same framework, Spartex allows both operations to be executed in a pipelined fashion. In the rest of this section we describe each component of Spartex.

3.1 Master

The master is not assigned any portion of the input graph; rather it orchestrates the workers' activity. Users submit their program to the master, which parses it and generates an execution plan for the entire pipeline of the generic algorithms and SPARQL queries. Then the master asks the workers to execute each step of the plan. Below, we define each of the master's components:

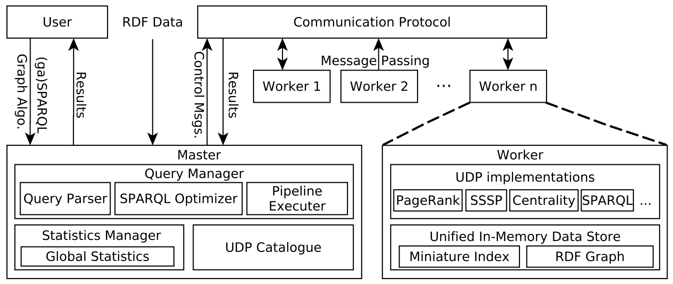


Fig. 4. Spartex Architecture.

Query Manager. It is responsible for parsing, optimizing and executing user programs. The *Query Parser* separates the procedural constructs from the declarative SPARQL patterns. It checks the existence of the called UDPs and the consistency of their parameters. Then, our *SPARQL optimizer* enumerates possible execution plans for the SPARQL part, estimates their costs using global statistics and generates an optimized plan. The *Pipeline Executer* receives a global pipelined execution plan consisting of the consolidated procedural and optimized SPARQL plan. It dictates which vertex-centric program to run for how many iterations or until the program converges. After each step, the executer initializes the next vertex-centric process to be executed, until the plan is completed.

Statistics Manager. The master gathers global statistics during the RDF graph loading phase. First, each worker loads and indexes its assigned vertices and their edges. Next, vertices report statistics about their neighbours and predicates, as well as the correlation between different predicates (refer to Section 4.2.2 for details). Finally, each worker synchronizes its collected statistics with the master. The master aggregates all statistics in a global structure.

UDP Catalogue. It contains meta-data about user-defined procedures that are registered with Spartex.

3.2 Worker

Vertex-centric frameworks divide the input graph into partitions. A vertex v with its outgoing edges is assigned to worker W based on a hashing scheme $W_{(v \ mod \ k)}$, where k is the number of partitions. Spartex slightly modifies the default partitioning scheme such that both the incoming and outgoing edges of each vertex v are assigned to the same worker; this is equivalent to partitioning on both subject and object vertices. In Figure 1, assume that vertices Ben and $Databases$ are assigned ids of 0 and 1, respectively and we have only two workers ($k = 2$). This means that vertex Ben along with its edges (type, student), (advisor, Lisa) and (takes, Databases) will be assigned to worker W_0 (= 0 mod 2). Similarly, vertex $Databases$ along with all its incoming and outgoing edges will be assigned to worker W_1 (=1 mod 2). Spartex works with any *disjoint* vertex partitioning scheme, i.e., schemes that assign each graph vertex to a single partition with all its incoming and outgoing edges. Furthermore, it can work with any a sophisticated vertex partitioning scheme that produces higher quality partitions and reduces the communication cost; the choice of partitioning scheme is orthogonal to our work.

Unified In-Memory Data Store. Generic graph algorithms and SPARQL access data differently. While SPARQL needs to access both incoming and outgoing edges using predicate labels, algorithms like PageRank need to access the outgoing edges only regardless of their labels. Spartex models the data in a uniform way while providing different data access methods. Specifically, our framework supports: (*i*) *label-based* neighbour access used for SPARQL query evaluation; and (*ii*) *label-oblivious* neighbour access used by most generic graph algorithms. Since computation is done at the vertex granularity, a set of miniature data indices per vertex are created. These indices are accessed through a set of API calls.

Miniature RDF Index. As workers parse their assigned partitions, they create a per-vertex (miniature) data store tailored to RDF data, which consists of the following: (*i*) Predicate-Object (PO) index: given an edge predicate p, PO index returns a list of all outgoing neighbors (objects). (*ii*) Predicate-Subject (PS) index: given an edge predicate p, PS index returns a list of all incoming neighbors (subjects). For label-oblivious access, our API allows to access these indices without providing a predicate. In this case, it returns all incoming or outgoing neighbours of the vertex regardless of the predicate label. Notice that edge labels are assigned to the same partition on which their corresponding vertices exist. Therefore, label-based and label-oblivious data access in Spartex is independent of the partitioning method. It is just an abstraction that allows different graph algorithms to access the data as needed.

Miniature Properties Store. Newly added triples are added to this store as vertex properties. Since these triples are updated frequently and to avoid imposing overhead on the PS and PO indices, we employ a separate store per vertex. Each vertex maintains an in-memory key-value store where different algorithms can delete, add or update a vertex property. This way the result of one algorithm can be read by others; enabling pipelined execution of graph algorithms.

UDP Implementations. It contains the same meta-data stored at the UDP specifications structure in the master. However, it also contains the actual implementation of the registered vertex-centric programs (e.g., PageRank, SSSP, etc). It is used by the worker to switch between different UDPs at runtime when directed by the master.

4 SPARQL Query Engine

In this section, we present our vertex-centric SPARQL operator. Consider query $\overline{Q_s}$ defined by the solid lines in query Q_s (Figure 3). $\overline{Q_s}$ consists of 3 triple patterns: q_1 : $\langle$?p, $teaches$, ?c$\rangle$, q_2 : $\langle$?s, $advisor$, ?p$\rangle$ and q_3 : $\langle$?s, $takes$, ?c$\rangle$. In the relational model, $\overline{Q_s}$ is answered[5] by scanning the data to find the matches of each triple pattern. Then, the intermediate results are joined to formulate the final answers. However, relational approaches are not suitable for Spartex because data, computation and communication are all vertex-centric. Instead, we employ graph exploration and use inter-vertex message passing for query evaluation.

5. In this example, a bushy execution plan is assumed.

Algorithm 1: ExploreEdge

```
Input: ExplorationEdge $\bar{e}$, MessageList $ml$, Iteration $i$
$eVertex \leftarrow \bar{e}$.expVertex;
$tVertex \leftarrow \bar{e}$.termVertex; $eDirection \leftarrow \bar{e}$.direction;
$ePredicate \leftarrow \bar{e}$.predicate;
$vertexSubQueries \leftarrow$ getVertexSubqueries($eVertex$);
if Matches($vertexSubQueries$, $eVertex$) then
    $neighbors \leftarrow$ Empty;
    if $eDirection$ is Outgoing then
        $neighbors \leftarrow$ PO[$ePredicate$];
    else
        $neighbors \leftarrow$ PS[$ePredicate$];
    if $i$ = 1 then
        $msg \leftarrow$ Empty;
        $msg[eVertex] \leftarrow vertexID$;
        sendMessageToAll($msg$, $neighbors$);
    else
        if isQueryVertexVisited($\bar{e}$.termVertex) then
            foreach $msg$ in $ml$ do
                if $msg[tVertex] \in neighbors$ then
                    $msg[eVertex] \leftarrow = vertexID$;
                    sendMessage($msg$, $msg[tVertex]$);
        else
            foreach $msg$ in $ml$ do
                $msg[eVertex] \leftarrow = vertexID$;
                sendMessageToAll($msg$, $neighbors$);
voteToHalt();
```

4.1 Query Evaluation

For a query graph Q, we select a trail on Q that traverses each edge at least once. A trail consists of a set of ordered exploration edges $\{\bar{q}_1, \ldots, \bar{q}_n\}$. An exploration edge $\bar{q}_i$ is defined as $(v_e, p, v_t, direction)$, where v_e and v_t are vertices in the query graph and p is the edge label. The direction is either outgoing or incoming relative to $\bar{q}_i.v_e$ in the query graph. v_e and v_t are referred to as exploration and termination vertex, respectively. The termination vertex of $\bar{q}_i$ is the exploration vertex of $\bar{q}_{i+1}$. For example, a possible trail in $\overline{Q_s}$ that starts from *?p* is ($\bar{q}_1$, $\bar{q}_2$, $\bar{q}_3$)=((?p, $teaches$, ?c, out), (?c, $takes$, ?s, in), (?s, $advisor$, ?p, out)). Obviously, there are many potential trails that can start from any of the query vertices. Query planning is discussed in Section 4.2.

A query is evaluated in $n + 1$ iterations, where n is the number of exploration edges in the trail selected by the query optimizer (see Section 4.2). At each iteration, a subquery is explored by each data vertex; matching vertices send messages carrying their values to the corresponding neighbors. In succeeding iterations, only vertices that received messages continue exploration. We do not solve each subquery independently, rather a subquery is explored starting from the results of the previous one. Messages exchanged between vertices carry the intermediate results; hence, messages at the last iteration are the final answer to the query. Our execution strategy resembles right-deep join tree planing in relational databases. Messages are hashed on the id of the destination vertex; which resembles hash-join.

In every iteration, each vertex executes ExploreEdge (Algorithm 1), whose inputs are the exploration edge $\bar{q}_i$, the messages received from the previous iteration and the iteration count. As an example, query $\overline{Q_s}$ is evaluated using the previous trail and the data graph in Figure 1 as follows:
Iteration 1 ($\bar{q}_1$): Initially, all vertices are active and each vertex executes ExploreEdge with $\bar{q}_1$ and empty message list

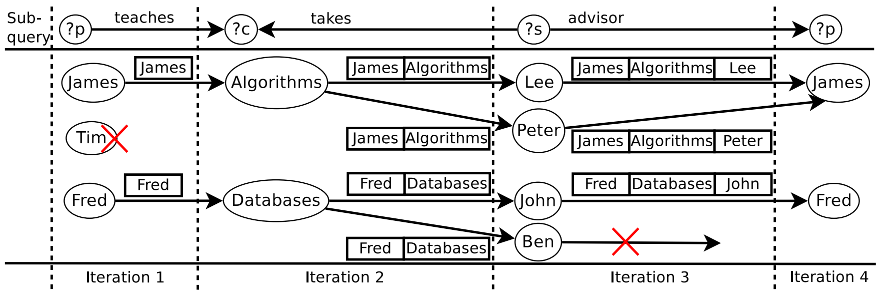


Fig. 5. Computation iterations for solving Q_s.

as inputs. Each vertex checks if it matches the exploration vertex $\bar{q}_1.v_e$. A vertex can be a match to v_e if it has all the subqueries attached to v_e in the query graph (lines 4-5). Based on the exploration edge direction, all matching vertices retrieve their neighbours connected by predicate $\bar{q}_1.p$ (lines 7-10). Each vertex creates a message containing its id and sends it to the retrieved neighbours. Finally, all vertices vote to halt; vertices become active if and only if they receive a message in the next iteration. In $\overline{Q_s}$, the exploration vertex $v_e = ?p$. A matching vertex for $?p$ needs to be a subject and an object for predicates *teaches* and *advisor*, respectively. From the data graph of Figure 1, *Fred* and *James* are matches of *?p*, while *Tim* is not, as he does not advise any students and will vote to halt. The direction of $\bar{q}_1$ is *out*, hence vertices use the PO index to get the list of neighbors (objects) connected via predicate *teaches* (line 7-8). A message is formulated from each matching vertex of *?p* and is sent to its neighbors connected via predicate *teaches* (lines 11-14). Figure 5 depicts the steps.

Iteration 2 ($\bar{q}_2$): Vertices *Databases* and *Algorithms* received messages from *Fred* and *James*, respectively; hence, they are the only active vertices. Each vertex checks if it matches the exploration vertex $\bar{q}_2.e_v = ?c$; both do. Each of them uses its PS indices to retrieve its neighbors connected via predicate *takes*. Each vertex appends its id to the received message and send the updated message to its list of neighbors (lines 21-24). Specifically, *Algorithms* sends message [James, Algorithms] to its neighbors *Lee* and *Peter*, whereas *Databases* sends [Fred, Databases] to *John* and *Ben*.

Iteration 3 ($\bar{q}_3$): Vertices *Lee*, *Peter*, *John* and *Ben* check if they match $\bar{q}_3.e_v = ?s$. Matching vertices use their PO indices to get their list of neighbors connected via predicate *advisor*. Since the termination vertex $\bar{q}_3.e_v = ?p$ has been visited before, messages are forwarded if and only if the $?p$ value in the message is also in the neighbors list. Notice that the message received by *Ben* has *Fred* as the $?p$ value, which is not in Ben's neighbors list. Therefore, the message is truncated because it is not a valid result (lines 16-20).

Iteration 4: All vertices that received messages in this iteration have the final answer of $\overline{Q_s}$. This iteration can be omitted because the terminal vertex of the last iteration has already been visited; hence, the results can be returned at the end of iteration 3. However, we keep iteration 4 for the sake of explanation.

Discussion: Our exploration approach utilizes the underlying vertex-centric framework for query evaluation. First, implicit join evaluation is achieved by inter-vertex message passing. This approach is different from the exploration approach discussed in Trinity.RDF [12], which resembles semi-join. Trinity.RDF requires a final centralized join, especially for cyclic queries [11]. In contrast, Spartex messages carry the intermediate results. Hence, no final join is needed as the final results are built and validated incrementally. Moreover, the bindings can reduce the size of the intermediate results significantly when queries have cycles: in the third iteration of our example, Ben discards its message because it can validate that Fred (a visited node) is not in his neighbors list; we call this optimization *pre-join*. Although carrying the historical bindings seems to incur high communication overhead, the maximum number of variables per query is usually small [24]. The second advantage is the search space pruning that happens because of vertex activation/halting. In an exploration iteration, only active vertices apply the compute function; inactive vertices do nothing. Hence, this activation mechanism prunes the search space by eliminating vertices that would not contribute to the query results.

4.2 Query Planning

Query evaluation performance is highly influenced by the trail followed during execution. In this section, we describe our cost-based optimizer, which for a given query, generates all possible query execution plans, estimates their costs and selects the plan with the minimum cost.

4.2.1 *Query Optimization*

The space of possible trails depends on the query graph structure and the fact that each edge has to be visited once. A trail can be defined if and only if exactly zero or two vertices have odd degree. In the former case, the graph is called *Eulerian*, while in the latter is called *traversable*. The difference is that trails in Eulerian graphs start and end at the same vertex. For example, $\overline{Q_s}$ is Eulerian and has two trails (cycles) that start and end at vertex $?p$: $?p - ?c - ?s - ?p$ and $?p - ?s - ?c - ?p$. The same applies to vertices $?c$ or $?s$. On the other hand, in a traversable graph, trails have to start from one of the odd degree vertices and end at the other odd degree vertex.

However, trails cannot be found for arbitrary queries that are neither Eulerian nor traversable. To solve this problem, the condition of visiting each edge once is relaxed by allowing the exploration of some edges more than once. This resembles the classical Chinese Postman Problem (*CPP*). Given a query graph, CPP finds a *minimum length* closed

Algorithm 2: Query Optimizer

```
Input: Query graph $Q = (V, E)$
Result: Exploration trail with minimum estimated cost
1  $maxLength \leftarrow 0$; $minCost \leftarrow$ Infinity; $bestPlan \leftarrow$ NULL;
2  if isEulerian ($Q$) then $maxLength \leftarrow Q.numEdges$ ;
3  else $maxLength \leftarrow Q.numEdges +$ CPP ($Q$) ;
4  $cost \leftarrow 0$; $listVisitedEdges \leftarrow$ Empty;
5  foreach vertex $v$ in $Q.vertices$ do
6      FindTrail ($v$, $cost$, $visitedEdges$);
7  return $bestPlan$;
8  Procedure FindTrail (Vertex $v$, $cost$, $visitedEdges$)
10     if $cost > minCost$ then return ;
11     if $visitedEdges.length > maxLength$ then return ;
12     if allVisited ($Q$, $visitedEdges$) then
13         if $cost < minCost$ then
14             $minCost \leftarrow cost$;
15             $bestPlan \leftarrow visitedEdges$;
16             return;
17     foreach Edge $e$ in $v.edges$ do
18         $newCost \leftarrow cost +$ getCost ($visitedEdges$, $e$);
19         $newVisitedEdges \leftarrow visitedEdges.add(e)$;
20         FindTrail ($e.trmVrtx$, $newCost$, $newVisitedEdges$);
21     return;
```

Algorithm 3: Exploration Trail Cost

```
Input: Exploration Trail $T$ $\{\bar{q}_1, \bar{q}_2, ..., \bar{q}_n\}$
Result: Estimated Trail cost $T_{cost}$
1  $T_{cost} \leftarrow 0$; $iterNo \leftarrow 0$;
2  foreach ExplorationEdge $\bar{q}_i \rightarrow T.edges$ do
3      $\bar{q}_i$.cost $\leftarrow$ ExplorationEdgeCost ($\bar{q}_i$, $iterNo$);
4      $T_{cost} \leftarrow T_{cost} + \bar{q}_i$.cost;
5      $iterNo$++;
6  Procedure ExplorationEdgeCost ($\bar{q}_i$, $iterNo$)
8      $iterCost \leftarrow 0$;
9      if $iterationNo = 1$ then
10         $iterCost \leftarrow$ estimateMatches ($\bar{q}_i$.ev) * $PC[\bar{q}$.pred ];
11         $coarsenedCost \leftarrow 1$;
12         foreach $\bar{q}_j \rightarrow \bar{q}_i$.ev.coarsenedEdges do
13             $coarsenedCost$ *= estimateMatches ($\bar{q}_j$.tv)
14         $iterCost \leftarrow iterCost * coarsenedCost$;
15         return $iterCost$;
16     if explored ($\bar{q}$.ev) is True then
17         if explored ($\bar{q}$.tv) is True then return $\bar{q}_{i-1}$.cost;
18         else return $\bar{q}_{i-1}$.cost * avgDegree ($q_{i-1}$.pred, $q_i$.pred) ;
19     else
20         $coarsenedCost \leftarrow 1$;
21         foreach $\bar{e_j} \rightarrow \bar{e_i}$.ev.coarsenedEdges do
22             $coarsenedCost$ *= estimateMatches ($\bar{e_j}$.tv)
23         if explored ($\bar{e}$.tv) is True then
24             return $\bar{e}_{i-1}$.cost * $coarsenedCost$;
25         else return $\bar{e}_{i-1}$.cost * avgDegree ($q_{i-1}$.pred, $q_i$.pred) *
             $coarsenedCost$ ;
27     return;
```

walk that traverses each edge at least once. For a non-Eulerian graph, CPP duplicates some edges to make it Eulerian; allowing for a larger space of possible trails.

Query Coarsening: The number of edges that the CPP will duplicate is positively correlated to the number of odd degree vertices. Therefore, we introduce a query coarsening optimization that minimizes the number of odd degree vertices. Recall that each vertex has direct access to its properties and incoming/outgoing neighbors. Therefore, all leaf vertices that have a single neighbor can be safely merged (coarsened) with their neighbor. For example, query $\overline{Q_s}$ is the coarsened version of Q_s; vertices that match $?c$ and $?p$ will validate if they have the rank and centrality predicates, respectively. This applies to all types of star queries, which are coarsened into a single vertex. Hence, star queries are solved in a single iteration without communication.

After coarsening, Algorithm 2 enumerates all possible trails and select the trail with the minimum estimated cost. We use the CPP to calculate how many edges need to be duplicated to make the graph Eulerian. CPP will return zero if the graph is already Eulerian. The maximum trail length is set to the number of edges in the graph plus the number of duplicate edges (line 3). Then, the algorithm searches for an exploration trail from each vertex in the query graph (lines 5-6) using procedure *FindTrail* (lines 8-21). We employ a branch and bound strategy to prune the search space of possible trails using the plan cost as upper-bound (plan cost estimation is discussed in Section 4.2.2). Initially, the plan cost is set to infinity. A branch is pruned in three cases: (*i*) if a valid exploration plan with less cost (so far) is found; i.e., all edges are visited, the cost bound and the best found plan are updated (lines 12-16). (*ii*) Since the cost is monotonically increasing, if the current exploration plan cost exceeded the bounded cost, the algorithm terminates (line 10). (*iii*) The algorithm terminates when the length of the exploration plan exceeds the maximum bounded length (line 11).

4.2.2 Cost Estimation

The number of exchanged messages during query evaluation depends on the order of exploration. Our optimizer tries to minimize the size of the intermediate results by exploring the most selective edges first. With the absence of a schema, selectivity estimation in SPARQL is a challenging task [5]. We propose a selectivity estimation method that captures the correlation between pairwise predicates. During data loading, each vertex collects the correlation information and sends it to the master, which aggregates the following statistics:

Predicate Counts $PC(p_i)$: for a predicate p_i, PC returns a pair (sc, oc), where sc and oc are the number of unique subjects and objects, respectively, attached to p_i in the data graph. For example, predicate *teaches* in Figure 3 appears three times and has $(sc, oc) = (3, 2)$. Similarly, *type* has 11 unique subjects and 5 unique objects.

Predicate Pairwise Degrees $PPD(p_i, p_j, d_i, d_j)$: given a pair of predicates (p_i, p_j) with their directions (d_i, d_j), $PPD(p_i, p_j, d_i, d_j)$ returns two values: (*i*) $count$ is the number of vertices that have both predicates with their respective directions. (*ii*) (ad_i, ad_j) is an estimate of the average number of predicates p_i and p_j for each vertex v. For example, $PPD(advisor, takes, out, out)$ returns {4, (1, 1)} because there are 4 vertices that have outgoing edges labeled $advisor$ and $takes$. On average, each vertex has one edge labeled $advisor$ and one edge labeled $takes$. From the exploration point of view, it means that there are 4 vertices that exist when transitioning between advisor and takes. These vertices will get one message from the previous iteration and send one message out.

Algorithm 3 calculates the cost of the exploration trail ($\bar{q}_1$, $\bar{q}_2$,..., $\bar{q}_n$) using n iterations. The plan cost is initially zero (line 1). For each exploration edge, Algorithm 3 increments

TABLE 1
Dataset statistics in millions (M). Number of triples, unique subjects, objects and predicates.

Dataset	Triples (M)	#S (M)	#O (M)	#S∩O (M)	#P
KEGG	89.18	8.63	35.68	8.50	140
LinkedGeoData	274.67	51.92	121.10	41.47	18,272
YAGO2	295.85	10.12	52.34	1.77	98
LUBM-10240	1,366.71	222.21	165.29	51.00	18
Bio2RDF	4,287.59	552.08	1,075.58	491.73	1,714

the total plan cost with the expected number of messages to be transferred during exploration (line 2-5). The subquery cost can be one of the following: (*i*) in the first iteration, the number of messages sent can be estimated from the number of matches of the current exploration vertex. The number of matches is estimated by considering all pairwise combinations of the predicates attached to the vertex. For each pair, we use PPD to get the unique number of nodes with this pair. The estimated number of matches is the minimum count of vertices in the graph that are attached to the pairwise predicates. Each matching vertex sends a number of messages equal to its average degree on predicate $\bar{q}_i.p$ (line 10). If ev has a set of coarsened subqueries attached to it, the number of messages is multiplied by the number of bindings to the coarsened leaf vertex (lines 11-14). (*ii*) If both ev and tv are already explored, then the same number of received messages is sent in this iteration (line 17). (*iii*) If ev is already explored and the termination vertex is not explored yet, then we simply forward the messages received through the exploration predicate $\bar{q}_i.p$. This serves as an upper bound on the number of messages to be sent (line 18). (*iv*) When exploration and termination vertices were not explored before, the number of messages sent is based on messages received, average degree of the exploration predicate and the number of bindings of the coarsened leaves (line 25). (*v*) If the termination vertex was visited before, the messages received are forwarded to the termination vertex after considering the coarsened leaves (lines 23-24).

5 Experimental Evaluation

The current version of Spartex is implemented on top of GPS [21]; an open-source Pregel clone. We deploy Spartex and other systems on a cluster of 12 machines each with 148GB RAM and two 2.1GHz AMD Opteron 6172 CPUs with 12 cores each. The machines run 64-bit 3.2.0-38 Linux Kernel and are connected by a 10Gbps Ethernet switch.

We use real and synthetic datasets containing from 89M to 4.2B triples; Table 1 shows the details. KEGG[6] is a real dataset that integrates biological, chemical and genomic information. LinkedGeoData[7] is a spatial knowledge base derived from OpenStreetMap. YAGO2[8] is a real dataset that combines facts from Wikipedia, WordNet and GeoNames. Bio2RDF[9] is another real dataset that interconnects 24 different biological datasets. We also use the synthetic LUBM[10] data generator to generate various synthetic datasets with up to 1.36B triples (corresponds to simulated data from 10,240 universities). We use the same YAGO2 and Bio2RDF queries used by AdPart [9], which span a wide range of structures and complexity classes. For LUBM, we use the queries defined by Atre et al. [25] and used by most RDF engines [8], [9], [10], [11], [12]; additionally, we define two more complex queries (see Appendix B).

6. http://www.genome.jp/kegg/
7. http://linkedgeodata.org/
8. http://yago-knowledge.org/
9. http://bio2rdf.org/
10. http://swat.cse.lehigh.edu/projects/lubm/

5.1 SPARQL Query Performance

In this section we evaluate the SPARQL query performance of Spartex against state-of-the-art RDF engines that are built on top of generic frameworks[11], namely: S2RDF [26], CliqueSquare [27], H2RDF+ [11] and SHARD [28]. We also tried to compare against HadoopRDF [29] but it crashed in several datasets and queries.

LUBM Dataset: LUBM queries can be classified into *simple* (L4, L5 and L6), which are very selective and touch the same small number of triples regardless of the dataset size; and *complex* (L1, L2, L3, L7, P and D), which consist of non-selective joins and generate a lot of intermediate results. Table 2 shows the result for the LUBM-10240 dataset. If a system fails to execute a query within 1 hour or crashes, we mark it N/A. H2RDF+ performs better than SHARD in all queries due to the utilization of HBase indices and its distributed implementation of sort-merge joins. The flat plans of CliqueSquare reduce the overhead of joins for complex queries. Therefore, it outperforms both SHARD and H2RDF+ in the non-selective complex queries: L1, P and D. However, for selective simple queries, H2RDF+ is faster because these queries are solved in a centralized manner; hence avoiding the overhead of MapReduce based joins. Compared to MapReduce systems, S2RDF shows a significant performance improvement due to its in-memory caching technique as well as the materialized join reduction tables. However, S2RDF requires loading multiple partitions into memory for processing each single query. Spartex utilizes the efficient inter-vertex communication of vertex-centric frameworks. For L1, L3 and L7, Spartex performs multiple joins concurrently because of the coarsening strategy. Then, only two distributed joins (2 iterations) are required for evaluating the final results. In contrast, S2RDF, H2RDF+, CliqueSquare and SHARD require multiple distributed joins. As a result, the geometric mean of Spartex is up to *two orders of magnitude* better than the competitors.

YAGO2 Dataset: Table 3 shows the results for the YAGO2 dataset. Y1 and Y2 are selective queries that result in small number of results. Y3 and Y4 are more complex and consist of non-selective object-object joins. H2RDF+ again outperforms CliqueSquare and SHARD. The flat plans do not improve the performance CliqueSquare compared to H2RDF+. The reason is that, while the flat plans reduce the number of MapReduce-based joins, H2RDF+ implements a more efficient join operator using traditional RDF indices. Furthermore, unlike CliqueSquare, H2RDF+ encodes the URIs and literals of RDF data; hence it incurs minimal overhead when shuffling intermediate results. On the other hand, S2RDF performs significantly better compared

11. In Section 5.5, we also compare the performance of Spartex against state-of-the-art native distributed RDF engines.

TABLE 2
Query runtime for LUBM-10240 (seconds)

LUBM-10240	L1	L2	L3	L4	L5	L6	L7	P	D	GMean
Spartex	**4.48**	**6.78**	**5.45**	**3.40**	**3.24**	**2.38**	**7.02**	**7.89**	**6.21**	**4.85**
S2RDF	46.55	35.80	21.53	9.22	2.72	4.55	47.34	48.10	55.89	20.04
CliqueSquare	125.02	71.01	80.01	90.01	24.00	37.01	224.04	161.02	160.02	88.35
H2RDF+	285.43	71.72	264.78	24.12	4.76	22.91	180.32	1142.10	568.58	105.860
SHARD	413.72	187.31	N/A	358.20	116.62	209.80	469.34	596.08	544.94	317.606

TABLE 3
Query runtimes for YAGO2 (sec)

YAGO2	Y1	Y2	Y3	Y4	GMean
Spartex	**2.8**	3.5	**2.0**	**2.7**	**2.7**
S2RDF	2,822	**3,032**	3,393	3,628	3,203
CliqueSquare	139.0	73.0	36.0	100.0	77.7
H2RDF+	10.9	12.3	43.9	35.5	21.4
SHARD	238.9	238.9	N/A	N/A	238.9

TABLE 4
Query runtimes for Bio2RDF (sec)

Bio2RDF	B1	B2	B3	B4	B5	GMean
Spartex	**2.0**	**3.2**	**3.4**	**7.9**	**2.3**	**3.3**
H2RDF+	5.6	12.7	322.3	7.9	4.3	15.0
SHARD	239.3	309.4	512.8	787.1	112.3	320.0

to CliqueSquare, H2RDF+ and SHARD, because it incurs heavy preprocessing of the RDF data to precompute and materialize join results. Spartex outperforms other systems by up to two orders of magnitude. The utilization of direct inter-vertex communication for join evaluation allows Spartex to evaluate queries in a more efficient way. Furthermore, the coarsening strategy helps in solving the queries in fewer iterations by evaluating multiple joins concurrently.

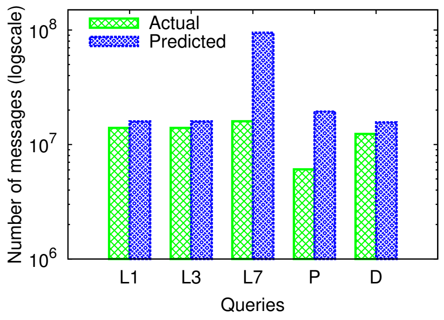
Bio2RDF dataset: Table 4 shows the results for Bio2RDF. S2RDF and CliqueSquare are excluded as they failed to process this dataset. Similar to its behavior in the other datasets, SHARD still performs worse than all other systems due to the MapReduce overhead and lack of efficient indices. Spartex outperforms both H2RDF+ and SHARD in all queries by up to an order of magnitude.

5.2 Optimizer

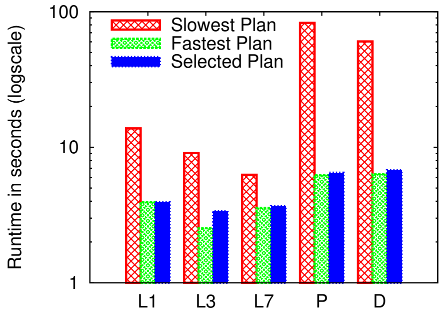
To evaluate the effectiveness of our query optimizer and cost model, we need to show that the selected plan is efficient. We use LUBM and consider only the complex queries; L1, L3, L7, P and D. These queries are solved in several iterations and generate large intermediate and/or final results[12]. In this experiment, we *execute all* possible plans for each query. Queries L1, L3 and L7 have the same structure; each has 6 possible trails. Queries P and D have 36 and 176 possible trails, respectively.

Figure 6(a) shows the estimated versus the actual number of messages transferred between vertices during the execution of the selected plan for each query. Our approach estimates accurately the total number of messages for most queries. Query L7 generates a huge number of intermediate results at the first few iterations; however, many intermediate results are dropped at the final iteration because of the

12. L6 is very selective and L2, L4, L5 are solved within a single iteration without communication



(a) Estimated vs. actual number of messages generated during query execution.



(b) Query execution plan search space.

Fig. 6. Query Optimization; LUBM-4000

cycle. Our cost function is monotonically increasing, hence, it can not capture this sudden drop of intermediate results. Nonetheless, the number of messages for the last iteration is the same for all plans; therefore, our optimizer selects an efficient plan.

Figure 6(b) shows the fastest and slowest execution times for each query and compares them with the execution time of the plan selected by our optimizer. For all complex queries our optimizer selects a plan that is either optimal, or performs close to the optimal. Note that for P, there were 19 plans that never finish because of the huge number of generated messages, which cause network contention.

5.3 Scalability

To evaluate the scalability of Spartex, we conduct two experiments: (*i*) varying the size of the data while fixing the number of workers (cores) and (*ii*) varying the number of workers while the data size is fixed.

Data Scalability: Using the LUBM data generator, we generate five datasets: LUBM-500, LUBM-1000, LUBM-2000, LUBM-4000 and LUBM-8000. Since simple queries touch almost the same amount of data regardless of the data size,

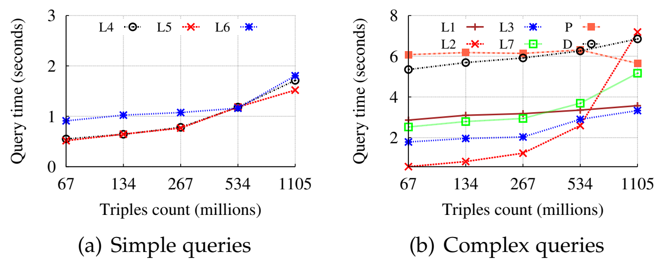


Fig. 7. Data Scalability using LUBM dataset.

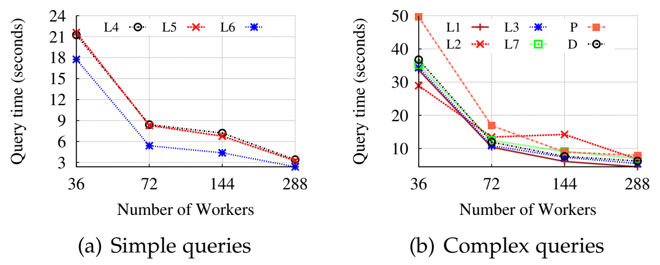


Fig. 8. Machine Scalability (LUBM-10240).

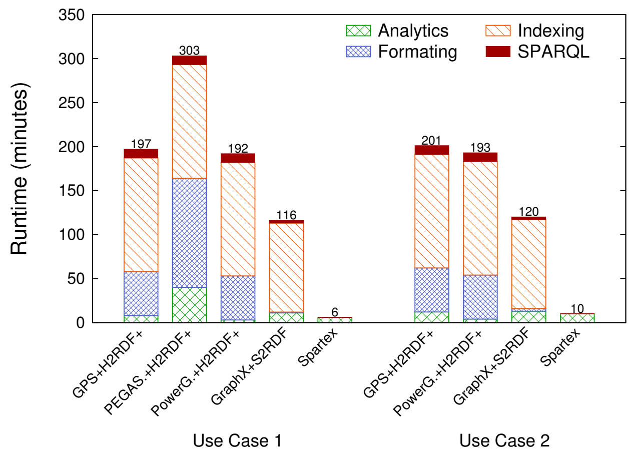


Fig. 9. Rich RDF Analytics: In the first use case, graph analytics output is utilized in a SPARQL query while in the second case, the output of a SPARQL query is used as input to a graph analytics algorithm.

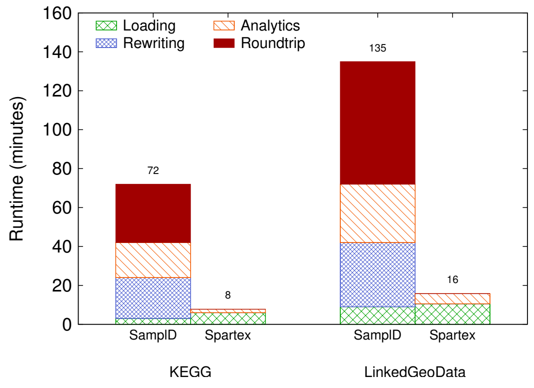


Fig. 10. Use case 3: SampID analytics pipeline.

Spartex provides almost a steady performance for these queries as shown in Figure 7(a). In Figure 7(b), we show that Spartex scales well for the complex LUBM queries. Notice that L2 is a reporting query that generates a proportional amount of results to the data size. Therefore, the response time of this query increases as the data grows.

Machine Scalability: In this experiment, we vary the number of workers while the dataset is fixed to LUBM-10240. Figure 8 shows the scalability results for Spartex as the number of workers increases. Spartex achieves almost ideal scalability up to 144 workers for both simple and complex queries. After that, query response times are dominated by the communication cost which grows as we increase the number of workers.

5.4 Rich RDF Analytics

To demonstrate the effectiveness of Spartex on rich RDF analytics, we evaluate the three case studies described in Section 2.2. Since no other system can fully support these case studies, we compare against combinations of SPARQL engines and graph processing systems. Specifically, we combine S2RDF [26] and H2RDF+ [11] SPARQL engines with four different graph analytics systems; GPS [21], GraphX [30], PowerGraph [17] and PEGASUS [31].

Figure 9 shows the wall time of the first two use cases (see Section 2.2) for the LUBM-4000 dataset. Some graph engines take more time than others to load the graph. For example, PowerGraph and GPS require around 10min for graph loading and indexing compared to only 2min for GraphX. For fair comparison, we exclude the graph loading time for all systems. All combinations in this experiment require the data to be moved between multiple systems and be formatted accordingly. In the first use case, graph analytics are executed prior to query evaluation. PowerGraph, GPS and PEGASUS are used to evaluate PageRank and degree centrality algorithms and the output is stored in HDFS. The output of GraphX is materialized as in-memory RDDs which are then merged with the original graph and given as input to S2RDF. PEGASUS performed worse than all other graph engines confirming that MapReduce approaches do not perform well for graph analytics. For H2RDF+ based approaches, the results are converted to RDF format and passed to H2RDF+ along with the original RDF graph. Then, H2RDF+ partitions the input data, builds its RDF indices, and evaluates the SPARQL query. Similarly, S2RDF+ preprocesses the RDF data (i.e., original graph and computed PageRank and centrality values) to build its indices. S2RDF requires heavy preprocessing to pre-compute join reductions for each two vertical partitions in the data. The results are then materialized as tables in HDFS. While both GraphX and S2RDF are based on Spark, S2RDF does not run directly on Spark; instead, it translates SPARQL queries into SQL jobs which are then executed on top of Spark SQL. In general, the cost of data formatting and indexing in all combinations is substantial, accounting for more than 80% of the runtime.

Spartex performs better than all above-mentioned systems, because all processing is done within the same framework, without additional data formatting or indexing. Note that, for pure SPARQL queries, Spartex performs significantly better than H2RDF+ and S2RDF. The

TABLE 5
Query engine. LUBM-10240; runtime (sec)

LUBM-10240	**L1**	**L2**	**L3**	**L4**	**L5**	**L6**	**L7**	**P**	**D**	**GMean**
Spartex-Native	2.881	0.406	2.953	**0.001**	**0.001**	**0.010**	**2.386**	**3.408**	**4.768**	0.222
AdPart-NA	**2.743**	**0.120**	**0.320**	0.001	0.001	0.040	3.203	5.724	4.793	**0.193**
TriAD	6.023	1.519	2.387	0.006	0.004	0.114	17.586	19.839	65.628	1.035
TriAD-SG (100K)	5.392	1.774	4.636	0.009	0.005	0.010	21.567	44.135	144.256	1.119
SHAPE	25.319	4.387	25.360	1.603	1.574	1.567	15.026	N/A	N/A	5.575

same applies on the second use case; however, since the SSSP algorithm is not available in PEGASUS, we only compare to GPS+H2RDF+, PowerGraph+H2RDF+ and GraphX+S2RDF. Notice that the SPARQL query evaluation part is minimal compared to the total time. Therefore, using a more efficient specialized RDF engine (e.g. AdPart or TriAD), will not help because the total runtime is dominated by the formatting and indexing phases. The same applies when using a more efficient graph analytics engine like PowerGraph.

The third use case, SamplD [2] (see Section 2.2.3), starts by loading the RDF data into HDFS and transforming it into an unlabeled directed graph, which is then processed by Apache Pig and Giraph to evaluate PageRank and degree centrality. The results are stored in RDF format with each triple assigned a score that denotes its importance. Figure 10 shows the time of each phase of the SamplD pipeline for two real datasets; KEGG and LinkedGeoData. The rewriting (RDF to unlabeled graph) and the reverse (unlabeled graph to RDF) phases consume most of the time. Spartex does not incur such overhead. As a result, Spartex provides a single system for the entire SamplD pipeline with almost one order of magnitude better performance.

5.5 Native Spartex Query Engine

In this section, we compare Spartex against state-of-the-art native distributed RDF systems. For fair comparison, we implemented our SPARQL query engine only (i.e., not the entire Spartex) outside any vertex-centric framework, using C++ and MPI. We call this version *Spartex-Native*. We compare Spartex-Native against the following systems: (*i*) AdPart [9] is the current state-of-the-art distributed in-memory RDF system. It uses workload adaptive graph partitioning and implements locality-aware query optimizations. Since our work is orthogonal to graph partitioning, we compare against the non-adaptive version, called AdPart-NA. (*ii*) TriAD[13] [8] is a distributed in-memory RDF engine that uses sophisticated graph partitioning and asynchronous message passing. We also compare against TriAD-SG, a version of TriAD that uses graph summaries for join-ahead pruning. (*iii*) SHAPE [10] is a distributed engine that uses RDF-3X for storage and relies on static replication. We configure SHAPE with all possible optimizations and make sure that all queries can be solved without communication between nodes.

Table 5 shows the runtime in seconds for the LUBM-10240 dataset. For the simple queries, all systems except SHAPE, perform similarly. SHAPE is much slower because replicas are stored together with the original triples, resulting in significant overhead. For complex queries, Spartex-Native is significantly faster than TriAD, TriAD-SG and SHAPE because of the better execution plans and the various optimizations. For example, queries L1 and L7 are cyclic, allowing Spartex-Native, TriAD and TriAD-SG to execute multiple joins concurrently; however, Spartex-Native minimizes communication. Compared to AdPart-NA, Spartex-Native is slightly better for some complex queries and slightly worse for others. For example, query L3 returns empty results; AdPart-NA evaluates the join that generates the empty set earlier than Spartex-Native.

13. Trinity.RDF [12] is not publicly available; therefore we compare to TriAD, which reported [8] performance superior to Trinity.RDF.

TABLE 6
Query engine. YAGO2; runtime (msec)

YAGO2	**Y1**	**Y2**	**Y3**	**Y4**	**GMean**
Spartex-Native	38	126	**35**	33	**49**
AdPart-NA	19	**46**	570	77	79
TriAD	**16**	1,568	220	**18**	100
SHAPE	1,824	665,514	1,823	1,871	8,022

TABLE 7
Query engine. Bio2RDF; runtime (msec)

Bio2RDF	**B1**	**B2**	**B3**	**B4**	**B5**	**GMean**
Spartex-Native	**1**	**1**	8	**26**	**2**	**3**
AdPart-NA	17	16	32	89	1	15
TriAD	4	4	**5**	N/A	2	4

Table 6 shows the runtime in msec for YAGO2. Queries Y1 and Y2 are simple, whereas Y3 and Y4 are complex. Compared to AdPart-NA and TriAD, Spartex-Native is slightly slower for some queries, but significantly faster for others. In terms of geometric mean (GMean) over all queries, Spartex-Native is clearly better. The bad performance of SHAPE is explained by the fact that the 2-hop forward partitioning placed all data in a single partition; therefore, although 12 machines are available, one is overloaded. TriAD-SG is not listed because we do not know the optimal number of summary graph partitions, a process that requires empirical evaluation [8]. Similar results are shown in Table 7 for Bio2RDF. Again, Spartex-Native is the fastest for the majority of the queries, and achieves the lowest geometric mean. TriAD-SG is excluded for the same reason as above. SHAPE is also excluded because, with 2-hop forward partitioning, it failed to preprocess the Bio2RDF dataset within reasonable time.

In summary, our core SPARQL query engine is at least as fast as the state-of-the-art distributed RDF systems. This shows the effectiveness of our query optimizer and the efficiency of our SPARQL query engine. It also demonstrates that our SPARQL operator can be used as an efficient standalone distributed RDF engine.

6 Related Work

Specialized SPARQL engines. Several distributed RDF systems [11], [27], [28], [32] are built on top of MapReduce [33]. While the underlying framework is capable of performing graph analytics, these systems are optimized for solving SPARQL only. Even for SPARQL query evaluation and due to the expensive overhead of MapReduce-based joins, these systems perform poorly compared to the specialized ones [8], [9], [12]. Furthermore, MapReduce is not suitable for iterative graph algorithms as it requires passing the entire graph state from one iteration to the next [16]. All specialized RDF engines [5], [6], [7], [8], [9], [12], [25] are designed and optimized for SPARQL query evaluation. Unlike Spartex, they are incapable of performing generic graph processing over RDF data.

SPARQL on graph frameworks. Many frameworks have been proposed for efficient graph analytics, including Pregel [16], GRACE [18], PowerGraph [17] and SociaLite [34]. However, these systems lack the capability of evaluating ad-hoc SPARQL queries, which means that a program has to be written for each SPARQL query [35]. Sedge [35] and Goodman et al. [36] proposed implementations for solving SPARQL queries using vertex-centric frameworks. However, both approaches do not have a query optimizer and therefore users have to select a good query evaluation plan manually. Such an approach is tedious, counter-productive and requires prior knowledge about the data. Evaluating a query using a non-optimal plan can take significant time or cause the system to run out of memory (see Section 5.2). In contrast, Spartex implements a fully-fledged SPARQL operator with an efficient cost-based optimizer that is capable of evaluating ad-hoc SPARQL queries. S2RDF [26] and Trinity.RDF [12] are distributed RDF engines built on top of Spark [37] and Trinity [23], respectively. Both Trinity.RDF and S2RDF are primarily designed and optimized for SPARQL query evaluation only. They do not have means for pipelining SPARQL with general graph operations. Therefore, a program for each pipeline has to be written to take the input of one operator and feed it to the next.

Cardinality estimation. Stocker et al. [38] proposed a selectivity estimation method of BGPs. It gathers statistics about each subject, predicate and object and assumes their distributions are independent. Therefore, it does not capture the correlations that exist among different triples, which leads to severe underestimation [39]. RDF-3X [5] uses specialized histograms and also precomputes the frequent paths in the data while keeping their exact join statistics. This technique suffers from combinatorial explosion and captures only a few cases for predicate correlation [39]. Trinity.RDF [12] maintains a two-dimensional $predicate \times predicate$ matrix, where each entry has all four predicate direction combinations. Since not all predicates are actually correlated, this matrix becomes extremely sparse. Neumann et al. [39] provide better cardinality estimation by counting the characteristic sets of the data. Each set stores a group of correlated predicates along with their statistics. In Spartex, the query execution plan follows a trail over predicates; therefore, our pairwise predicate correlations provide accurate estimates when moving between triple patterns. For star-shaped queries, characteristic sets would provide better estimation as they capture correlation among multiple predicates. However, this has more computation overhead due to the limited granularity (vertex-based) of vertex-centric frameworks which require more communication/synchronization in order to collect the characteristic sets.

Rich RDF analytics. Deweese et al. [40] and Qi et al. [14] used SPARQL to implement different clustering algorithms. Similarly, Techentin et al. [15] used the SPARQL 1.1 update capability to implement few iterative algorithms. These approaches are limited to certain graph algorithms. Furthermore, expressing graph algorithms in SPARQL results in lengthy and verbose queries that are hard to evaluate and understand. uRiKA [13] allows the invocation of a few predefined graph algorithms. These algorithms are tailored to uRiKA by experts; users can not add any new algorithm or modify any existing one. In contrast, Spartex allows users to write procedural code for implementing any algorithm using the simple vertex-centric API's.

Generic subgraph pattern matching. Gao et al. [41] and Fard et al. [42] proposed techniques for approximate subgraph pattern matching over vertex-centric frameworks. However, these solutions are approximate, while SPARQL requires exact subgraph pattern matching. Horton+ [43] solves reachability queries over large attributed multigraphs. It targets only path queries with closures and cannot solve generic SPARQL queries with complex structure or cycles. Moreover, unlike reachability queries, SPARQL allows variable vertices that can match any node in the graph.

Unified frameworks. Vertexica [44] is a relational database system capable of performing graph analytics. It does not focus on RDF; hence, SPARQL queries are not supported. GraphX [30] is a graph processing system built on top of Spark [37]. GraphX unifies graph-parallel (e.g., vertex-centric) and data-parallel computations (e.g., map-reduce) in a single system. Consequently, it is not as fast as specialized graph engines [30]. Yet, it allows users to stay within a single framework and remove the burden of moving data between systems and format it accordingly. Spartex is inspired by the same motivation which tries to unify both SPARQL structural querying and generic graph computations. In fact, Spartex along with its query optimizations can be implemented on top of GraphX to support rich RDF data analytics on top of Spark.

7 Conclusions and Future Work

Spartex bridges the gap between specialized RDF stores and generic graph engines. Our proposed SPARQL extension allows the invocation of user defined vertex-centric programs. Declarative SPARQL queries as well as procedural graph algorithms can be pipelined without moving and reformatting data between different systems. By coupling our SPARQL operator with a cost-based optimizer, complex SPARQL queries are evaluated efficiently. Spartex significantly outperforms approaches that employ combinations of existing systems when performing rich RDF analytics. For pure SPARQL queries, Spartex is highly scalable and as fast as the best specialized existing RDF engines. For future work, we will consider the oportunity for cross-algorithms optimizations that may be possible when combining SPARQL and graph analytics in the same query. For example, most of the execution time is spent on analytical algorithms. Therefore, multiple query optimization techniques can be employed to evaluate multiple algorithms concurrently. Notice that analytical queries may consist of a pipeline of operators. Hence, pipelines of multiple queries need to be aligned in a way that minimizes the overall execution time.

Appendix A
Implementation Details

We show below the *compute* function for PageRank in GPS, and hence Spartex, for generic graphs. The inputs are the set of incoming messages to the current vertex (if any) and the current iteration (superstep) number. In the first iteration, all vertices are assigned rank value equal to $1/N$, where N is the number of vertices (line 5-7). Then, each vertex sends its value to all its neighbours. In the following iterations, each vertex aggregates the incoming messages, calculates the new PageRank value (lines 11 -16), sends it to all neighbours (lines 18-20) and updates the current vertex state (line 22). After reaching the maximum number of iterations, vertices vote to halt.

```
1  public void compute(Iterable<DoubleWritable>
       incomingMessages, int superstepNo) {
2      int numVertices = ((IntSumGlobalObject)
           getGlobalObjectsMap().getGlobalObject(
3        GlobalObjectsMap.NUM_TOTAL_VERTICES))
4        .getValue().getValue();
5      if (superstepNo == 1) {
6        setValue(new DoubleWritable((double) 1 /
             (double) numVertices));
7        sendMessages(getNeighborIds(), getValue());
8        return;
9      }
10
11     double sum = 0.0;
12     for (DoubleWritable messageValue :
           incomingMessages) {
13       sum += messageValue.getValue();
14     }
15
16     double currentState = 0.85 *
           sum/getNeighborIds().length + 0.15 / (double)
           numVertices;
17
18     int[] neighborIds = getNeighborIds();
19     DoubleWritable messageValue = new
           DoubleWritable(currentState);
20     sendMessages(neighborIds, messageValue);
21
22     setValue(new DoubleWritable(currentState));
23     if (superstepNo == numMaxIterations) {
24       voteToHalt();
25     }
26   }
```

The same *compute* function can be executed in Spartex on the RDF data without any changes. To allow for this flexibility, we modified the underlying data-access module of GPS. The main changes are: (*i*) prior to calling the compute function on a vertex, Spartex checks whether the vertex should be skipped, by evaluating the given user filters. (*ii*) To materialize the results, Spartex stores the newly computed vertex value in its miniature properties store right after evaluating the compute function. (*iii*) For label-based and label-oblivious neighbour access, we changed the data access procedures accordingly. For example, the function $getNeighborIds()$ in lines 7 and 18 is modified to return the set of incoming and outgoing neighbours regardless of the predicate. We also enriched GPS data-access module to allow for retrieving only incoming (or outgoing) neighbours with or without label constraints. Finally, (*iv*) before returning the set of edges/neighbours, Spartex applies any edge filters.

Appendix B
Queries

We use the standard 7 LUBM queries defined in [25] and used by most distributed RDF systems [8], [9], [11], [12]. Additionally, we define two more complex queries to test the systems rigorously. The two queries are defined below. For YAGO and Bio2RDF, we use the same queries used by AdPart [9].

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://swat.cse.lehigh.edu/onto/univ-bench.owl#>
P: SELECT ?y ?z WHERE {      ?t ub:worksFor ?z .
?z ub:subOrganizationOf ?y . ?z rdf:type ub:Department .
?x ub:advisor ?t .           ?x rdf:type ub:GraduateStudent .
?x ub:memberOf ?z .          ?x ub:undergraduateDegreeFrom ?y .
?t ub:mastersDegreeFrom ?y . ?y rdf:type ub:University .
}

D: SELECT ?y ?z WHERE {
?z ub:subOrganizationOf ?y .  ?z rdf:type ub:Department .
?x ub:memberOf ?z .           ?x rdf:type ub:GraduateStudent .
?y rdf:type ub:University .   ?x ub:undergraduateDegreeFrom ?y .
?x ub:advisor ?t .            ?t ub:worksFor ?z . }
```

References

[1] 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*, vol. 10, p. S4, 2009.

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

[3] C. Tao, P. Wu, and Y. Zhang, "Linked vaccine adverse event data representation from vaers for biomedical informatics research," in *PAKDD*, 2014, pp. 652–661.

[4] Y. Zhang, C. Tao, Y. He, P. Kanjamala, and H. Liu, "Network-based analysis of vaccine-related associations reveals consistent knowledge with the vaccine ontology." *J. Biomedical Semantics*, vol. 4, p. 33, 2013.

[5] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *VLDB J.*, vol. 19, no. 1, pp. 91–113, 2010.

[6] 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*, vol. 6, no. 7, pp. 517–528, 2013.

[7] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao, "gStore: A Graph-based SPARQL Query Engine," *VLDB J.*, vol. 23, no. 4, pp. 565–590, 2014.

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

[9] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, "Accelerating sparql queries by exploiting hash-based locality and adaptive partitioning," *VLDB J.*, pp. 1–26, 2016.

[10] K. Lee and L. Liu, "Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning," *PVLDB*, vol. 6, no. 14, 2013.

[11] 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.

[12] K. Zeng, J. Yang, H. Wang, B. Shao, and Z. Wang, "A distributed graph engine for web scale RDF data," *PVLDB*, vol. 6, no. 4, 2013.

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

[14] L. Qi, H. Lin, and V. Honavar, "Clustering remote RDF data using SPARQL update queries," in *ICDEW*, 2013.

[15] 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, pp. 216–223.

[16] 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.

[17] Gonzalez, Joseph E. and Low, Yucheng and Gu, Haijie and Bickson, Danny and Guestrin, Carlos, "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *OSDI*, 2012.

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

[19] "Apache Giraph," https://giraph.apache.org/.

[20] "Apache Pig," https://pig.apache.org/.

[21] S. Salihoglu and J. Widom, "GPS: A Graph Processing System," in *SSDBM*, 2013.

[22] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola and J. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[23] B. Shao, H. Wang, and Y. Li, "Trinity: a distributed graph engine on a memory cloud," in *SIGMOD*, 2013.

[24] M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente, "An empirical study of real-world sparql queries," in *USEWOD*, 2011.

[25] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler, "Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data," in *WWW*, 2010.

[26] A. Schätzle, M. Przyjaciel-Zablocki, S. Skilevic, and G. Lausen, "S2rdf: Rdf querying with sparql on spark," *PVLDB*, vol. 9, no. 10, 2016.

[27] F. Goasdoué, Z. Kaoudi, I. Manolescu, J.-A. Quiané-Ruiz, and S. Zampetakis, "Cliquesquare: Flat plans for massively parallel rdf queries," in *ICDE*, 2015.

[28] K. Rohloff and R. E. Schantz, "High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store," in *Programming Support Innovations for Emerging Distributed Applications*, 2010.

[29] M. Husain, J. McGlothlin, M. Masud, L. Khan, and B. Thuraisingham, "Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing," *TKDE*, vol. 23, no. 9, 2011.

[30] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: Unifying data-parallel and graph-parallel analytics," *arXiv:1402.2394*, 2014.

[31] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *ICDM*, 2009, pp. 229–238.

[32] A. Schätzle, M. Przyjaciel-Zablocki, and G. Lausen, "PigSPARQL: Mapping sparql to pig latin," in *SWIM*, 2011.

[33] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[34] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed socialite: A datalog-based language for large-scale graph analysis," *PVLDB*, vol. 6, no. 14, pp. 1906–1917, 2013.

[35] S. Yang, X. Yan, B. Zong, and A. Khan, "Towards effective partition management for large graphs," in *SIGMOD*, 2012.

[36] E. L. Goodman and D. Grunwald, "Using vertex-centric programming platforms to implement sparql queries on large graphs," in *IA3*, 2014, pp. 25–32.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, 2010.

[38] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds, "Sparql basic graph pattern optimization using selectivity estimation," in *WWW*, 2008.

[39] T. Neumann and G. Moerkotte, "Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins," in *ICDE*, 2011, pp. 984–994.

[40] K. Deweese, J. R. Gilbert, A. Lugowski, S. Reinhardt, J. Kepner, and J. R. Gilbert, "Graph Clustering in SPARQL," in *SIAM Workshop on Network Science*, 2013, pp. 930–941.

[41] J. Gao, C. Zhou, J. Zhou and J. Yu, "Continuous Pattern Detection over Billion-Edge Graph Using Distributed Framework," in *ICDE*, 2014.

[42] A. Fard, M. Nisar, L. Ramaswamy, J. A. Miller, and M. Saltz, "A distributed vertex-centric approach for pattern matching in massive graphs," in *IEEE BigData*, 2013.

[43] M. Sarwat, S. Elnikety, Y. He and M. Mokbel, "Horton+: A Distributed System for Processing Declarative Reachability Queries over Partitioned Graphs," *PVLDB*, vol. 6, no. 14, pp. 1918–1929, 2013.

[44] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker, "Vertexica: Your Relational Friend for Graph Analytics!" *PVLDB*, vol. 7, no. 13, pp. 1669–1672, 2014.

Ibrahim Abdelaziz received his BS and MSc degrees in computer science from Cairo University, Egypt. He is currently working toward the Ph.D. degree in computer science at KAUST, Saudi Arabia. Prior to joining KAUST, he used to work on pattern recognition and information retrieval in several research organizations in Egypt. His current research interest are Data Mining over large scale graphs, Distributed Systems and Machine Learning.

Razen Al-Harbi received his BS degree in information and computer science from King Fahd University of Petroleum and Minerals (KFUPM), Saudi Arabia in 2006. Then, he received his MEng. degree in 2006 from Cornell University and PhD degree from King Abdullah University of Science and Technology (KAUST) in 2016. Razen currently works as a petroleum engineering system analyst at Saudi Aramco. His primary research interest includes large scale data management with emphasis on managing and querying large-scale RDF graphs.

Semih Salihoglu is an assistant professor at University of Waterloo's Cheriton School of Computer Science. He is a member of the Data Systems Research Group. Semih received his BS in Computer Science and Economics from Yale university in 1998 and his PhD from the Stanford university in 2015. For graph processing, Semih's research has ranged from building an open-source platform for scalable graph processing, to algorithms for distributing graphs across machines, to developing extensions to existing graph processing systems with the goal of making them easier to program.

Panos Kalnis is an associate professor of Computer Science in the King Abdullah University of Science and Technology (KAUST). In 2009, he was visiting assistant professor in the Dept. of Computer Science, Stanford University. Before that, he was an assistant professor in the Dept. of Computer Science, National University of Singapore (NUS). In the past he was involved in the designing and testing of VLSI chips in the Computer Technology Institute, Greece. He also worked in several companies on database designing, e-commerce projects and web applications. He is an associate editor for the IEEE Transactions on Knowledge and Data Engineering and serves on the editorial board of The VLDB Journal. He received his Diploma in Computer Engineering from the Computer Engineering and Informatics Dept. , University of Patras, Greece in 1998 and his PhD from the Computer Science Dept., Hong Kong University of Science and Technology (HKUST) in 2002. His research interests include Databases, Cloud Computing, Distributed Systems, Large Graphs and Data Privacy.