Distributed Subgraph Matching on Big Knowledge Graphs Using Pregel

QIANG XU1, XIN WANG1,2, (Member, IEEE), JIANXIN LI3, (Member, IEEE), QINGPENG ZHANG4, (Member, IEEE), AND LELE CHAI1

1College of Intelligence and Computing, Tianjin University, Tianjin 300350, China  
2Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin 300350, China  
3School of Information Technology, Deakin University, Burwood, VIC 3125, Australia  
4School of Data Science, City University of Hong Kong, Hong Kong

Corresponding author: Xin Wang (wangx@tju.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61572353, in part by the National High-Tech Research and Development Program of China (863 Program) under Grant 2013AA013204, and in part by the Natural Science Foundation of Tianjin under Grant 17JCYBJC15400.

ABSTRACT With RDF becoming the de facto standard for representing knowledge graphs, it is indispensable to develop scalable subgraph matching algorithms over big RDF graphs stored in distributed clusters. In this paper, we propose a novel distributed subgraph matching method SP-Tree, using the Pregel model, to answer subgraph matching queries on big RDF graphs. In our method, the query graph is transformed to a variant spanning tree based on the shortest paths. Two optimization techniques are proposed to improve the efficiency of our algorithms. One employs RDF shapes to filter out local computations and messages passed, the other postpones the Cartesian product operations in the matching process to reduce intermediate results. The extensive experiments on both synthetic and real-world datasets show that our SP-Tree subgraph matching method outperforms the state-of-the-art methods by an order of magnitude.

INDEX TERMS Subgraph matching, distributed, knowledge graphs, RDF, Pregel.

I. INTRODUCTION

The Resource Description Framework (RDF) is a W3C recommendation used for representing and organizing resources in knowledge graphs. The RDF data can be represented as a labeled, directed graph, which consists of a set of triples \((s, p, o)\). An RDF triple \((s, p, o)\), where \(s\) is the subject, \(p\) the predicate, and \(o\) the object, can be viewed as a directed edge in an RDF graph. Each triple \((s, p, o)\) represents a statement that \(s\) has the relationship \(p\) with \(o\). Figure 1 shows an example of the RDF graph excerpted from the DBpedia dataset. With the proliferation of knowledge graphs, large amounts of RDF graphs have been released, which often contain hundreds of millions of triples. So how to efficiently query big RDF graphs has been widely recognized as a challenging problem in data management.

Subgraph matching is one of the most fundamental types of graph queries. Given an RDF graph \(G\) and a query graph \(Q\), subgraph matching is to find subgraphs over \(G\) that satisfy all the triple patterns in \(Q\), which is actually a conjunctive query (CQ) on \(G\). It is known that conjunctive queries are NP-complete w.r.t. combined complexity, while its data complexity is in polynomial time [1]. For instance, the following CQ \(Q_1\), consisting of two triple patterns, finds the starrings and producers of the film TheDebt, with one of the query results being highlighted in red, as shown in Figure 1.

\[
Q_1(?x, ?y) \leftarrow \text{TheDebt(starring, ?x)} \land \text{TheDebt(producer, ?y)}
\]

Currently, there has been some research works on subgraph matching in a distributed environment. One type of methods is based on query decomposition. In [2]–[4], query graphs are decomposed into a sequence of triple patterns (i.e., edges) that are executed iteratively using MapReduce, thus not leveraging structural information from query graphs, which may lead to unbalanced and potentially large intermediate results. Similarly, S2X [5] and S2RDF [6] also decompose a SPARQL query into a collection of triple patterns, in which answering the SPARQL query can be converted into a subgraph matching problem. To evaluate a SPARQL query of \(n\) triple patterns, both methods need
FIGURE 1. An example RDF graph $G$. 

In this paper, we propose a novel method which transforms a query graph to a variant of the conventional spanning tree structure, called \textit{SP-Tree}, to evaluate RDF subgraph matching. The \textit{SP-Tree} is then matched against the data graph in parallel by using Pregel [13], a vertex-centric graph computational model. This matching starts with the leaves of the \textit{SP-Tree} in a bottom-up manner; in each iteration, one level of the tree is matched. As to the optimization techniques, we employ the newly endorsed W3C RDF Shapes recommendation [14] as the \textit{priori} knowledge to filter out invalid local computations and messages passed among vertices in Pregel iterations, the other decomposing the \textit{SP-Tree} into a set of \textit{SP-Paths} to postpone part of Cartesian product operations; and (3) extensive experiments on both synthetic and real-world RDF graphs have been conducted to verify the efficiency and scalability of our method. The experimental results show that our method outperforms the state-of-the-art methods by an order of magnitude.

The rest of this paper is organized as follows. Section 2 briefly reviews related work. In Section 3, preliminary definitions on RDF subgraph matching and Pregel are introduced. In Section 4, we describe in detail the generation of \textit{SP-Tree}, the basic subgraph matching algorithm using Pregel, and its complexity analysis. We then present two optimization techniques in Section 5. Section 6 shows experimental results, and we conclude in Section 7.

II. RELATED WORK

We first introduce the related work on big graph processing to justify why we need to choose Pregel as our parallel framework.

In the last decade, the research of big graph processing has attracted considerable attention from both the industry and academia [13], [15]–[20]. It has turned out that the traditional MapReduce parallel programming model is inappropriate for implementing graph processing algorithms due to the iterative nature in their computation. The Pregel model [13], which is based on bulk synchronous parallel abstraction [21], is a parallel graph computational model that employs the vertex-centric message passing approach to naturally implement graph processing algorithms in a sequence of superstep iterations. GraphLab [16] is another major parallel graph processing model, which uses an asynchronous distributed shared-memory abstraction to realize machine Learning and data mining algorithms on graphs. However, in this paper, we choose Pregel as the graph parallel programming framework for our \textit{SP-Tree} algorithms since Pregel is more widely implemented in the mainstream big data platform, such as Spark and Giraph, which mostly rely on share-nothing architectures. Still, there exist other distributed/parallel graph processing systems, including GPS [19] which is actually an extension of Pregel with a couple of optimizations, Trinity [20] which is a dedicated main-memory distributed graph
engine developed in Microsoft, PowerGraph [17] which is a graph-parallel abstraction particularly designed for power-law graph computation, and GRAPE [22] which can parallelize sequential graph computations in terms of partial and incremental evaluation.

The existing research work on distributed subgraph matching over big knowledge graphs can be categorized as follows:

**A. QUERY DECOMPOSITION BASED METHODS**

SHARD [2] persists the RDF graph as RDF triples and answers SPARQL queries over this graph using MapReduce. Each iteration consists of a MapReduce operation for a single triple pattern in SPARQL queries, and this iteration of map-reduce-join continues until all triple patterns are processed. Similarly, in HadoopRDF [3] a single triple pattern cannot simultaneously take part in more than one join in a single Hadoop job and the order of triple pattern joins is determined by a greedy algorithm in terms of summary statistics for estimating join selectivity. The SGIA-MR algorithm [4] also deals with the problem of subgraph matching in MapReduce, in which each round adds one edge with the join operation. Despite S2X [5] builds on the Spark GraphX [15] parallel framework, it does not employ any structural information of query graphs. Instead, the query graph in S2X is decomposed into triple patterns, all of which are matched first; then a part of invalid intermediate results of these triple patterns are discarded by iterative computation, and the remaining matching results are joined in the end. S2RDF [6] introduces the relational partitioning model ExtVP to store RDF data, by which it can effectively minimize the query input size. However, the cost of the semi-join preprocessing in S2RDF is prohibitively expensive. All these above methods do not take full advantage of structural information of query graphs, thus a large number of join operations may incur expensive costs.

The multiway method [8] partitions vertices of a query graph into two sets by which the query graph is decomposed into two subgraphs to optimize a CQ. In this method, only one round of MapReduce is needed and each computation node maintains a complete copy of the data graph. Therefore, the multiway method might encounter the scalability problem when the data graph is large. In [7], a query graph is decomposed into multiple STwig structures (trees of height 1) which are matched in a specific order. The results of STwigs are aggregated into the final result using join operations. Lai et al. [9] use an A* algorithm to optimize the query decomposition, in which the basic join element is a TwinTwig structure (an edge or two incident edges). They [10] also propose a distributed subgraph enumeration method SEED decomposing the query into stars and groups. However, these methods take a great deal of effort to find a (sub)optimal query decomposition and matching order only under which the query efficiency can be improved, thus the cost of join operations among all subquery results might be highly expensive.

**B. QUERY GRAPH BASED METHODS**

PSgl [11] uses a width-first search strategy, which relies entirely on the traversal of the data graph to avoid join operations. PSgl adopts traditional optimization strategies in unlabeled and undirected graphs, including automorphism breaking, initial vertex selection, and degree information filtering. All these strategies are not easily adapted to RDF graphs with URIs as the unique vertex labels. In addition, the distributed gStore system [12] performs SPARQL queries using partial evaluation and assembly. In this method, data graph is partitioned by the METIS algorithm, each slave machine keeps a complete version of the query graph and evaluates the query in the partial computation phase, and then the local partial matches are joined together in the assembly phase to obtain the final result. It is worth noting that how to reduce the enormous intermediate results is the most crucial problem in these methods.

In our method, the spanning tree structure SP-Tree is first extracted from the query graph, which can be further decomposed into SP-Paths that can be used to reduce invalid intermediate results even further. We only need to consider minimizing the height of the SP-Tree to reduce the number of iterations in Pregel computation without taking into account the different SP-Trees and matching orders. Instead of joining all partial results, the matching results of SP-Paths, whose roots match to the same vertex $v_r$ in RDF graph, are joined together in computation of the vertex $v_r$ to substantially reduce unpromising join operations.

As to RDF shapes, ShEx and SHACL both define the notion of a shape. Prud’hommeaux et al. [23] describe the Shape Expression (ShEx) definition language to validate RDF data by declaring constraints on RDF, which can be seen as domain specific language to define Shapes of RDF graph based on regular expressions; moreover, Staworko et al. [24] investigate the complexity and expressiveness of ShEx for RDF, and propose two alternative semantics, single-type and multi-type; to this end, Boneva et al. [25] formalize the semantic of ShEx 2.0 and prove the semantics of it. In addition, Shapes Constraint Language (SHACL) [14] is newly published by the RDF data Shapes Working Group, to validate RDF graphs against a set of conditions. ShEx is a W3C Community Group specification while SHACL Core and SHACL-SPARQL are a W3C Recommendation, and both two technologies are maintained as separate solutions. ShEx has a limited ability to combine “shapes graphs”. In fact, ShEx schemas are not necessarily RDF graphs. However, in SHACL those conditions are provided as shapes and other constructs expressed in the form of an RDF graph, i.e., “shapes graphs”. Therefore, in Section V-A, we leverage SHACL to optimize our basic idea by inferring shapes for RDF resources in RDF data.

In [26], the survey pointed out that specialized in-memory systems, such as AdPart [27] or TriAD [28] provided the best performance, however, these two systems assumed that the data must fit in the cumulative memory of the computing cluster. Whereas, our work does not have such a restriction.
Moreover, this survey referred to that MapReduce based systems, e.g. H2RDF+ [29], are also an acceptable alternative, but such kind of systems leverage large amounts of indexes to accelerate lookups with high selectivity. In addition, the survey also mentioned that the startup costs of some systems, e.g., S2RDF [6], severely limit their applicability to large datasets. In this work, we focused on the analytical processing scenario of RDF graphs without any prebuilt indexes. Therefore, this is unfair to compare our work with those that use intensive indexes. To this end, we did the comparison with S2X and SHARD that are the most relevant works to this paper.

III. PRELIMINARIES

In this section, we introduce background definitions which will be used in our subgraph matching algorithms.

A. RDF AND CONJUNCTIVE QUERIES (CQ)

An RDF dataset is a set of triples, each of which can be represented as a directed edge. Thus, the entire RDF dataset is modeled as a directed labeled graph.

Definition 1 (RDF Graph): Let $U$ and $L$ be disjoint infinite sets of URIs and literals, respectively. A tuple $(s, p, o) \in U \times U \times (U \cup L)$ is called an RDF triple, where $s$ is the subject, $p$ is the predicate (a.k.a. property), and $o$ is the object. A finite set of RDF triples is called an RDF graph.

The RDF graph defined in this paper does not consider blank nodes. The reason is that the problem we focus on in this work is orthogonal to the inclusion of blank nodes. The omission of blank nodes in our RDF definition could make our discussion of our method more concise. Given an RDF graph $G$, let $V, E$, and $\Sigma$ denote the set of vertices, edges, and edge labels in $G$, respectively. Formally, $V = \{ s \mid (s, p, o) \in G \} \cup \{ o \mid (s, p, o) \in G \}$, $E \subseteq V \times V$, and $\Sigma \subseteq \{ p \mid (s, p, o) \in G \}$.

Let $Var$ be an infinite set of variables disjoint from $U$ and $L$, and the name of every element in $Var$ starts with the character $\$$. Conventionally (e.g., $\$? in Var). A triple $(s, p, o) \in (U \cup Var) \times (U \cup L \cup Var)$ is called a triple pattern.

Definition 2 (Query Graph): Given an RDF graph $G$, a CQ $Q$ over $G$ is defined as: $Q(z_1, \ldots, z_n) \leftarrow \bigwedge_{1 \leq i \leq m} p_i$, where $p_i = (x_i, a_i, y_i)$ is a triple pattern, $x_i, y_i \in V \cup Var$, $a_i \in \Sigma \cup Var$, $z_j \in \{ x_i \mid 1 \leq i \leq m \} \cup \{ y_i \mid 1 \leq i \leq m \}$. A CQ $Q$ is also referred to as a query graph $\mathcal{G}_Q$.

Before giving the definition of subgraph matching, we recapitulate certain definitions about the mapping. For a mapping $\mu$, $\text{dom}(\mu)$ is its domain. Two mappings $\mu_1$ and $\mu_2$ are called compatible denoted as $\mu_1 \sim \mu_2$, if for every element $v \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$ it holds that $\mu_1(v) = \mu_2(v)$. Furthermore, the set-union of two compatible mappings, i.e., $\mu_1 \cup \mu_2$, is also a mapping.

Definition 3 (Subgraph Matching): The semantics of a CQ $Q$ over an RDF graph $G$ is defined as: (1) $\mu$ is a mapping from vertices in $\bar{x}$ and $\bar{y}$ to vertices in $V$, where $\bar{x} = (x_1, \ldots, x_m)$, $\bar{y} = (y_1, \ldots, y_n)$; (2) $(G, \mu) \models Q$ iff $(\mu(x_i), \mu(a_i), \mu(y_j)) \in E$ and the labels of $x_i$, $a_i$, and $y_j$ are the same as that of $\mu(x_i)$, $\mu(a_i)$, and $\mu(y_j)$, respectively, if $x_i, a_i, y_j \not\in \text{Var}$; and (3) $\Omega_Q$ is the set of $\mu(z)$, where $z = (z_1, \ldots, z_n)$, such that $(G, \mu) \models Q$. $\Omega_Q$ is the answer set of the subgraph matching query $G_Q$ over $G$.

From the above definition, we can see that a subgraph matching query is actually a basic graph pattern (BGP), which is the most common query pattern used in SPARQL.

B. PREGEL PARALLEL GRAPH COMPUTATIONAL MODEL

Pregel is a vertex-centric parallel computational model introduced by Google for processing large-scale graph data [13]. Based on the Bulk Synchronous Parallel (BSP) [21], a Pregel computation is composed of a sequence of iterations, named supersteps, which are separated by global synchronization barriers.

The input to Pregel computation is a directed graph in which each vertex is identified with a unique identifier and is associated with a modifiable, user defined value. In our paper, the data graph of Pregel computation is an RDF graph. Let $I_V$ and $D_V$ denote the set of identifiers and values of vertices in $G$, respectively. For each vertex $v \in V$, the following functions are defined: (1) $\text{id}: V \rightarrow I_V$ returns the unique identifier for $v$; (2) $\text{val}: V \rightarrow D_V$ returns the value associated with $v$; and (3) $\text{inEdges}$ (resp. $\text{outEdges}$): $V \rightarrow \mathcal{P}(E)$ returns the set of incoming (resp. outgoing) edges of $v$. For each edge $e \in E$, the following functions are defined: (1) $\text{p}: E \rightarrow \Sigma$ returns the label of $e$; and (2) $\text{srcId}$ (resp. $\text{dstId}$): $E \rightarrow I_V$ returns the source (resp. destination) vertex identifier for $e$.

Each vertex $v \in V$ in RDF graph is in one of the two states, i.e., active and inactive. The function state: $V \rightarrow \{\text{active}, \text{inactive}\}$ gets the current state of a vertex. The function $\text{voteToHalt}$: $V \rightarrow \emptyset$ is called by a vertex $v$ to make $\text{state}(v) = \text{inactive}$. Let $\text{Msg}$ be the set of messages. The function $\text{sendMsg}$: $I_V \times I_V \times \text{Msg} \rightarrow \emptyset$ sends a message $m$ from one vertex $v_1$ to $v_2$.

Definition 4 (Parallel Computation): Let $G = (V, E)$ be an RDF graph, the Pregel parallel computation is defined as a sequence of supersteps on the active vertices. In superstep 0, all the vertices are active. An inactive vertex will be reactivated by receiving messages sent to it. Within a superstep, the user-defined function $\text{compute}$: $V \times \mathcal{P}(\text{Msg}) \rightarrow \emptyset$ is executed by every active vertex in parallel. In $\text{compute}(v, \text{Msg})$, the vertex $v$ can: (1) get the number of current superstep by $\text{superstep}$; (2) receive messages sent to $v$ in the previous superstep; (3) get or update the value of $v$ by $\text{val}(v)$; (4) get $\text{inEdges}(v)$ and $\text{outEdges}(v)$; (5) deactivate $v$ by $\text{voteToHalt}$; and (6) send messages to other vertices to be received in the next superstep by $\text{sendMsg}(v)$. The parallel computation terminates when there is no message to send.

IV. THE SP-TREE ALGORITHM

In this section, we present the process which transforms a query graph into the SP-Tree. Then the SP-Tree-based RDF subgraph matching algorithm is described in detail, which is developed by using Pregel.
A. SP-TREE OF A QUERY GRAPH

Given a query graph $G_Q$, we assume that $G_Q$ is a connected graph, otherwise, every connected component is considered as a query graph. In an undirected graph $G$, the geodesic distance between two vertices is the number of edges in a shortest path connecting them, and the eccentricity $\epsilon(v)$ of a vertex $v$ is the greatest geodesic distance between $v$ and any other vertex in $G$. Then a central vertex in an undirected graph is a vertex $v$ such that $v = \arg \min_{v \in V} \epsilon(v)$. When the query graph $G_Q$ is viewed as an undirected graph $G'_Q$, let $u_r$ be a central vertex in $G'_Q$. The SP-Tree of query graph $G_Q$ is defined as follows:

**Definition 5 (SP-Tree):** Given a query graph $G_Q = (V_Q, E_Q)$, let $sp(u, v)$ denote the shortest path from vertex $u$ to $v$, where $u, v \in V_Q$ and $|sp(u, v)|$ is the length of the path. The SP-Tree of $G_Q$ is a tree, denoted by $T = (u_r, V_T, E_T)$, where (1) $u_r = \arg \min_{v \in V_Q} \max_{u \in V_Q} |sp(u, v)|$ is the root, where $u_r$ is a central vertex in $G'_Q$; (2) $V_T$ is a set of nodes, for each vertex $v \in V_Q$, there exists at least one node $u \in V_T$, the label of $v$ is the same as that of $u$; (3) $E_T$ is the set of edges, for each edge $e = (x, a, y) \in E_Q$, there exists an edge $e' = (x, a, \text{forward}, y) \in E_T$ or $e' = (y, a, \text{reverse}, x) \in E_T$, where $e' = (x, a, \text{forward}, y)$ (resp. $e' = (y, a, \text{reverse}, x)$) denotes $x$ (resp. $y$) is the parent of $y$ (resp. $x$) and the direction of $e'$ from $x$ (resp. $y$) to $y$ (resp. $x$) is forward (resp. reverse).

The vertex $u_r$ is defined as the minimum height of SP-Tree $T$ which may not be unique. Our method randomly selects one of these $u_r$s and searches $G_Q$ from $u_r$ to span the corresponding SP-Tree $T$, which is shown in Algorithm 1. Please note that the SP-Tree is a variant of the conventional spanning tree structure.

**Example 1:** Given a CQ $Q_2 = (\exists x, \text{portrayer}, ?z) \land (\exists y, \text{starring}, ?z) \land (\exists x, \text{lastAppearance}, ?y) \land (\exists x, \text{series}, \text{T.Serise}) \land (\exists y, \text{editing}, ?w) \land (\exists y, \text{producer}, ?d)$, the query graph of $Q_2$ is shown in Figure 2(a). According to Definition 5, vertices $?x$, $?y$, and $?z$ can be the root in an SP-Tree of $Q_2$. The SP-Tree $T_{Q_2}$ rooted at $?z$ is shown in Figure 2(b). Note that the query graph $G_{Q_2}$ contains a cycle, thus there exist two nodes labeled $?y$ in $T_{Q_2}$.

B. SUBGRAPH MATCHING ALGORITHM USING PREGEL

Next, we show how to use Pregel to match nodes of an SP-Tree $T$ over an RDF graph $G$ to answer the subgraph matching query in a bottom-up manner. First, every vertex in $G$ matches leaves of $T$ in parallel. Then vertices in $G$, matching to at least one leaf, send the partial matching results of these leaves to their neighbouring vertices. In the next supersteps, vertices in $G$ who receive massages match parents of those nodes matched in the previous superstep. Similarly, these vertices send messages about partial matching results of subtrees rooted at these parents. Intuitively, in each superstep, one level of the SP-Tree $T$ is matched. This process is iterated until the root in $T$ is matched. Finally, query results are collected from vertices that can match the root of $T$.

We first prepare some auxiliary notations in order to give the SP-Tree algorithm. Given a node $u$ in $V_T$, $\Omega(u)$ is used to denote a partial matching result of the subtree rooted at $u$, i.e., a set of mappings. Let $\text{Child}(u)$ and $\text{Leaves}$ denote the set of children of node $u$ and the set of leaves in $T$, respectively. Given a node $u$ in $V_T$, we assume...
a 3-tuple \( \text{list}(u) = (\text{lab}, \text{dir}, \text{parent}) \). For every edge \( e = (u_1, a, \text{forward/reverse}, u_2) \in E_T \), we have a 3-tuple \( \text{list}(u_2) = (a, \text{reverse/forward}, u_1) \), where \( \text{list}(u_2).\text{lab} \) (i.e., \( a \)) is the label of \( e \), \( \text{list}(u_2).\text{dir} \) (i.e., \( \text{reverse/forward} \)) is the direction from \( u_2 \) to \( u_1 \), which is opposite to \( e \), and \( \text{list}(u_2).\text{parent} \) (i.e., \( u_1 \)) is the parent of \( u_2 \).

In our algorithm, the set of messages \( Msg \) mentioned in Section III-B is a set of 2-tuples \( (u, \Omega(u)) \). The set of values of vertices in RDF graph \( D_V \) mentioned in Section III-A is a set of 3-tuples \( (lab, M_p, Res) \), where for every vertex \( v \in V \), (1) \( lab \) is the label of vertex \( v \); (2) \( M_p = \{(u_i, \Omega(u_i)) \mid (u_i, \Omega(u_i)) \in Msg \land u_i \in \text{Child}(u)\} \), where the label of node \( u \) is \( lab \) or a variable and \( Msg \) denotes the messages that vertex \( v \) can receive; and (3) \( Res \) is the query result in vertex \( v \), i.e., \( \Omega(u_i) \), where the label of \( u_i \) in \( \text{Var} \cup \{\text{lab}\} \). Our SP-Tree algorithm is shown in Algorithm 2.

**Algorithm 2: SP-Tree-Match(G, T)**

**Input**: An RDF graph \( G \), the SP-Tree \( T \) of a given CQ \( Q \).

**Output**: The answer set \( \Omega_Q \).

1. \( \Omega_Q \leftarrow \emptyset; \)
2. \( Msg \leftarrow \emptyset; \)
3. for \( \text{superstep()} = 0 \) to \( d \) do // \( d \) is the height of \( T \)
   4. every active vertex \( v \in V \) calls \( \text{compute}(v, Msg) \) in parallel;
5. foreach \( v \in V \)
6. \( \Omega_Q \leftarrow \Omega_Q \cup \text{val}(v).\text{Res}; \)
7. return \( \Omega_Q \);

The initial message is an empty set (line 2) in Algorithm 2. Then the iterative computation is executed in parallel (lines 3-4), when the iterations terminate, the subgraph matching result consists of a subset of vertices in \( G \). After iterations, the algorithm traverses each vertex in \( G \) to collect the query results (lines 5-6).

Algorithm 3 follows an iterative manner of computation: (1) In superstep 0, every vertex in \( G \) tries to match leaves in SP-Tree \( T \), then adds the partial matching results of these leaves to \( \text{OutMsg} \) (lines 2-6), and sends these messages to its neighbouring vertices (lines 22-30). (2) In the next supersteps, active vertices execute the local computation to handle received messages \( Msg \) (lines 7-21). For every message \( m = (u, \Omega(u)) \in Msg \), an active vertex \( v \) tries to match parent \( u' \) of node \( u \) and adds \( m \) to \( M_p \) (lines 10-15). If vertex \( v \) receives partial results of subtrees rooted at all children \( u_i \) of \( u' \), the vertex \( v \) can match against the node \( u' \) (lines 16-21). Then the algorithm does Cartesian product operations on \( \Omega(u_i) \) and merges mappings in \( \Omega(u_i) \) to obtain \( \Omega(u') \), i.e., a partial matching result of the subtree rooted at node \( u' \) (line 18). If the vertex \( v \) can match the root \( u_r \), it adds the subgraph matching result to \( \text{val}(v).\text{Res} \) and no longer sends messages, otherwise adds the partial matching result to \( \text{OutMsg} \) (lines 20-21). (3) In each superstep, for every message \( m = (u, \Omega(u)) \in \text{OutMsg} \) in vertex \( v \), it sends \( m \)

**Algorithm 3: Compute(v, Msg)**

**Input**: Vertex \( v \), messages sent to \( v \) in the previous superstep: \( Msg \)

**Output**: state(v)

1. \( \text{OutMsg} \leftarrow \emptyset; \) // send to neighbouring vertices
2. if \( \text{superstep()} = 0 \) then // matches all leaves
   3. foreach \( l_i \in \text{Leaves do} \)
   4. Let \( lab' \) be the label of \( l_i \);
   5. if \( lab' \in \text{Var} \cup \{\text{lab}\} \) then
      6. \( \mu \leftarrow \{(lab', lab)\}, \text{OutMsg} \leftarrow \text{OutMsg} \cup \{(l_i, \{\mu\})\}; \)
   7. else // in supersteps 1 to \( d \), handle received messages \( Msg \)
      8. \( \text{MatchV} \leftarrow \emptyset; \) // the set of nodes in \( T \) that \( v \) can match
      9. foreach \( m = (u, \Omega(u)) \in \text{OutMsg} \) do
         10. Let \( u' \) be the parent of node \( u \) and \( lab' \) be the label of \( u' \);
         11. if \( lab' \in \text{Var} \cup \{\text{lab}\} \) then // \( v \) tries to match \( u' \)
             12. \( \text{MatchV} \leftarrow \text{MatchV} \cup \{u'\}; \)
         13. if \( \exists m' \in M_p \land m'.u = u \) then
             14. \( M_p \leftarrow M_p \cup \{m\}; \)
         15. else \( M_p \leftarrow M_p \cup \{m\}; \)
      16. foreach \( u' \in \text{MatchV} \) do // \( v \) matches \( u' \)
         17. if \( \forall u_i \in \text{Child}(u') \) then \( 1 \leq i \leq t \):
              18. \( \exists (u_i, \Omega(u_i)) \in M_p \), then
                 19. \( \mu \leftarrow \{(lab', \text{lab})\}; \)
                 20. \( \Omega(u') \leftarrow \{\mu_1 \cup \ldots \cup \mu_t \cup \mu \mid \mu_i \in \Omega(u_i) \land \mu_1 \sim \ldots \sim \mu_t \sim \mu \}; \)
                 21. \( M_p \leftarrow M_p \setminus \{(u_i, \Omega(u_i)) \mid 1 \leq i \leq t\}; \)
         22. if \( u' = u \), then \( \text{Res} \leftarrow \text{Res} \cup \Omega(u'); \)
         23. else \( \text{OutMsg} \leftarrow \text{OutMsg} \cup \{(u', \Omega(u'))\}; \)
      24. foreach \( m = (u, \Omega(u)) \in \text{OutMsg} \) do
         25. if \( \text{list}(u).\text{dir} = \text{forward} \)
            26. \( \text{OutEdges}(v) \)
               27. \( \text{sendMsg}(\text{srcId}(e), \text{dstId}(e), m); \)
         28. else
            29. \( \text{inEdges}(v) \)
               30. \( \text{sendMsg}(\text{dstId}(e), \text{srcId}(e), m); \)
      31. return \( \text{voteToHalt}(); \)
to its neighbouring vertex $v'$, if the label and direction of the edge $(u, u')$ in $T$ are the same as that of edge $(v, v')$ in $G$ (lines 22-30). (4) At the end of each local computation, every active vertex deactivates itself (line 31).

Example 2: Consider the SP-Tree $T_Q$ in Figure 2(b) over the RDF graph $G$ in Figure 1. As listed in Table 1, all leaves, i.e., $Leaves = \{\text{?y}, \text{T.Series}, \text{?d}, \text{?w}\}$, are matched in superstep 0. Then active vertices match parents $?z$, $?x$, and $?y$ of these leaves. In superstep 1, since vertices Marton, CiaranH, SamWor, RosieH, and TomKen receive messages about $\Omega(?y)$, and $Child(?z) = \{?z\}$, they can match node $?z$. Similarly, vertices Fury, TransF.3, T.Series, and TheDebt can match node $?y$. Finally, in superstep 2, the root $?x$ is matched and the query result of $Q_2$ is formed in vertex ClaryW.

Theorem 1: Given a CQ $Q$ and the corresponding SP-Tree $T$ over an RDF graph $G$, let $d$ denote the height of $T$. Algorithm 2 is correct and evaluates $Q$ in $d$ supersteps.

Proof (Sketch): (1) Let $d_i$ denote the height of node $u_i$ in $T$. In Algorithm 2, the node $u_i$ is matched in superstep $d_i$. Therefore, in superstep $d$, all nodes in $T$ are matched which guarantees the correctness of Algorithm 2. (2) A path in $T$ from a leaf $u$ to the root $r$ is a sequence $\rho = u_0e_{i_1}u_1...u_{n-1}e_{i_n-1}u_n$, where $n \geq 1$, $u_0 = l$, $u_n = r$, $u_{i_{n-1}} = \text{list}(u).parent$, and $e_i = (\text{list}(u).lab, \text{list}(u).dir)$ for every $i \in \{0,\ldots,n-1\}$. $|\rho|$ is the length of this path. Let $\rho_m$ denote the longest path in $T$, then $|\rho_m| = d$. Assume that Algorithm 2 evaluates $Q$ in $k$ supersteps: (i) $k < d$, in superstep $s$, $s < k$, node $u_s$ is matched to some vertices in $G$. After superstep $k$, for path $\rho_m$, these nodes $u_s$, $k < s < d$ have not been matched. Thus the result query of $Q$ cannot be obtained; (ii) $k > d$, in superstep $s$, $s < d$, node $u_s$ is matched. In superstep $d$, the root $u_d$ is matched and no messages are sent, i.e., the iteration computation is terminated. Therefore, $k = d$. \hfill $\square$

C. COST ANALYSIS

Given an SP-Tree $T$ over an RDF graph $G = (V, E)$, for each superstep, we consider three categories of costs, which are defined as follows: (1) $w_v$, the cost of the local computation in vertex $v$, $v \in V$; (2) $h_i$, the number of messages sent or received by vertex $v$; and (3) $l$, the cost of the barrier synchronization at the end of the superstep. So the cost of one superstep for all vertices is $w + h_1 + l$, where $w = \max_{v \in V} w_v$, $h = \max_{v \in V} h_v$. In addition, for any internal node (i.e., non-leaf and non-root) $u$ in $T$, let $\text{des}(u)$ denote the set of descendant nodes of node $u$.

For our algorithm, (1) in superstep 0, $w_0 = |Leaves|$ and $h_0 = |deg_m| \times |Leaves|$, where $|deg_m|$ denote the maximum degree in $G$; (2) in superstep $s$, $s < d$, $w_s = |deg_m|\text{des}_m$ and $h_s = |deg_m|\text{des}_m+1$, where $\text{des}_m = \max_{u \in V} |\text{des}(u)|$ and the height of node $u$ is $s$; and (3) in superstep $d$, $w_d = |deg_m|\text{deg}_T|\text{deg}_T^{-1}|$, $h_d = 0$. Therefore, the cost $C$ of our SP-Tree algorithm is the sum of the costs of each superstep:

$$C = \sum_{s=0}^{d-s} w_s + g \sum_{s=0}^{d-1} h_s + dl = O(|deg_m|\text{deg}_T|)$$

V. OPTIMIZATION TECHNIQUES

In this section, two optimization techniques are applied to improve the efficiency of the SP-Tree algorithm. One is RDF shape filtering for pruning invalid local computations and messages passed, and the other is postponing Cartesian product operations. Although the techniques do not change the complexity of the SP-Tree algorithm, in practice, they can improve query efficiency significantly.

A. RDF SHAPE FILTERING

The SHACL [14] is a language for validating RDF graphs against a set of conditions. These conditions are provided as shapes which are URIs and other constructs expressed in the form of an RDF graph, called a shape graph. The SHACL Core language defines two types of shapes including node shapes and property shapes whose SHACL types are sh:NodeShape and sh:PropertyShape, respectively. In this paper, we leverage node shapes which specify constraints that need to be met with respect to focus nodes, i.e., resources in RDF graphs.

Given an RDF graph $G = (V, E)$, the triple $(s, \text{rdf:type}, C) \in G$ declares that the resource $s$ is an instance of class $C$ and denoted by $s \in C$, e.g., the triple (GaiusJ, rdf:type, FictionalCharacter) in Fig. 1 denotes that GaiusJ is an instance of the class FictionalCharacter. Other such triples in Fig. 1 are omitted. In general, most resources belong to one or more classes in the real world. For example, most of resources in DBpedia dataset belong to at least one class. This RDF Shape Filtering optimization strategy can be applied only when resources have rdf:type. In fact, for most real-world datasets, it is rare that an resource does not have at least one (s, rdf:type, C) constraint. The class set of $G$ is $\mathcal{C}(G) = \{C \mid (s, \text{rdf:type}, C) \in G\}$. We generate the RDF shape of $G$ using the following definition:

Definition 6 (RDF Shape): Given an RDF graph $G$, for each class $C \in \mathcal{C}(G)$, a node shape $s_C$ is defined by declaring
(sC, rdf:type, sh:NodeShape). The RDF Shape of G is denoted by S(G) = \{(sC, rdf:type, sh:NodeShape) | C ∈ C(G) \} ∪ \{(sC, sh:property, p) | (s, p, o) ∈ G ∧ s ∈ C\}. The constraints of sC is denoted by P(sC) = \{p | (sC, sh:property, p) ∈ S(G)\}. For any resource r ∈ C, r is a focus node of sC.

Example 3: As shown in Fig. 1, we have P(sFictionalCharacter) = \{ producer, editing, starring, rdf:type \}, P(sFictionalCharacter) = \{ producer, lastAppearance, series, rdf:type \}.

Given a query graph GQ and the SP-Tree T of GQ over an RDF graph G, for every vertex u ∈ VQ, let N(u) = \{(u, p, o) ∈ GQ\}. The RDF shape of G is used to filter out local computations and messages passed as follows: (1) when matching node u in T to vertex v in G, if v ∈ C1 ∪ ... ∪ Cn and N(u) ∈ P(sC1) ∪ ... ∪ P(sCn), the local computation at vertex v is pruned. Note that a large number of intermediate matching results of leaves in T are reduced by pruning such computation; (2) when vertex v1 sends message m = \{(u, Ω(u))\} to vertex v2, if v2 ∈ C1 ∪ ... ∪ Cn and N(list(u).parent) ∈ P(sC1) ∪ ... ∪ P(sCn), the message passed is also pruned.

Example 4: When the leaf of SP-Tree Tg in Fig. 2(b) is matched to vertex GaiusJ in RDF graph G in Fig. 1, since GaiusJ is FictionalCharacter and N(y) = \{ producer, editing, starring \} ∈ P(sFictionalCharacter), the local computation at vertex GaiusJ can be pruned in our optimization method.

B. POSTPONING CARTESIAN PRODUCT OPERATIONS

Given an RDF graph G and an SP-Tree T, when matching a non-leaf node u, Child(u) = \{u1, ..., u\}, Algorithm 3 computes Ω(u) by the Cartesian product operation \{μ1 ∪ ... ∪ μn | μ | μ1 ∈ Ω(u1), μ2 ~ ... ~ μn ~ μ}, i.e., \{Ω(u1) | ... | Ω(un)\} merge operations of mappings. However, not all elements of Ω(u) may contribute to the final query results. To this end, we decompose T into a set of paths, called SP-Paths, to postpone and reduce invalid Cartesian product operations.

The SP-Path is defined as:

Definition 7 (SP-Path): Given an SP-Tree T = \{(ui, Vi, Ei)\}, for every leaf l ∈ Leaves, the SP-Path of l is defined as a sequence ρ = u0e0u1 ... un−1en−1u, where n > 0, u0 = l, un = u, ur+1 = list(u).parent, and e = (list(u).lab, list(u).dir).

Given an SP-Tree T, the SP-Paths of T can be easily obtained. Our optimized method matches a non-leaf node u in different SP-Paths including u and computes Ω(u) by \{μ1 ∪ μ | μ1 ∈ Ω(u1), μ2 ~ μ\} without Cartesian product operations which are postponed to the last step.

Example 5: Tq in Figure 2(b) can be decomposed into four SP-Paths, e.g., the SP-Path of leaf Fury, \( ρ = \text{?w} (\text{editor, reverse}) \), is highlighted in green. Considering Tq, over G in Figure 1, Algorithm 3 computes Ω(?) in vertex Fury by 6 merge operations of mappings, i.e., \( \{Ω(?)\} \). In our optimized method, the node ? is matched to vertex Fury in two SP-Paths and the Cartesian product operation is postponed to match root ?x. As shown in Figure 1, the partial results of Ω(?) in Fury cannot contribute to the final answer, thus the Cartesian product operations in Fury are reduced.

VI. EXPERIMENTS

We have carried out extensive experiments on both synthetic and real-world RDF graphs to verify the efficiency and scalability of SP-Tree and compared it with the optimized method, SHARD using MapReduce, and S2X using GraphX in Spark [15]. The other methods are not included in our comparison for the following reasons: (1) the distributed gStore system builds the VS*-tree index to speed up queries, it is unfair to compare with it; (2) S2RDF also uses the ExtVP indexes to improve query efficiency, however its semi-join preprocessing is expensive, e.g., for the LUBM400 dataset used in our experiments, the time of generating ExtVP tables is 29.1 min; and (3) these methods [4], [7]–[9], [11] evaluate the problem of subgraph matching over undirected and (un)labeled graphs, which cannot be easily adapted to RDF graphs. It is worth noting that our algorithm is orthogonal to the graph partitioning and placement strategies in the cluster environment. In particular, for the implementation of SP-Tree, we use the default graph partitioner employed by the Spark GraphX framework. Moreover, the source code of our SP-Tree method is available on GitHub.1

A. SETTINGS

The prototype program, which is implemented in Scala using GraphX over Spark, is deployed on an 8-site cluster connected by a gigabit Ethernet. Each site has one Intel(R) Core(TM) i7-7700 CPU with 4 cores, 16GB memory, and 500GB disk storage. We used Hadoop 2.7.4 and Spark 2.2.0 and all the experiments are carried out on Linux (64-bit CentOS) operating systems.

We used three RDF datasets, including synthetic datasets, i.e., LUBM,2 WatDiv,3 and real-world dataset DBpedia4 in our experiments. (1) LUBM, as a standard RDF synthetic

<table>
<thead>
<tr>
<th>Dataset</th>
<th>TestGraphs</th>
<th>RDFFiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM400</td>
<td>943,944</td>
<td>493,944</td>
</tr>
<tr>
<td>LUBM600</td>
<td>844,239</td>
<td>549,754</td>
</tr>
<tr>
<td>WatDiv1M</td>
<td>159,114</td>
<td>1,275,577</td>
</tr>
<tr>
<td>WatDiv10M</td>
<td>1,02,937</td>
<td>10,914,977</td>
</tr>
<tr>
<td>DBpedia</td>
<td>20,652,529</td>
<td>23,445,442</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query</th>
<th>E</th>
<th>S</th>
<th>F</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM</td>
<td>Q1, Q4</td>
<td>Q2</td>
<td>Q</td>
<td>Q3</td>
</tr>
<tr>
<td>WatDiv</td>
<td>Q1, Q2, Q3</td>
<td>Q4</td>
<td>Q7</td>
<td>Q8</td>
</tr>
<tr>
<td>DBpedia</td>
<td>Q5, Q6, Q7</td>
<td>Q8</td>
<td>Q9</td>
<td>Q2, Q3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>TestGraphs</th>
<th>RDFFiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM400</td>
<td>943,944</td>
<td>493,944</td>
</tr>
<tr>
<td>LUBM600</td>
<td>844,239</td>
<td>549,754</td>
</tr>
<tr>
<td>WatDiv1M</td>
<td>159,114</td>
<td>1,275,577</td>
</tr>
<tr>
<td>WatDiv10M</td>
<td>1,02,937</td>
<td>10,914,977</td>
</tr>
<tr>
<td>DBpedia</td>
<td>20,652,529</td>
<td>23,445,442</td>
</tr>
</tbody>
</table>

1https://github.com/jingjia3/SP-Tree
2http://swat.cse.lehigh.edu/projects/lubm/
3http://dsg.uwaterloo.ca/watdiv/
4http://wiki.dbpedia.org/downloads-2016-10
TABLE 4. The sizes of queries.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Query</th>
<th>Vertices</th>
<th>Edges</th>
<th>Query</th>
<th>Vertices</th>
<th>Edges</th>
<th>Query</th>
<th>Vertices</th>
<th>Edges</th>
<th>Query</th>
<th>Vertices</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM</td>
<td>Q1</td>
<td>1</td>
<td>2</td>
<td>Q2</td>
<td>4</td>
<td>4</td>
<td>Q3</td>
<td>1</td>
<td>3</td>
<td>Q4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Q5</td>
<td>4</td>
<td>4</td>
<td>Q6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q7</td>
<td>2</td>
<td></td>
<td>Q8</td>
<td>1</td>
<td>2</td>
<td>Q9</td>
<td>1</td>
<td>2</td>
<td>Q10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Q11</td>
<td>2</td>
<td></td>
<td>Q12</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WatDiv</td>
<td>Q1</td>
<td>1</td>
<td>2</td>
<td>Q2</td>
<td>4</td>
<td>4</td>
<td>Q3</td>
<td>1</td>
<td>3</td>
<td>Q4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Q5</td>
<td>4</td>
<td>4</td>
<td>Q6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q7</td>
<td>2</td>
<td></td>
<td>Q8</td>
<td>1</td>
<td>2</td>
<td>Q9</td>
<td>1</td>
<td>2</td>
<td>Q10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Q11</td>
<td>2</td>
<td></td>
<td>Q12</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q13</td>
<td>4</td>
<td></td>
<td>Q14</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DBpedia</td>
<td>Q1</td>
<td>1</td>
<td>2</td>
<td>Q2</td>
<td>4</td>
<td>4</td>
<td>Q3</td>
<td>1</td>
<td>3</td>
<td>Q4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Q5</td>
<td>4</td>
<td>4</td>
<td>Q6</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q7</td>
<td>2</td>
<td></td>
<td>Q8</td>
<td>1</td>
<td>2</td>
<td>Q9</td>
<td>1</td>
<td>2</td>
<td>Q10</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Q11</td>
<td>2</td>
<td></td>
<td>Q12</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q13</td>
<td>4</td>
<td></td>
<td>Q14</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 3. Execution time of different queries over all datasets.

B. VERIFYING THE EFFICIENCY AND SCALABILITY

We compared our optimized method $\text{SP-Tree}_{\text{opt}}$, basic algorithm $\text{SP-Tree}$, S2X, and SHARD. Since the size of query graphs is much less than RDF graphs, the time of Algorithm 1 can be ignored.

1) EFFICIENCY

Experiments were conducted to verify the query efficiency of our method. As shown in Figure 3, $\text{SP-Tree}_{\text{opt}}$ demonstrates the best query efficiency on all datasets and queries. The query times of $\text{SP-Tree}$ are also much better than that of S2X and SHARD. For $Q_2$, $Q_9$ in LUBM40 and $C_2$ in DBpedia, S2X reported errors. When querying $F_3$ (resp. $C_2$) over WatDiv100M, it can be observed that S2X cannot finish in the time limit ($1 \times 10^4$ s), denoted by INF, while $\text{SP-Tree}_{\text{opt}}$ and $\text{SP-Tree}$ can return the answers within 20s (resp. 48s) and 104s (resp. 100s), respectively. For the remaining 37 queries, our $\text{SP-Tree}_{\text{opt}}$ algorithm is up to 900 times and on average 70 times faster than S2X. Similarly, $\text{SP-Tree}_{\text{opt}}$ is up to 60 times and on average 28 times faster than SHARD. Thus $\text{SP-Tree}_{\text{opt}}$ on average, outperforms S2X and SHARD by an order of magnitude.

We believe that the reasons include: (1) S2X and SHARD joined the matching results of all triple patterns in queries, incurring expensive cost; (2) $\text{SP-Tree}$ did Cartesian product operations on matching results of subtrees in $\text{SP-Tree}$, which consumed a lot of memory space; and (3) in $\text{SP-Tree}_{\text{opt}}$, a part of local computations and messages passed were pruned by...
utilizing RDF shapes, a large number of Cartesian operations were postponed and reduced, and for all active vertices in the final superstep, the match results of SP-Paths in every such vertex were joined in parallel, instead of joining all match results together.

2) SCALABILITY ON THE NUMBER OF SITES

We conducted experiments on three datasets, i.e., LUBM400, WatDiv100M, and DBpedia, with the number of cluster sites varying from 4 to 8, as shown in Figure 4. During these experiments, we randomly selected one query from each of the 4 query categories over these three datasets. For most queries in all these methods, the query times decreased as the number of cluster sites increased. This is because when the number of sites increased, the degree of parallelism increased. As we can see, in S2X and SHARD, the query times of these four queries over LUBM400 decreased first and then increased as the number of cluster sites increased. The reason is most likely that when the degree of parallelism increases to a certain extent, the cost of communication increases and gradually takes up the main factor. Moreover, the experimental result over LUBM400 verifies that the communication cost with SP-Treeopt is relatively low as the number of cluster sites increasing. Although, with the number of sites increasing, the average speedup ratio of SP-Treeopt is about 1.2 times of S2X over these three datasets. For query Q9 over LUBM400, S2X reported errors, so the related data is missing in Figure 4. Similarly, when querying C2 over WatDiv100M regardless of the number of cluster sites, S2X cannot finish in $1 \times 10^4$s.

3) SCALABILITY ON BIG RDF GRAPHS

Experiments were carried out over LUBM and WatDiv to verify the scalability on different sizes of datasets. We randomly selected one query from each of the 4 query categories over these two datasets. When changing the size of LUBM from LUBM4 to LUBM400, as shown in Figure 5, query times of all these methods have increased. It can be observed that the growth rates of query times in S2X and SP-Tree are higher than that of SP-Treeopt in most cases. We can also see that the growth rates of query times in SP-Treeopt are higher than that of SHARD, but the execution of SHARD takes much more time than our SP-Treeopt. Similarly, the result of experiments on WatDiv in Figure 5 also verified that SP-Treeopt is always the best one. When querying Q9 over LUBM400, S2X reported errors.

4) EFFICIENCY OF OPTIMIZATION TECHNIQUES

The average times of each query category were computed on all three datasets. Furthermore, the ratios of these average times of SP-Treeopt to that of SP-Tree is also computed, as shown in Figure 6. Compared with SP-Tree, the query
time of $\text{SP-Tree}_{\text{opt}}$ over these three datasets is reduced from 10.26% to 84.01%. The time improved of non-linear queries are more prominent, which verifies our intuition that when evaluating non-linear queries, there exist more intermediate computations and Cartesian product operations than linear queries need to be optimized. The experimental result demonstrates that $\text{SP-Tree}_{\text{opt}}$ reduced local computations, messages passed, and Cartesian product operations by a large margin. As we can see, the optimization effect on the complex queries over real-world dataset is the most significant.

VII. CONCLUSION

In this paper, we proposed the $\text{SP-Tree}$ method for efficiently answering the subgraph matching problem on big RDF graph data using the Pregel parallel graph computing model. Moreover, we also developed two optimization techniques, including RDF shapes filtering and postponing Cartesian product operations, to improve the basic $\text{SP-Tree}$ algorithm. Our extensive experimental results on both synthetic and real-world datasets have verified the efficiency and scalability of our method, which outperforms SHARD and S2X by one order of magnitude.

REFERENCES


