World Wide Web (2020) 23:1465–1496
https://doi.org/10.1007/s11280-019-00739-0

# Distributed Pregel-based provenance-aware regular path query processing on RDF knowledge graphs

**Xin Wang[1,2] · Simiao Wang[1,2] · Yueqi Xin[1,2] · Yajun Yang[1,2] · Jianxin Li[3] · Xiaofei Wang[1,2]**

Received: 7 February 2019 / Revised: 16 September 2019 / Accepted: 23 September 2019 /
Published online: 14 November 2019


**Abstract**
With the proliferation of knowledge graphs, massive RDF graphs have been published on the Web. As an essential type of queries for RDF graphs, Regular Path Queries (RPQs) have been attracting increasing research efforts. However, the existing query processing approaches mainly focus on RPQs under the standard semantics, which cannot provide the *provenance* of the answer sets. We propose a distributed Pregel-based approach `DP2RPQ` to evaluating provenance-aware RPQs over big RDF graphs. Our method employs Glushkov automata to keep track of matching processes of RPQs *in parallel*. Meanwhile, three optimization strategies are devised according to the cost model, including vertex-computation optimization, message-communication reduction, and counting-paths alleviation, which can reduce the intermediate results of the basic `DP2RPQ` algorithm dramatically and overcome the counting-paths problem to some extent. The proposed algorithms are verified by extensive experiments on both synthetic and real-world datasets, which show that our approach can efficiently answer the provenance-aware RPQs over large RDF graphs. Furthermore, the RPQ semantics of `DP2RPQ` is richer than that of RDFPath, and the performance of `DP2RPQ` is still far better than that of RDFPath.

**Keywords** Regular path query · Provenance-aware · RDF graph · Pregel

## 1 Introduction

With the increasing popularity of knowledge graphs, *Resource Description Framework* (RDF) has been widely recognized as a flexible graph-like data model to represent large-scale knowledge bases. It has become essential to realize efficient and scalable query processing for big RDF graphs in various domains, such as social networking [23] and

✉ Xiaofei Wang
xiaofeiwang@tju.edu.cn

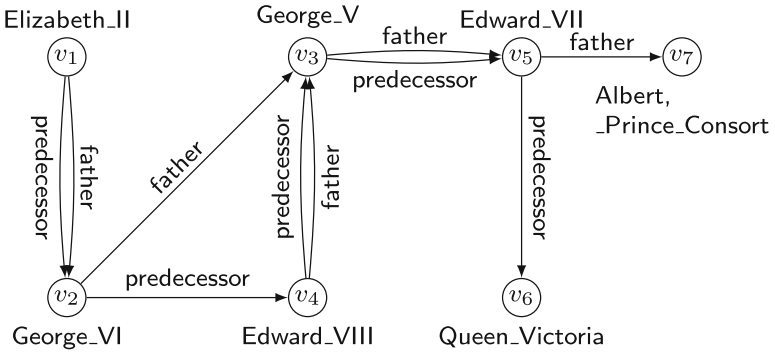
[1] College of Intelligence and Computing, Tianjin University, Peiyang Park Campus, Tianjin, China

[2] Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin, China

[3] School of Information Technology, Deakin University, Geelong, Australia

bioinformatics [14, 27], stored in distributed clusters. As one of the fundamental operations for querying graph data [6], regular path queries (RPQs) can explore RDF graphs in a navigational manner, which is an indispensable building block in most graph query languages. The latest version of the standard query language of RDF, SPARQL 1.1 [13], has provided the *property path* [16] feature which is actually an implementation of RPQ semantics. In particular, answering an RPQ $Q = (x, r, y)$ over an RDF graph $T$ is to find a set of pairs of resources $(v_0, v_n)$ such that there exists a path $\rho$ in $T$ from $v_0$ to $v_n$, where the label of $\rho$, denoted by $\lambda(\rho)$, satisfies the regular expression $r$ in $Q$.

However, from the above standard semantics of RPQs, we cannot tell what such a path $\rho$ from $v_0$ to $v_n$ looks like. To provide the *provenance* why a pair of resources in an RDF graph satisfies $Q$, we focus on the *provenance-aware* semantics of RPQs which actually returns a *subgraph* of the RDF graph consisting of all the "witness triples". For example, Figure 1a depicts an RDF graph $T_1$ excerpted from DBpedia [17], which shows predecessor and father relationships among seven British monarchs [25]. The RPQ $Q_1 = (x, (\mathsf{predecessor}|\mathsf{father})^+, y)$ asks to find pairs of monarchs $(v_0, v_n)$ such that $v_0$ can navigate to $v_n$ via one or more predecessor or father edges. The answers under the standard semantics to $Q_1$ are shown in Figure 1b. In contrast, the provenance-aware answer to $Q_1$ is a subgraph that contains all the paths whose labels satisfy $Q_1$. In this example, the subgraph (i.e., answer) is exactly $T_1$, which can efficiently encode the conventional answers to $Q_1$ in Figure 1b.




(a) An example RDF graph $T_1$

| $x$ | $y$ | $x$ | $y$ |
|---|---|---|---|
| Elizabeth_II | George_VI | George_VI | Queen_Victoria |
| Elizabeth_II | George_V | Edward_VIII | George_V |
| Elizabeth_II | Edward_VIII | Edward_VIII | Edward_VII |
| Elizabeth_II | Edward_VII | Edward_VIII | Albert,_Prince_Consort |
| Elizabeth_II | Albert,_Prince_Consort | Edward_VIII | Queen_Victoria |
| Elizabeth_II | Queen_Victoria | George_V | Edward_VII |
| George_VI | George_V | George_V | Albert,_Prince_Consort |
| George_VI | Edward_VIII | George_V | Queen_Victoria |
| George_VI | Edward_VII | Edward_VII | Albert,_Prince_Consort |
| George_VI | Albert,_Prince_Consort | Edward_VII | Queen_Victoria |

(b) The standard answers to RPQ $Q_1 = (x, (\mathsf{predecessor}|\mathsf{father})^+, y)$ on $T_1$

**Figure 1** An example RDF graph $T_1$ and answers to RPQ $Q_1$

Currently, there have been some research works on RPQs over RDF graphs under both standard and provenance-aware semantics. To answer RPQs under the standard semantics, some approaches leverage views [9] or other auxiliary structures, such as "rare labels" [15]. The RPQ evaluation system Vertigo [20] is implemented based on *Brzozowski's derivatives* using the Giraph parallel framework [2]. Wang et al. [26] employ the partial evaluation to obtain partial answers to RPQs in parallel and assemble the partial answers using an automata-based algorithm. However, the above methods may lead to potential large intermediate results and suffer from performance bottleneck when evaluating RPQs on large-scale RDF graphs. Although Dey et al. [11] have done the first work to investigate provenance-aware RPQs, they translate RPQs into standard Datalog queries, which is hardly scalable when evaluating on large RDF graph data. Another representative work [24] is based on product automata to evaluate provenance-aware RPQs, which may incur the costly construction process of product automata and excessive communications when handling large-scale RDF graphs.

A variety of parallel models and systems have been developed these years, such as Neo4j,[1] Trinity,[2] and BSP. (1) Neo4j is a graph database optimized for graph traversal, but it performs badly on a large distributed environment; (2) Trinity is a distributed system based on hypergraphs, and does not support regular path queries; (3) BSP models parallel computations in supersteps to synchronize communication among workers. Pregel [19] implements BSP with vertex-centric programming, where a superstep executes a user-defined function at each vertex in parallel. This vertex-centric approach works well with a series of iterative programs on graph data. Further, Pregel is more widely implemented in the mainstream big data platform, such as Spark and Giraph, which mostly rely on share-nothing architectures. In summary, Pregel is a state-of-the-art distributed graph computing framework.

Pregel is a computational model suitable for programs that can be expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate the graph topology. Since our proposed method for answering provenance-aware RPQs needs to traverse the graph, we can reasonably implement graph processing algorithms in a sequence of superstep iterations using Pregel.

To this end, in this paper, we propose a distributed *Pregel-based* parallel approach `DP2RPQ` to answering provenance-aware RPQs using *Glushkov automata*, which consists of a series of supersteps. The query processing starts with the vertices in an RDF graph to match against the states in the corresponding automaton of the RPQ; in each superstep, one hop of edges in the paths of the RDF graph are matched forward to obtain the intermediate partial answer to the provenance-aware RPQ.

In addition, we design several optimization strategies in three aspects: (1) to reduce vertex-computation cost, we design *edge-filtering* and *candidate-states* techniques to improve the performance of vertex computation, which can filter out those edges whose labels not occurring in $r$ and avoid traversals via outgoing edges of the vertex, respectively; (2) to reduce message-communication cost, *pruning-sending-messages* and *variable-length-byte encoding* techniques are proposed to reduce intermediate results and communication overhead significantly; (3) to address the *counting-paths* problem [1], we further propose another two techniques, which combine multiple equivalent messages into a single message and compress the messages that are sent via different outgoing edges. Although our

[1] https://neo4j.com/

[2] http://research.microsoft.com/en-us/projects/trinity/

method is devised for answering provenance-aware RPQs over RDF graphs, it can be well adapted to the RPQs under the standard semantics. Actually, the answer of provenance-aware RPQs can be regarded as a subgraph of the RDF graph. In contrast, an RPQ with standard semantics is to find a set of pairs of vertex, and these vertices are completely included in the subgraph of provenance-aware answers. Further, our method can be easily extended to handle RPQs over general (un)directed labeled graphs.

Our main contributions include: (1) we propose an automata-based distributed algorithm, called DP2RPQ, for RPQs under the provenance-aware semantics using the Pregel graph parallel computing framework; (2) several optimization strategies in three aspects are presented to reduce the overhead of the basic DP2RPQ algorithm and alleviate the counting-paths problem; and (3) the extensive experiments were conducted to verify the efficiency and scalability of the proposed method on both synthetic and real-world datasets.

The rest of this paper is organized as follows. Section 2 reviews related work. In Section 3, we introduce preliminary definitions of RPQs. In Section 4, we describe in detail the DP2RPQ algorithm for answering provenance-aware RPQs. We then present the optimization techniques in Section 5. Section 6 shows experiment results, and we conclude in Section 7.

## 2 Related work

Most of the existing approaches aim to evaluate RPQs under the standard semantics, but relatively fewer works focus on RPQs under the provenance-aware semantics. Currently, we are not aware of any distributed Pregel-based approach to evaluating provenance-aware RPQs. We classify the existing approaches into the following two categories.

### 2.1 Standalone RPQs evaluations on a single machine

**Standard semantics of RPQs** The approach proposed in [9] answers RPQs using views, which can be interpreted as checking whether a pair of nodes is one of the answers. The view-based approach for RPQs has been extensively investigated, while the types of data and queries that this approach can handle are restricted under certain assumptions. Koschmieder et al. [15] propose a rare-labels-based approach that decomposes RPQs into a series of smaller RPQs. The rare labels denote the elements in RPQs that have few matches by utilizing the labels and their frequencies in data graph. However, the performance of the method highly depends on a specific query decomposition and selectivity of rare labels.

**Provenance-aware semantics of RPQs** Dey et al. [11] first translate the provenance-aware RPQs into standard Datalog queries or SQL queries, in which auxiliary predicates are introduced to evaluate queries represented by Datalog. In this work, two evaluators for RPQs and provenance-aware RPQs are both implemented on the relational DBMS. However, from the experimental results, we can observe that the approach is hardly scalable for large-scale RDF graphs.

### 2.2 Distributed RPQs evaluations in parallel

**Standard semantics of RPQs** A distributed algorithm for evaluating RPQs on large-scale RDF graphs is proposed in [26], which is the first work to investigate RPQs using partial

evaluation. It employs a dynamic programming method to compute partial answers in parallel, which are then assembled to obtain the final results using an automata-based algorithm. Nevertheless, the experiments on the real-world datasets are not shown in the paper. Sartiani et al. [20] exploit Brzozowski's derivatives [8] of regular expressions to evaluate RPQs in a vertex-centric and message-passing-based manner, which is implemented on top of the Giraph framework. However, the experimental results are only evaluated on the Erdös-Rényi models and the power-law graphs, lacking experiments on synthetic and real-world RDF graphs to verify the algorithm. A system for processing GXPath queries on a large data graphs is proposed in [21], in which GXPath [18] is the most powerful extension of RPQ. In this system, built on top Hadoop MapReduce, a query is complied into an acyclic graph of MapReduce jobs, similar in spirit to a database query plan. However, each graph must be indexed before becoming available for querying.

**Provenance-aware semantics of RPQs** Wang et al. [24] propose an automata-based approach, which employs product automata for evaluating RPQs under the provenance-aware semantics in parallel. The product automaton is constructed using two NFA converted from the regular expression of an RPQ and the RDF data graph, respectively. Then the answer paths are extracted by running the product automaton recursively. Nevertheless, the product automata construction in this method may incur high overhead and excessive communication cost when dealing with large-scale RDF graphs. RDFPath [22] implements an expressive RDF path query language using the MapReduce framework. A query in RDFPath is translated as a sequence of location steps, in which the predicate in query is specified by the next adjacent edge attribute and separated by ">". Since the location steps in RDFPath are deterministic predicates, queries are executed only on triples in graphs associated with these predicates. The execution plan of query is generated by dividing these location steps, which corresponds to a join between an intermediate set of paths and the corresponding RDF graph partition. However, it cannot implement the complete expressiveness of regular path queries, especially the Kleene closure operation. A new query language on the graph is presented in [3, 4], called G-Path, which focuses on complex path pattern query processing on a very large graph. Further, a system called Para-G [5] is introduced to process G-Path queries, which is based on a BSP-like model as well as MapReduce model [10], and can effectively handle distributed graph data operations and queries. Nevertheless, the experimental results are only evaluated on synthetic datasets, lacking experiments on real-world datasets.

Unlike the above previous works, we propose a Pregel-based algorithm for evaluating provenance-aware RPQs on big RDF graphs. To the best of our knowledge, it is the first work to implement an efficient and scalable evaluation of provenance-aware RPQs using the Pregel parallel graph computing model.

## 3 Preliminaries

We start by formally defining background knowledge, this section also serves to establish the notation we will use through the rest of paper.

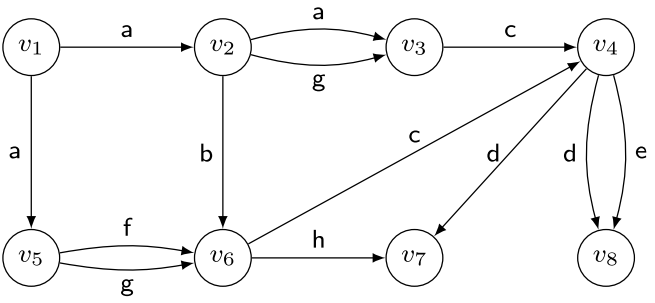
**Definition 1** *RDF graph* Let $U$ and $L$ be the disjoint infinite sets of URIs and literals, respectively. A tuple $(s, p, o) \in U \times U \times (U \cup L)$ is called an *RDF triple*, where $s$ is the *subject*, $p$ is the *predicate (a.k.a. property)*, and $o$ is the *object*. A finite set of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

Given an RDF graph $T = (V, E, \Sigma)$, where $V$, $E$, and $\Sigma$ denote the set of vertices, edges, and edge labels in $T$, respectively. Formally, $V = \{s \mid (s, p, o) \in T\} \cup \{o \mid (s, p, o) \in T\}$, $E = \{(s, o) \mid (s, p, o) \in T\}$, and $\Sigma = \{p \mid (s, p, o) \in T\}$. In addition, we define an infinite set $Var$ of variables that is disjoint from $U$ and $L$. An example *RDF graph* $T_2$ is shown in Figure 2a, which consists of 13 triples (i.e., edges). For instance, $(v_1, \mathtt{a}, v_2)$ is an RDF triple as well as an edge with label a in $T_2$, and $V_{T_2} = \{v_i \mid 1 \leq i \leq 8\}$, $\Sigma_{T_2} = \{\mathtt{a,b,c,d,e,f,g,h}\}$.

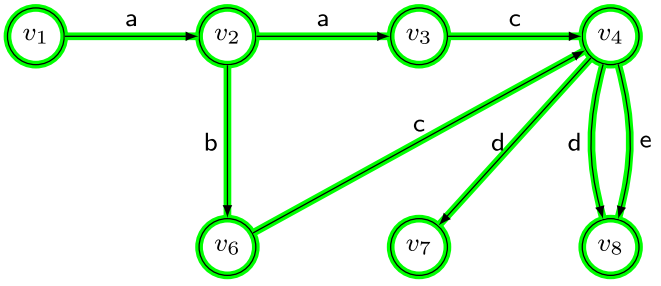
**Definition 2** *Regular path queries* Let $Q = (x, r, y)$ be a regular path query over an RDF graph $T = (V, E, \Sigma)$, where $x, y \in Var$ are variables, and $r$ is a regular expression over the alphabet $\Sigma$. Regular expression $r$ is recursively defined as $r ::= \varepsilon \mid p \mid r/r \mid r|r \mid r^*$, where $p \in \Sigma$ and /, |, and $^*$ are concatenation, alternation, and the Kleene closure, respectively. The shorthands $r^+$ for $r/r^*$ and $r?$ for $\varepsilon|r$ are also allowed. $L(r)$ denotes the language expressed by $r$ and $\lambda(\rho)$ is the label of path $\rho$. The answer set of $Q$ under the standard semantics, denoted by $[\![Q]\!]_T$, is defined as $\{(x, y) \mid \exists$ a path $\rho$ in $T$ from $x$ to $y$ s.t. $\lambda(\rho) \in L(r)\}$.

Given a regular expression $r$, let $Pos(r) = \{1, 2, \ldots, |r|\}$ be the set of positions in $r$, where $|r|$ is the length of $r$. Thus, the symbols in $r$ can be denoted as $r[1], r[2], \ldots, r[|r|]$.

**Definition 3** *Automata of RPQs* Given an RDF graph $T$ and an RPQ $Q = (x, r, y)$ over $T$, the *automaton of RPQ* $Q$ is the Glushkov automaton $A_Q$ converted from the regular expression $r$ by using the Glushkov's construction algorithm [7]. The function $\mathtt{first}(r)$




(a) An RDF graph $T_2$




(b) Provenance-aware answer of $Q_2 = (x, \mathtt{a/b^*/c/(d|e)}, y)$ on $T_2$

**Figure 2** An RDF graph $T_2$ and provenance-aware answer of $Q_2$ on $T_2$

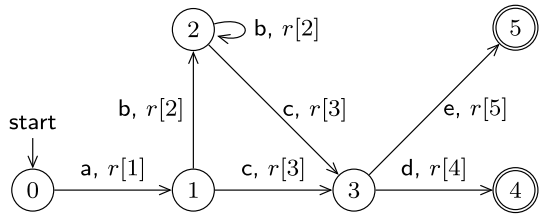
(resp. $\texttt{last}(r)$) is the set of positions in $r$ that can match the first (resp. last) symbol of some string in $L(r)$, and the function $\texttt{follow}(r, i)$ is the set of positions in $r$ that can follow position $i$ when matching some string in $L(r)$. $A_Q$ is defined as a 5-tuple $(St, \Sigma, \delta, q_0, F)$, where (1) $St = \{0\} \cup Pos(r)$ is a finite set of states, (2) $\Sigma$ is the alphabet of $r$, (3) $\delta : St \times \Sigma \to \mathcal{P}(St)$ is the transition function, (4) $q_0 = 0$ is the initial state, (5) and $F$ is the set of final states. Here, $\delta$ and $F$ are further defined as follows:

$$\delta(q, a) = \begin{cases} \{i \mid i \in \texttt{first}(r) \wedge r[i] = a\} & \text{if } q = q_0 \\ \{i \mid i \in \texttt{follow}(r, q) \wedge r[i] = a\} & \text{if } q \in Pos(r) \end{cases}$$

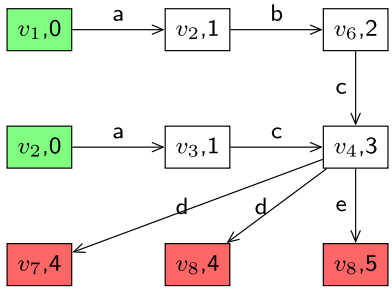
$$F = \begin{cases} \{q_0\} \cup \texttt{last}(r) & \text{if } \varepsilon \in L(r) \\ \texttt{last}(r) & \text{otherwise} \end{cases}$$

*Example 1* Given an RPQ $Q_2 = (x, r, y)$ and $r = \mathsf{a/b^*/c/(d|e)}$, we build $A_{Q_2} = \{St, \Sigma, \delta, q_0, F\}$ based on $r$, where $St = \{0, 1, 2, 3, 4, 5\}$, $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}\}$, $q_0 = \{0\}$, $F = \{4, 5\}$, and the transition function $\delta$ is represented in the form of the transition graph shown in Figure 3a.

**Definition 4** *Provenance-aware answer set of RPQs* Given an RDF graph $T = (V, E, \Sigma)$ and an automaton $A_Q = (St, \Sigma, \delta, q_0, F)$ of an RPQ $Q$, the *provenance-aware answer set* of $Q$ over $T$, denoted by $[\![Q]\!]^{\mathfrak{p}}_T$, is defined as a 5-tuple $(V_{\mathfrak{p}}, E_{\mathfrak{p}}, L_{\mathfrak{p}}, I_{\mathfrak{p}}, F_{\mathfrak{p}})$, where (1) $V_{\mathfrak{p}} \subseteq V \times St$ is a set of vertices, (2) $E_{\mathfrak{p}} \subseteq V_{\mathfrak{p}} \times V_{\mathfrak{p}}$ is a set of edges, (3) $L_{\mathfrak{p}}$ is a function that assigns each edge a label in $\Sigma$, and (4) $I_{\mathfrak{p}} = \{(v, q_0) \mid v \in V\} \subseteq V_{\mathfrak{p}}$ and $F_{\mathfrak{p}} = \{(v, q_f) \mid v \in V \wedge q_f \in F\} \subseteq V_{\mathfrak{p}}$ are the sets of start and final vertices, respectively. Here, $[\![Q]\!]^{\mathfrak{p}}_T$ is constructed by the following process: for each path $v_0 a_0 v_1 \cdots v_{n-1} a_{n-1} v_n$ in $T$ such that




(a) The automaton $A_Q$ of $Q_2 = (x, \mathsf{a/b^*/c/(d|e)}, y)$




(b) The provenance-aware answer of $Q_2 = (x, \mathsf{a/b^*/c/(d|e)}, y)$ over $T$

**Figure 3** The automaton and the provenance-aware answer of $Q_2$

**Table 1** The functions in Pregel framework

| Functions | Description |
|---|---|
| `superStep` | to get the number of the current superstep |
| `getState` | to get the current state of a vertex |
| `voteToHalt` | to deactivate a vertex $v$, i.e., $\texttt{getState}(v) = inactive$ |
| `sendMsg` | to send messages $M$ from one vertex $v$ to another vertex $v'$ |

there exists a sequence of states $st_0, st_1, \ldots, st_n$ in $St$ satisfying $st_0 = q_0$, $st_{i+1} \in \delta(st_i, a_i)$ for $i \in \{0, \ldots, n-1\}$, and $st_n \in F$, a path $\rho = v_{\mathfrak{p}_0} a_0 v_{\mathfrak{p}_1} \cdots v_{\mathfrak{p}_{n-1}} a_{n-1} v_{\mathfrak{p}_n}$ in $[\![Q]\!]^{\mathfrak{p}}_T$ is constructed, where $v_{\mathfrak{p}_i} = (v_i, st_i)$ for $i \in \{0, \ldots, n\}$ and by definition $v_{\mathfrak{p}_0} \in I_{\mathfrak{p}} \wedge v_{\mathfrak{p}_n} \in F_{\mathfrak{p}}$.

*Example 2* The *provenance-aware answer set* of $Q_2$ over $T_2$ as defined in Definition 4 is shown in Figure 3b, which can be considered as the extended version of the *provenance-aware answer* to $Q_2$ (as a *subgraph* of $T_2$ marked in green in Figure 2b), which attaches the matched states of $A_{Q_2}$ to the corresponding vertices.

Pregel is a vertex-centric parallel model for graph computation . The computation in Pregel is composed of a sequence of iterations, i.e., *supersteps*, conforming to the *Bulk Synchronous Parallel* (*BSP*) model [12].

There are several functions involving in the process of Pregel-based algorithms, as shown in Table 1. In particular, for each vertex $v$, we define $Val(v)$ as the set of values associated with $v$. Within a superstep, each vertex executes the user-defined vertex computation $\texttt{vertexCompute}(T, M)$ to get and update $Val(v)$ *in parallel*.

The description of notation introduced in the definition are shown in Table 2, which we will use in the rest of paper.

**Definition 5** *Pregel framework* Given an RDF graph $T = (V, E, \Sigma)$ as the input data, in the first superstep, all the vertices are active. The entire computation terminates when all vertices are inactive. Let $M$ be the set of messages. Within a superstep, the user-defined function $\texttt{vertexCompute}(T, M)$ is executed on each active vertex in parallel. An inactive vertex will be reactivated by the incoming messages sent to it. When $\texttt{vertexCompute}(T, M)$ is invoked on each active vertex $v$, it (1) gets the number of current superstep by `superStep`; (2) receives messages (i.e., each $m \in M$) sent to $v$ in the previous superstep; (3) obtains and/or updates $Val(v)$; (4) modifies $M$ to generate the set of new messages $M'$; (5) invokes $\texttt{sendMsg}(v, v', M')$ to send $M'$ to the adjacent vertex $v'$, and (6) invokes `voteToHalt`.

**Definition 6** *Matching pair* Given an RDF graph $T = (V, E)$ and an automaton $A_Q$ of an RPQ $Q = (x, r, y)$, the query processing in Pregel is matched forward by generating the *matching pair* in each superstep. The matching pair is denoted as a pair $(v, q)$ satisfying $\exists\, a \in \mathcal{S}(v) \wedge q \in \{St \setminus q_0\} \wedge v \in V \wedge a = r[q]$, where $\mathcal{S}(v)$ is the set of symbols labeled on the incoming edges of $v$.

In particular, $(v, q_0)$ $(q_0 \in A_Q)$ is also called a matching pair. The macthing pairs generated in query processing can be demonstrated as the vertices $V_{\mathfrak{p}} \subseteq V \times St$ in the

**Table 2** The notations in the definition

| Notations | Description |
|---|---|
| $U$ | the infinite sets of URIs |
| $L$ | the infinite sets of literals |
| $(s, p, o)$ | an *RDF triple* |
| $s$ | the *subject* of an *RDF triple*, where $s \in U$ |
| $p$ | the *predicate (a.k.a. property)* of an *RDF triple*, where $p \in U$ |
| $o$ | the *object* of an *RDF triple*, where $o \in U \cup L$ |
| $T$ | an RDF graph, where $T = (V, E, \Sigma)$ |
| $V$ | the set of vertices in $T$, where $V = \{s \mid (s, p, o) \in T\} \cup \{o \mid (s, p, o) \in T\}$ |
| $E$ | the set of edges in $T$, where $E = \{(s, o) \mid (s, p, o) \in T\}$ |
| $\Sigma$ | the set of edge labels in $T$, where $\Sigma = \{p \mid (s, p, o) \in T\}$ |
| $Var$ | an infinite set of variables that is disjoint from $U$ and $L$ |
| $Q = (x, r, y)$ | a regular path query, where $x, y \in Var$ |
| $r$ | a regular expression, where $r ::= \varepsilon \mid p \mid r/r \mid r\|r \mid r^*$ |
| $L(r)$ | the language expressed by $r$ |
| $\lambda(\rho)$ | the label of path $\rho$ |
| $\|r\|$ | the length of $r$ |
| $Pos(r)$ | the set of positions in $r$, where $Pos(r) = \{1, 2, \ldots, \|r\|\}$ |
| $A_Q$ | an automaton of an RPQ $Q$, where $A_Q = (St, \Sigma, \delta, q_0, F)$ |
| $[\![Q]\!]_T$ | the answer set of $Q$ over $T$ under the standard semantics |
| $[\![Q]\!]_T^{\mathfrak{p}}$ | the *provenance-aware answer set* of $Q$ over $T$ |

provenance-aware answer set $[\![Q]\!]_T^{\mathfrak{p}} = (V_{\mathfrak{p}}, E_{\mathfrak{p}}, L_{\mathfrak{p}}, I_{\mathfrak{p}}, F_{\mathfrak{p}})$ in Definition 4. In general, an answer path in $[\![Q]\!]_T^{\mathfrak{p}}$ can be denoted as a sequence of matching pairs.

## 4 The Pregel-based algorithm

In this section, we propose the Pregel-based algorithm for answering provenance-aware RPQs, which employs the automata introduced in Section 3. First, we describe the overall evaluation, then elaborate the computation in each vertex of each superstep. Finally, we discuss the cycle detection mechanism to avoid loop or infinite matching when evaluating on cyclic RDF graphs.

### 4.1 Architecture overview

The architecture of the Pregel-based algorithm for evaluating provenance-aware RPQs is shown in Figure 4. ① Given an RDF graph $T$ and an RPQ $Q = (x, r, y)$ over $T$, the *automaton of RPQ* $Q$ is the Glushkov automaton $A_Q$ converted from the regular expression $r$ by using the Glushkov's construction algorithm; ② The query processor, which starts with the vertices in an RDF graph to match against the states in the corresponding automaton of $Q$, operates using the Pregel parallel computing framework to compute the matched intermediate results; ③ Several optimization strategies are presented to reduce the overhead

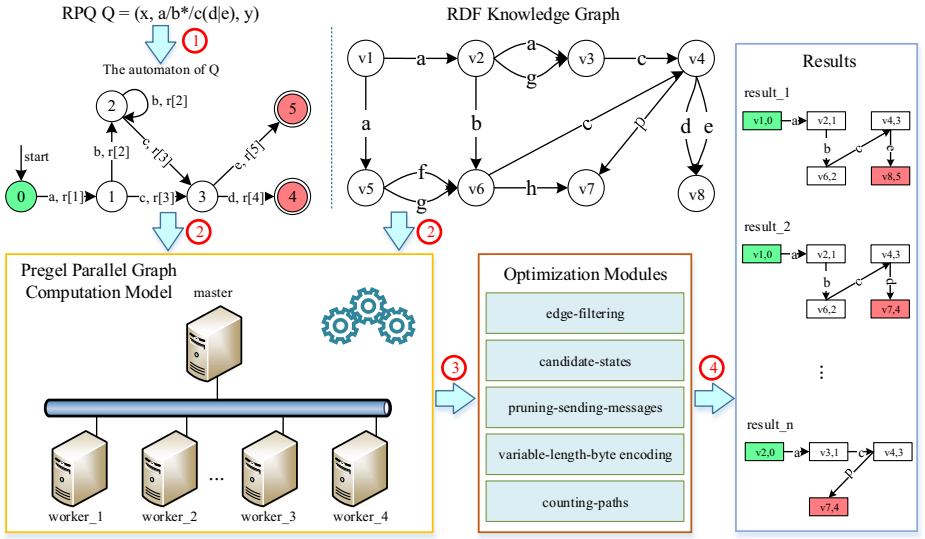



**Figure 4** The architecture of DP2RPQ

of the basic DP2RPQ algorithm and alleviate the counting-paths problem; ④ After the entire computation completes, we can obtain the final *provenance-aware* results.

### 4.2 Provenance-aware RPQs based on Pregel

The overall evaluation DP2RPQ is shown in Algorithm 1, in which we construct an automaton $A_Q = \{St, \Sigma, \delta, q_0, F\}$ of an RPQ $Q = (x, r, y)$ (line 1). In each superstep, each active vertex $v$ invokes `vertexCompute` *in parallel* (line 4) to match against a state $q \in St$, while the updated partial answers are maintained in $Val(v)$. The matching process is executed repeatedly in the following supersteps until the computation terminates. When the entire computation completes, we combine $Val(v)$ of each vertex $v$ to obtain the final *provenance-aware answer set* $[\![Q]\!]_T^{\mathfrak{p}}$ (line 5).

**Algorithm 1** DP2RPQ.

**Input** : An RDF graph $T = (V, E, \Sigma)$ and an RPQ $Q = (x, r, y)$
**Output**: The provenance-aware answer set $[\![Q]\!]_T^{\mathfrak{p}}$

1 Build the automaton $A_Q = \{St, \Sigma, \delta, q_0, F\}$ of $Q$;
2 Compute $r[i]$, `first`$(r)$, `last`$(r)$, and `follow`$(r, i)$ based on $r$ of $Q$;
3 $M_r \leftarrow \emptyset$;
4 `vertexCompute`$(T, M_r)$; `/* at each vertex in parallel */`
5 $[\![Q]\!]_T^{\mathfrak{p}} \leftarrow \bigcup_{v \in V} Val(v)$;
6 **return** $[\![Q]\!]_T^{\mathfrak{p}}$;

The process of the vertex computation `vertexCompute` is shown in Algorithm 2. It is executed at each vertex $v \in V$ *in parallel*, which has the following three phases:

1) In the first superstep (lines 2-7), $v$ is considered to be matched against the initial state $q_0$ only if there exists an outgoing edge $(v, v')$ such that the label of $(v, v')$ is the same

as $r[q]$ satisfying $q \in \texttt{first}(r)$ (lines 3-4). Then, the first matched message $m$ is generated, which formally is a set of matching pair $(v, q_0)$. Further, the message set $M_s$ is a set of matched messages, which is sent to the adjacent vertices by invoking `sendMsg` (line 7). Finally, $\texttt{getState}(v) = \textit{inactive}$ by invoking `voteToHalt` (line 21).

2) As to the remaining supersteps (lines 8-20), if the set $M_r$ of the receiving messages from the adjacent vertices in the previous superstep is empty, $v$ is deactivated by `voteToHalt` (line 21), otherwise each active vertex is matched forward based on the messages in $M_r$ (lines 10-19). First, the set $R_{q'}$ of the next possible states is computed by `follow` w.r.t. the current matched state $q$ (line 12). Next, if $v$ has an outgoing edge labeled with the same symbol as $r[q']$ ($q' \in R_{q'}$), a new message $m$ is built by appending $(v, q')$ to $m'$ and then added to the message set $M_s$ to be sent (lines 13-16). Finally, $\texttt{sendMsg}(v, v', M_s)$ is invoked to send $M_s$ from $v$ to $v'$ (line 20). Meanwhile, $v$ checks whether the current matched state $q'$ of the new set of matching pairs $m$ is a final state (i.e., $q' \in \texttt{last}(r)$)(line 17). If it is, then $m$ is regard as the answer path $\rho_f$ in form of a sequence of matching pairs, which is added to the partial provenance-aware answer set, denoted by $Val(v)$ (lines 18-19).

**Algorithm 2** `vertexCompute(`$T$`,` $M_r$`)`.

```
   Input : An RDF graph T and a set M_r of receiving messages
1  V_n ← all the adjacent vertices of v;
2  if superStep = 1 then
3      foreach outgoing edge (v, v') of v do
4          if ∃ q ∈ first(r) ∧ r[q] = λ((v, v')) then
5              m ← ((v, q_0));
6              M_s ← M_s ∪ {m};
7      foreach v' ∈ V_n do sendMsg(v, v', M_s) ;
8  else
9      if M_r ≠ ∅ then
10         foreach m' ∈ M_r do
11             (v_t, q) ← the last element of m';
12             R_q' ← follow(r, q);
13             foreach outgoing edge (v, v') of v do
14                 if ∃ q' ∈ R_q' ∧ r[q'] = λ((v, v')) then
15                     m ← append (v, q') to m';
16                     M_s ← M_s ∪ {m};
17                     if q' ∈ last(r) then
18                         ρ_f is the equivalent path of m' ;      /* λ(ρ_f) ∈ L(r) */
19                         Val(v) ← Val(v) ∪ {ρ_f} ;      /* answer paths */
20     foreach v' ∈ V_n do sendMsg(v, v', M_s) ;
21 voteToHalt
```

In $\texttt{sendMsg}(v, v', M_s)$ (line7, 20), the condition of sending a message $m \in M_s$ from $v$ to $v'$ is that the current matched state $q$ satisfies $\exists\, q' \in \texttt{follow}(r, q) \wedge r[q'] = \lambda((v, v'))$. In addition, when $v'$ receives messages from different adjacent vertices, it merges all the messages into $M_r$.

The correctness of the DP2RPQ algorithm is guaranteed by the following theorem.

**Theorem 1** *Given an RPQ $Q = (x, r, y)$ over an RDF graph $T$, $(v_0, v_n) \in [\![Q]\!]_T$ iff $\exists$ $\{(v_0, q_0), (v_1, q_1), \ldots, (v_n, q_n)\} \in [\![Q]\!]_T^{\mathfrak{p}}$ in DP2RPQ.*

*Proof* (Sketch)

(i) **"If"** direction: for $\{(v_0, q_0), (v_1, q_1), \ldots, (v_n, q_n)\} \in [\![Q]\!]_T^{\mathfrak{p}}$, $\exists$ a path $\rho_1$ in $T$ from $v_0$ to $v_n$ and a path $\rho_2$ in $A_Q$ from $q_0$ to $q_n$. It can be observed that $q_i \in \delta(q_{i-1}, \lambda((v_{i-1}, v_i)))$, for $1 \in \{1, \ldots, n\}$, holds in DP2RPQ. The label of $\rho_1$ is the same as that of $\rho_2$, i.e., $\lambda(\rho_1) \in L(r)$. Therefore, $(v_0, v_n) \in [\![Q]\!]_T$.

(ii) **"Only if"** direction: for $(v_0, v_n) \in [\![Q]\!]_T$, assume a path in form of $v_0 a_0 \cdots v_{n-1} a_{n-1} v_n$ in $T$ such that there exists a sequence of states $st_0 \ldots st_n$ in $St$ of $A_Q$ in DP2RPQ satisfying $st_0 = q_0$, $st_{i+1} \in \delta(st_i, a_i)$ for $i \in \{0, \ldots, n-1\}$, and $st_n \in F$. The path $\rho = (v_0, q_0)a_0(v_1, q_1) \cdots (v_{n-1}, q_{n-1})a_{n-1}(v_n, q_n)$ is constructed in DP2RPQ, i.e., $\{(v_0, q_0), (v_1, q_1), \ldots, (v_n, q_n)\} \in [\![Q]\!]_T^{\mathfrak{p}}$. □

**Theorem 2** *The complexity of the DP2RPQ algorithm is bounded by $O(|deg_m^+|^k \cdot |r|^k \cdot |deg_m^-|^{k-1})$, where $|r|$ is the length of the regular expression $r$ in $Q$, $k$ is the total number of supersteps, and $|deg_m^+|$ (resp. $|deg_m^-|$) is the maximum outdegree (resp. indegree) of the vertices in $T$.*

*Proof* (Sketch)

(i) **Basis**: When $k = 1$, in the first superstep, there exists a vertex that is matched for $O(|deg_m^+| \cdot |r|)$ times since at most $|deg_m^+|$ outgoing edges of the vertex are matched against the states in $\texttt{first}(r)$. Thus, the complexity is $O(|deg_m^+| \cdot |r|)$.

(ii) **Induction step**: For $k$ ($k \geq 1$) supersteps, the complexity is $O(|deg_m^+|^k \cdot |r|^k \cdot |deg_m^-|^{k-1})$ as the induction hypothesis. Thus, there exists a vertex that is matched $O(|deg_m^+|^k \cdot |r|^k \cdot |deg_m^-|^{k-1})$ times, and all these matches are sent as messages via the outgoing edges. Then, for $(k+1)$ supersteps, since the maximum number of incoming edges of a vertex may be $|deg_m^-|$, the receiving message set of the vertex includes $O(|deg_m^+|^k \cdot |r|^k \cdot |deg_m^-|^{k+1})$ messages. Next, each receiving message of the vertex are matched against at most $|r|$ states in the automaton based on the label of $|deg_m^+|$ outgoing edges of the vertex to generate $O(|deg_m^+|^{k+1} \cdot |r|^{k+1} \cdot |deg_m^-|^k)$ messages. Thus, the vertex is matched $O(|deg_m^+|^{k+1} \cdot |r|^{k+1} \cdot |deg_m^-|^k)$ times, which is the complexity for $(k+1)$ supersteps. □

In particular, $|deg_m^+|$ and $|deg_m^-|$ can be further reduced to $|r|$ by our optimization techniques in Section 5. Thus, the complexity of the optimized DP2RPQ algorithm is $O(|r|^{3k-1})$. In general, $|r|$ is short in length and the number of total supersteps $k$ is also limited.

### 4.3 Cycle detection

Loop matching may be generated in the matching process of RPQs when evaluating on cyclic RDF graphs. A cycle in an RDF graph is a path of edges and vertices wherein a vertex is reachable from itself.

**Definition 7** (*Loop matching*) Given a provenance-aware answer set $[\![Q]\!]^{\mathfrak{p}}_T = (V_{\mathfrak{p}}, E_{\mathfrak{p}}, L_{\mathfrak{p}}, I_{\mathfrak{p}}, F_{\mathfrak{p}})$ of an RPQ $Q$ over an RDF graph $T$, an answer path $\rho_f = \{(v_s, q_0), \ldots, (v_i, q_m), \ldots, (v_j, q_n), \ldots\}$ in $[\![Q]\!]^{\mathfrak{p}}_T$ is called a *loop matching* if and only if $j = i$.

In Definition 7, only if a vertex is matched more than once in a path, the matching can be regarded as a loop matching. Considering the edges and states additionally, loop matching can be further refined into four cases.

(1) **only vertex**: $(v_i, q_m)$ and $(v_i, q_k)$ $(m \neq k)$ occur in the same path
(2) **vertex and state**: $(v_i, q_m)$ and $(v_i, q_k)$ $(m = k)$ occur in the same path
(3) **vertex and edge**: $(v_i, q_m)(v_j, q_n)$ and $(v_i, q_k)(v_j, q_l)$ $(m \neq k \wedge n \neq l \wedge r[q_n] = r[q_l])$ occur in the same path
(4) **vertex, edge and state**: $(v_i, q_m)(v_j, q_n)$ and $(v_i, q_k)(v_j, q_l)$ $(m = k \wedge n = l \wedge r[q_n] = r[q_l])$ occur in the same path

In particular, when closure operations * and/or $^+$ occur in $r$, it may lead to infinite number of matches when evaluating on cyclic RDF graphs. Therefore, we introduce a cycle detection mechanism in DP2RPQ to ensure that a vertex is not matched with the same state twice in intermediate partial answers, i.e., considering vertex and state. For a message $m'$ in a set of receiving messages, if a 2-tuple matching pair $(v, q) \in m'$, then $(v, q)$ cannot be added to $m'$ again for generating a new message in the following supersteps.

## 5 Optimization strategies

To improve the efficiency of our method, we evaluate the cost of the Pregel computation. Further, three optimization strategies are devised according to the cost model, which can reduce the cost of vertex-computation, reduce the intermediate results of the basic DP2RPQ algorithm dramatically, and address the counting-paths problem to some extent, respectively.

### 5.1 Cost estimation

The cost of the Pregel computation is determined as the sum of the cost of all supersteps. The cost of each superstep consists of the following terms: (1) $w_i$ is the maximum cost of vertex computation among all vertices in the $i$-th superstep; (2) $h_i$ is the maximum number of messages sent or received by each vertex; and (3) $l$ is the cost of the barrier synchronization at the end of a superstep. Thus, the cost of a Pregel-based algorithm is $\sum_{i=1}^{k} w_i + g \sum_{i=1}^{k} h_i + kl$, where $g$ is the cost to deliver a message and $k$ is the number of total supersteps. Therefore, the cost of DP2RPQ is determined by the cost of vertex computation, the cost of passing messages, and the number of the supersteps.

### 5.2 Vertex-computation optimization

In this section, we design two techniques to reduce the cost of vertex-computation in each superstep.

#### 5.2.1 Edge filtering

Generally, the input graph is stored into main memory in Pregel, which may result in an inefficient memory utilization. In order to improve the efficiency and scalability of the DP2RPQ

algorithm, we design the *edge-filtering* technique, which only loads the edges labeled with the symbols that occur in $r$ of $Q = (x, r, y)$ by filtering other edges. We use $\Sigma_r$ to denote the subset of the alphabet that appears in $r$. Then, an edge $(v, \mathtt{a}, u)$ in the input RDF graph $T$ is loaded if and only if $\mathtt{a} \in \Sigma_r$. In Example 2, with edge filtering, only the edges labeled with the symbols in $\Sigma_r = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$ are involved in the processing, while the edges labeled with `f`, `g`, and `h` are filtered.

### 5.2.2 Candidate states

In Algorithm 2, the traversal operations over the incoming edges of each vertex (line 13) are not efficient when evaluating RPQs on large-scale RDF graphs. Thus, we leverage the *prior knowledge* in a given query RPQ $Q$ over an RDF graph $T$ to construct an auxiliary structure called *candidate states*, denoted by $R_c$, which keeps a set of the state $q$ of the RPQ automaton in each vertex $v$ satisfying a matching pair $(v, q)$.

$R_c$ is precomputed before processing and calculated only once. The construction of the candidate-states consists of two situations. First, we construct a set including $\{(v_1, q_0), \ldots, (v_i, q_0), \ldots, (v_n, q_0)\}$ $(n \leq |V|)$ and check whether the outgoing edges of each vertex $v$ labeled with the same symbols as $r[q]$ $(q \in \mathtt{first}(r))$. If it satisfies, $q_0$ is put into $R_c$ of $v$. Then, at each vertex $v$, we check whether the incoming edges of $v$ labeled with the same symbols as $r[q]$ $(q \in St)$. If it satisfies, $q$ is put into $R_c$ of $v$. With candidate-states technique, we traverse the edges of each vertex only once, which avoids the excessive cost of traversal operations in each superstep of the Pregel-based processing.

**Algorithm 3** `candidateStates()`.

```
1 R_c ←the candidate states of v;
2 (v_t, q) ←the last matching pair in a received message m';
3 R_{q'} ← follow(r, q);
4 if R_c ∩ R_{q'} ≠ ∅ then
5     foreach q' ∈ R_c ∩ R_{q'} do
6         m ← append (v, q') to m';
7         M_s ← M_s ∪ {m};
8 else no new message m to be generated;
```

In vertex computation, we compare the states in $R_c$ with that in $R_{q'}$ instead of the costly iteration of all adjacent vertices. Algorithm 3 is an optimized version of lines 13-16 in Algorithm 2 by using the candidate-states technique, in which the modified matching process is: (1) $R_{q'}$ is computed by function `follow`; (2) $v$ receives the message $m' \in M_r$, if $q' \in R_c \cap R_{q'}$, a new message $m$ will be built by appending $(v, q')$ to $m'$, otherwise there is no new message to be generated for this particular message $m'$.

*Example 3* Consider the RDF graph $T_2 = (V, E, \Sigma)$, the RPQ $Q_2 = (x, r, y)$, and the automaton $A_{Q_2} = \{St, \Sigma, \delta, q_0, F\}$ in Example 2. First, $q_0$ is appended to $R_{c_{v_1}}$ and $R_{c_{v_2}}$ since $v_1$ and $v_2$ have outgoing edges labeled with `a` satisfying $\mathtt{first}(r) = \{1\}$ and $r[1] = \mathtt{a}$. Then, for example, there exists an incoming edge of $v_2$ labeled with `a` and $r[1] = \mathtt{a}$, the candidate states for $v_2$ are $R_{c_{v_2}} = \{0, 1\}$, as shown in Figure 5. If $v_2$ receives a message $m' = ((v_1, 0))$ from $v_2$, $R_{q'_0} = \mathtt{follow}(r, 0) = \{1\}$ is computed. Thus, $(v_2, 1)$ is appended to $m'$ to generate a new matched message.

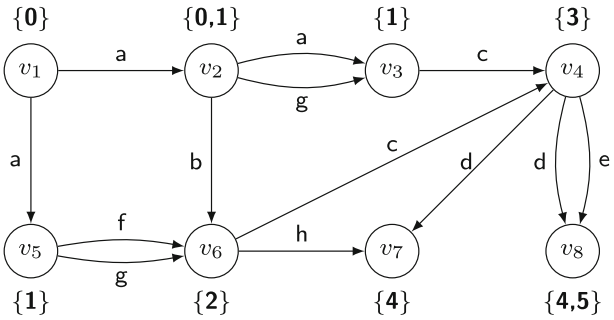



**Figure 5** The RDF graph $T_2$ with candidate-states technique

### 5.3 Message-communication optimization

With the number of the superstep increasing, the size of each message to be sent will become larger. Meanwhile, from the analysis of algorithm complexity, it can be seen that the number of messages to be sent in the $k$-th superstep can reach exponential complexity of the maximum degree of outdegree/indegree in an RDF graph. When dealing with large-scale RDF knowledge graphs, the communication cost of sending a large number of messages is time-consuming, in this section, we consider reducing the communication cost. On one hand, we prune unnecessary messages by using the provenance-aware RPQs' features and the Kleene closure operations to reduce the message-passing cost and the number of supersteps. On the other hand, we encode the messages that have to be sent by using variable-length-byte encoding to reduce the sizes of messages.

#### 5.3.1 Pruning sending messages

In order to realize pruning some matching pairs in the messages to be sent, we subtract the matching pairs that has already been added to the partial provenance-aware answer set. Further, we subtract the duplicate matching pairs which are incurred by the Kleene closure operations.

**Pruning answer matches** In the matching process of the `DP2RPQ` algorithm, when the current matched state of a message that received at the vertex is an accept state, the message is converted into an equivalent answer path, meanwhile, it is matched forward by appending a matching pair. Obviously, as the number of the superstep increases, the length of this message will increase accordingly. To this end, we prune the matching pairs that have already been added into the partial provenance-aware answer set to reduce the length of the message.

For example, given an RPQ $Q_3 = (x, r_3, y)$ and $r_3 = \mathsf{a/b/c}^*$, it is evaluated on the RDF graph $T_3$ shown in Figure 6. In the third superstep, $v_3$ receives the message $m' = (v_1, 0)(v_2, 1)$ from $v_2$, then $v_3$ append $(v_3, 2)$ to $m'$ and maintain the new message $m = (v_1, 0)(v_2, 1)(v_3, 2)$ as the answer path into the partial answer set $Val(v_3)$. The message $m$ is also sent to $v_4$ to be matched forward, among which $(v_1, 0)(v_2, 1)$ is already kept in $Val(v_3)$. Thus, the matching pairs $(v_1, 0)$ and $(v_2, 1)$ in $m$ are pruned, only $m = (v_3, 2)$ is sent to $v_4$ as the message. It has no effect on the final result and can reduce the length of messages sent.

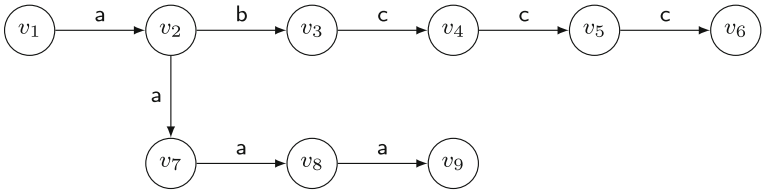



**Figure 6** An RDF graph $T_3$

**Pruning duplicate matches** When evaluating RPQs, the number of the supersteps required to process different queries may vary considerably. For example, in DP2RPQ, given $Q_4 = (x, a/b/c/d, y)$, $|r| = 4$, it needs $(|r| + 1)$ supersteps at most; given $Q_5 = (x, (a/a)^+, y)$, $|r| = 2$, it needs $(|r| * k + 1)$ $(k \in \{1, ..., n\})$ supersteps, and $k$ is proportional to the size of an RDF graph.

When evaluating $Q_5$ on $T_3$, it needs five supersteps. The longest answer path of $Q_5$ is of the form $(v_1, 0)(v_2, 1)(v_7, 2)(v_8, 1)(v_9, 2)$, which is generated at $v_9$ in the fifth superstep. For the matching process, at the third superstep, $v_7$ generates a new message $(v_1, 0)(v_2, 1)(v_7, 2)$, which is to be sent and maintained in the partial provenance-aware set of $v_7$, i.e., $Val(v_7)$. Meanwhile, $v_9$ generates a new message $(v_7, 0)(v_8, 1)(v_9, 2)$ and maintains it in $Val(v_9)$ at the third superstep. Next, in the fourth superstep, $v_8$ receives the message $(v_1, 0)(v_2, 1)(v_7, 2)$ from $v_7$ and appends $(v_8, 1)$ to it to generate a new message. The matching pair $(v_8, 1)$ is a duplicate match, which cannot be appended to the receiving message. Thus, there is no new message in $v_8$ in the fourth superstep. With pruning duplicate matches, the number of total supersteps decreases into $|r| + 1$, which is only related to $|r|$. Generally, $|r|$ can be considered as a constant, which reduces the number of total supersteps and the number of intermediate results.

In summary, the size of messages to be sent can be reduced by pruning matches. The maximum number of sending messages in the $k$-th superstep decreases from exponential complexity to polynomial complexity, i.e., $O(|deg_m^+| \cdot |r| \cdot |deg_m^-|)$.

### 5.3.2 Variable-length-byte encoding

In fact, the cost of passing messages has a significant impact on query performance. Therefore, we need to compress the storage space of messages based on encoding messages.

The message includes a series of matching pairs $(v, q)$, where (1) $v$ can correspond to the vertex label or vertex ID, both of which are unique identifiers; (2) $q$ is the matched state of the automation of an RPQ. In general, $v$ and $q$ can be represented by an integer variable using 4 bytes. If using integer variables to represent $v$ and $q$, when the value is small, space utilization will be very low. It can be seen that fixed length byte encoding that represents $v$ and $q$ wastes a lot of space. Therefore, we use variable-length-byte encoding method, as shown in Algorithm 4. The storage space $\texttt{varbyteEncoding}(x)$ occupied by the integer variable $x$ is as follows:

$$\texttt{varbyteEncoding}(x) = \begin{cases} 1B & \text{if } x < 2^7 \\ 2B & \text{if } 2^7 \leq x < 2^{14} \\ 3B & \text{if } 2^{14} \leq x < 2^{21} \end{cases}$$

**Algorithm 4** `varbyteEncoding(x)`.

**Input** : An integer variable $x$
**Output**: Variable-length-byte encoding of $x$

```
1 B_f ← byte buffers;
2 while (x & ~0x7f) ≠ 0 do
3 |   B_f ← append byte (x & 0x7f | 0x80) to the end of B_f;
4 |   x ← x ≫ 7;
5 B_f ← append byte x to the end of B_f;
6 return B_f;
```

### 5.4 Counting-paths alleviation strategy

In particular, due to the provenance-aware semantics, the counting-paths problem in `DP2RPQ` is often caused. To address it, we attempt to decrease the times of matching and compress the messages to be sent.

#### 5.4.1 Counting-paths problem

It is inevitable to cause the counting-paths problem for a distributed Pregel-based algorithm to generate provenance-aware answers to RPQs, which may incur the prohibitively expensive overhead [1].

**Theorem 3** *In distributed provenance-aware RPQs evaluation, counting-paths problem is inevitable in most cases.*

*Proof* (Sketch) For example, given an RDF graph $T_4$ in Figure 7, the vertex $x$ has $k$ incoming edges labeled with `a` and $n$ outgoing edges labeled with `b`. If $x$ receives a set $M_r$ of $k$ messages via the incoming edges $(v_1, x), (v_2, x), \ldots, (v_k, x)$, there exists a set including $\{m'_j, \ldots, m'_p\} \in M_r$ $(j \geq 1 \wedge p \leq k)$ such that the last element $(v_i, q_i)$ $(j \leq i \leq q)$ in these messages satisfying $R_{q'}$=`follow`$(q_i) \wedge r[q'] =$ `a` $\wedge\, q' \in R_{q'}$. Then, for each message $m' \in \{m'_j, \ldots, m'_p\}$, $|r| \times k$ new messages may be built by appending $(x, q')$ to $m'$ and sent to the adjacent vertices by the $n$ outgoing edges labeled `b` when $R_{q''} =$ `follow`$(q') \wedge r[q''] =$ `b` $\wedge\, q'' \in R_{q''}$. Obviously, $|r| \times k \times n$ messages need to be sent in total, which is actually the Cartesian product of the $k$ receiving messages and $n$ outgoing edges. It is known that the Cartesian product is the key factor in causing the counting-paths problem. □

#### 5.4.2 Message selection

To partly address the counting-paths problem, we reduce the number of matches between a vertex and a state, which is the dominant cost in vertex computation of each superstep. Let

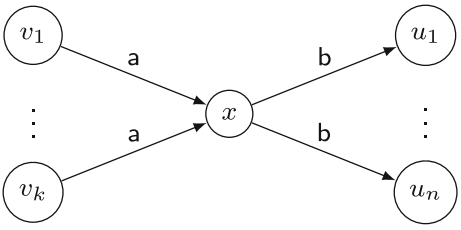



**Figure 7** RDF graph $T_4$ with the Cartesian product

a message set $M = \{m'_1, \ldots, m'_k\}$ be a subset of $M_r$. If the next matched states of the last elements of all messages in $M$ are the same, we just select any message $m'$ from $M$ and append the element $(v, q')$ to $m'$ to built the message, and the remaining message set $M \setminus m'$ is cached in $v$. If there exists an answer path that contains $m'$, then $M \setminus m'$ is appended to the final provenance-aware answer set. This optimization technique is referred to as *message selection*, which can avoid the Cartesian product by reducing the number of messages from $O(k \cdot |r| \cdot n)$ to $O(|r| \cdot n)$.

In Figure 8, there are $|r| \times k$ messages generated at $x$ satisfying $q' \in R_{q_i} \wedge R_{q_i} = \texttt{follow}(q_i) \wedge R_{q_i} = R_{q_j} \wedge 1 \leq i, j \leq k$. Then we just select any message among $|r| \times k$ messages to be matched.

### 5.4.3 Message compression

In Algorithm 2, a message $m \in M_s$ generated at a vertex $v$ may be sent several times via different edges when $v$ has more than one outgoing edges labeled with the same symbol, which may incur excessive message passing cost. Since the messages are sent via the edges from one vertex to another vertex in Pregel, it is inevitable to send some message multiple times.

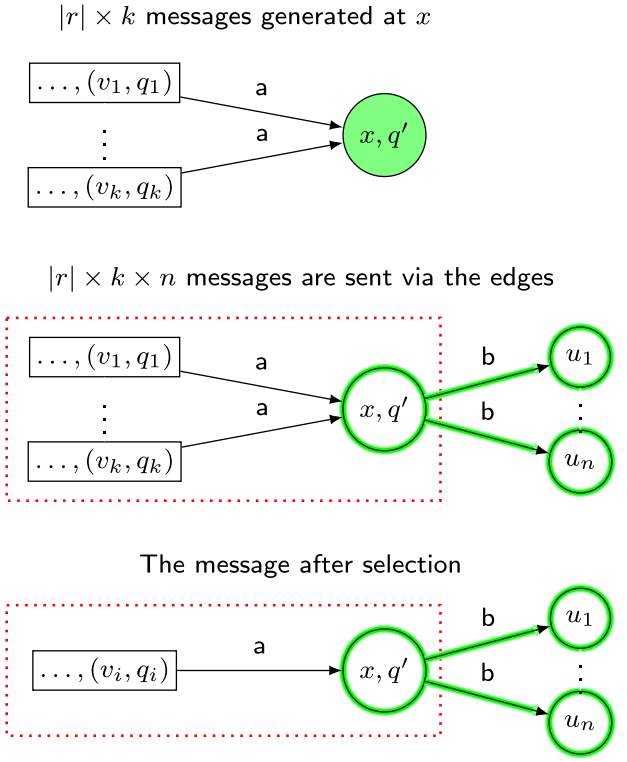



**Figure 8** Matching processing with message selection

**Definition 8** (*Duplicated-passing message*) In DP2RPQ evaluation, if a generated message can be sent via more than one edges from one vertex to multiple adjacent vertices labeled with the same symbol, it is called a *duplicated-passing message*.

**Algorithm 5** `messageCompression()`.

```
1  S_m(v) ← the sequence of the messages attached to the vertex v;
2  foreach m ∈ M_s ∧ m is a duplicated-passing message do
3      if m ∈ S_m(v) then
4          M_s ← M_s \ {m};
5          i ← the index of m in S_m(v);
6          m ← ((C_m, i), (v, q′));          /* the compressed message */
7          M_s ← M_s ∪ {m};
8      else if m ∉ S_m(v) then
9          M_s ← M_s \ {m};
10         S_m(v) ← append m to S_m(v);
11         i ← the index of m in S_m(v);
12         m ← ((C_m, i), (v, q′));          /* the compressed message */
13         M_s ← M_s ∪ {m};
```

To reduce the cost of message-passing, in Algorithm 5, we compress the *duplicated-passing messages*. Thus, Algorithm 5 is an optimized version of line 20 in Algorithm 2. We leverage a sequence $S_m(v)$ to keep the original uncompressed messages, which is attached to $v$. Then a *duplicated-passing messages* $m$ is compressed into a message $((C_m, i), (v, q'))$ to be sent at $v$, which only consists of two elements compared to the original message, where $C_m$ is a flag representing the message is compressed, $i$ denotes the index of the original message in $S_m(v)$, and $q'$ is the matched state of $v$ in the current superstep. Then, the compressed message is appended to the message set $M_s$ (line 9). Finally, when transforming a message to an equivalent path in line 18 in Algorithm 2, we uncompressed the partial message $((C_m, i), (v, q'))$ by employing index searching strategy, in which $i$ is regarded as the searching index to lookup the uncompressed message in $S_m(v)$. With the message-compression technique, the process of matching is shown as follows.

**I. Construction of the compressed messages.** For RDF graph $T_2$ in Example 2, in the third superstep, there exist the messages that can be compressed at $v_4$. When it receives a message $m' = ((v_2, 0), (v_3, 1))$, a new message $m = ((v_2, 0), (v_3, 1), (v_4, 3))$ will be built and then sent via the two outgoing edges $(v_4, \mathtt{d}, v_7)$ and $(v_4, \mathtt{d}, v_8)$ in Figure 9. In particular, the matching pairs in the message are represented as rectangle nodes, which are connected by the edges between the vertices. Next, the original message is compressed to become $((C_m, 0), (v_4, 3))$, where the index $i = 0$ since $m$ is the first element of $S_m(v)$. For the matching in the next superstep, $(v_4, 3)$ denotes the cached position of the compressed message. Meanwhile, the original message $m$ is appended to the sequence $S_m(v)$ of $v_4$. The message passing cost can be reduced dramatically when the original message is large in length.

**II. Uncompression of the answer paths.** When completing Algorithm 2, we collect all the answer paths by traversing $Val(v)$ over each vertex $v$, shown in line 5 in Algorithm 1.

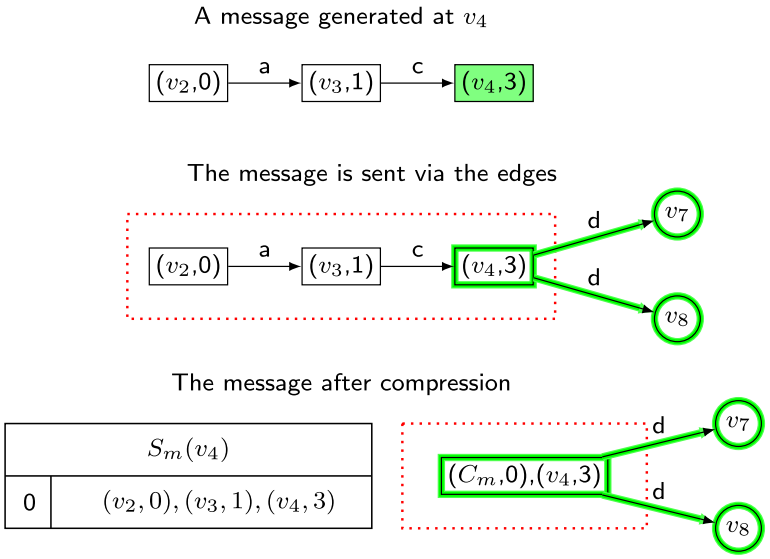



**Figure 9** The building of the compressed message in RDF graph $T_2$

In Example 2, the set of answer paths, i.e., $Val(v)$, is not empty at $v_7$ and $v_8$ merely, as shown in Table 3. Taking the answer (($C_m$,0),($v_4$,3),($v_7$,4)) in $Val(v_7)$ for example, as shown in Figure 10, we exhibit the processing of uncompressing the answer into the original answer path for presenting provenance-aware answer set. First, the compressed message is determined by the $C_m$. Next, since the index $i$=0, the original message is the first element of the sequence $S_m(v)$ of $v_4$. At last, we can uncompress the answer path by the original message $(v_2, 0)$, $(v_3, 1)$, $(v_4, 3)$.

In addition, we also employ the message selection technique in Section 5.4.2 to reduce the cost of message passing to help alleviate the counting-paths problem even further.

## 6 Experimental evaluation

In this section, we evaluate the performance of our method. We conducted extensive experiments to verify the efficiency and scalability of our proposed algorithms on both synthetic and real-world datasets.

**Table 3** The set of answer path $Val(v)$ cached in each vertex

| $v$ | $Val(v)$ |
|---|---|
| $v_7$ | $\{((C_m, 0), (v_4, 3), (v_7, 4)),((C_m, 1), (v_4, 3), (v_7, 4))\}$ |
| $v_8$ | $\{((C_m, 0), (v_4, 3), (v_8, 4)), ((C_m, 1), (v_4, 3), (v_8, 4)),$ $((C_m, 0), (v_4, 3), (v_8, 5)), ((C_m, 1), (v_4, 3), (v_8, 5))\}$ |

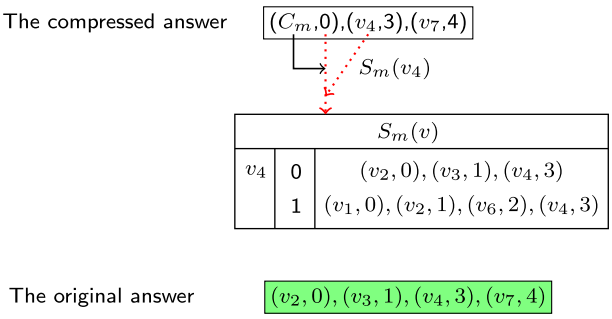



**Figure 10** The uncompressing of answer path in RDF graph $T_2$

### 6.1 Experimental settings

The proposed algorithms were implemented in Scala using Spark GraphX, which were deployed on a 10-site cluster in the *Tencent Cloud*.[3] Each site in this cluster installs a 64-bit CentOS 7.3 Linux operating system, with a 4-core CPU and 16GB memory. Our algorithms were executed on Java 1.8, Scala 2.11, Hadoop 2.7.4, and Spark 2.2.0.

For the verification of our algorithms, we use three datasets, including two benchmark datasets (LUBM[4] and WatDiv[5]) and a real-world dataset (DBpedia[6]), which are listed in Table 4. At present, there is no benchmark for RPQs. We designed twelve RPQs based on the characteristics of operators in regular expressions, including simple queries and complex queries, denoted by $Q_1$ to $Q_{12}$ in Table 5. For an RPQ $Q$, if it contains the closure operators $^*$ and/or $^+$, it is a complex query, otherwise it is a simple one. Since the complex query contains closure operations, the length of the language expressed by $r$ is uncertain, and it is likely to increase the number of supersteps in the query process. The above query is a combination of typical operators in the regular expression, which can cover all typical patterns of RPQs that match paths of different lengths.

### 6.2 Experimental results

We compared the performance of the basic algorithm and three optimized algorithms, which are denoted as `DP2RPQ`, `DP2RPQ`$_{opt}$ (vertex-computation optimization), `DP2RPQ`$_{msg}$ (message-communication reduction), and `DP2RPQ`$_{cnt}$ (counting-paths problem alleviation), respectively, using different queries over 3 datasets. In addition, we evaluated the scalability of `DP2RPQ` and `DP2RPQ`$_{opt}$. Finally, our approach is compared with `RDFPath` [22].

[3] https://cloud.tencent.com/
[4] http://swat.cse.lehigh.edu/projects/lubm/
[5] http://dsg.uwaterloo.ca/watdiv/
[6] http://wiki.dbpedia.org/

**Table 4** Datasets

| Datasets | $\|V\|$ | $\|E\|$ |
|---|---|---|
| LUBM10 | 314,853 | 1,316,700 |
| LUBM100 | 3,301,718 | 1,387,997 |
| LUBM200 | 6,574,860 | 27,643,644 |
| WatDiv10 | 158,118 | 1,109,678 |
| WatDiv100 | 1,526,677 | 10,958,704 |
| WatDiv200 | 3,228,213 | 24,098,747 |
| DBpedia | 5,526,330 | 18,295,010 |

### 6.2.1 Efficiency of the algorithms

I. **Different queries over some dataset.** We evaluate twelve queries ($Q_1$ to $Q_{12}$) over LUBM100 and DBpedia datasets, respectively.

The results are shown in Figure 11. Obviously, the basic and the optimized algorithm, i.e., DP2RPQ and DP2RPQ$_{opt}$, return answers to the queries in limited time, which verifies the efficiency of the algorithms. Although the size of LUBM100 is relatively smaller than the size of DBpedia dataset, the query time of an RPQ over LUBM100 dataset is not always less than that over DBpedia dataset, which is due to a large number of results when evaluating $Q_1$ and $Q_6$.

Especially, the experimental results on LUBM100 and DBpedia datasets indicate that DP2RPQ$_{opt}$ performs better than DP2RPQ in all cases. When evaluating on LUBM100 dataset, we notice that the query time of $Q_6$ is more than other queries. In fact, the number of the intermediate partial results has reached millions of paths in the query processing of $Q_6$. When evaluating on DBpedia dataset, for DP2RPQ$_{opt}$, the most significant improvement is for the most complex query $Q_{11}$, which takes 52.44% of the query time of DP2RPQ. Meanwhile, the average improvement ratio is 40.62%.

II. **Same queries over the datasets in different sizes.** We evaluate the queries in Table 5 over the LUBM datasets and WatDiv datasets with different scale factors, i.e., 10, 100, and 200, respectively.

(1) *LUBM dataset.* Both DP2RPQ and DP2RPQ$_{opt}$ are executed on LUBM datasets, whose results are illustrated in Figure 12. It can be observed that the time of the algorithms scales linearly with the size of the data. However, the time of $Q_1$, $Q_3$, and $Q_6$ increases rapidly along with the increasing size of the data because the query results

**Table 5** Regular path queries

| | |
|---|---|
| Simple RPQs | |
| $Q_1 = (a/b)\|(c/d)$ | $Q_3 = a/b/(c\|d\|e)$ |
| $Q_2 = a/b/c/d$ | $Q_4 = (a\|b)/(a\|c)$ |
| Complex RPQs | |
| $Q_5 = (a/a)^+$ | $Q_9 = (a/b)^+\|(c/d)^+$ |
| $Q_6 = (a\|b\|c)^+$ | $Q_{10} = (a/(b/c)^*)^+\|(d/e)^+$ |
| $Q_7 = a^+$ | $Q_{11} = ((a/b)/(c\|d)^*)^+/(e\|f)^*$ |
| $Q_8 = a/(a\|b\|c)^*$ | $Q_{12} = (a\|b)^+/(c\|d)^+$ |

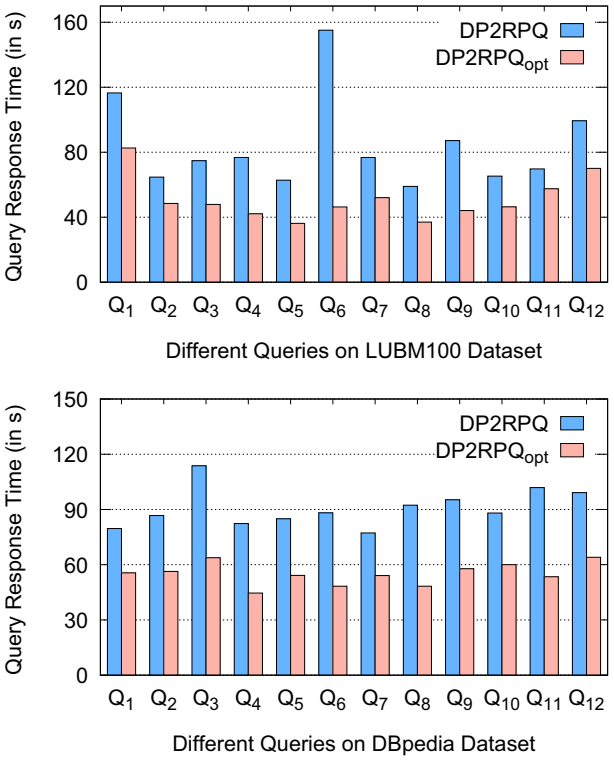



**Figure 11** The experimental results of LUBM100 and DBpedia datasets

have reached a relatively large scale (i.e., millions of paths). In most cases, the query time of DP2RPQ$_{opt}$ is reduced significantly compared with DP2RPQ, which verifies the effectiveness of our optimization strategies. However, when $Q_3$ and $Q_{11}$ are evaluated on LUBM10, DP2RPQ$_{opt}$ takes slightly longer time than DP2RPQ. The reason is that $\Sigma_r$ of these queries involve more various symbols than other queries and LUBM10 is relatively small in size, which result in filtering out fewer useless edges than other queries. We can see as a general rule that the larger the dataset is, the better DP2RPQ$_{opt}$ performs. The optimization effect of DP2RPQ$_{opt}$ for all queries on LUBM200 has become much more significant than that on other datasets.

(2) *WatDiv dataset.* Four representative queries ($Q_4$, $Q_6$, $Q_7$, and $Q_{10}$) are selected from Table 5 and evaluated on the WatDiv datasets of varying scale factors (SF), i.e., 10, 100, and 200, respectively. In Figure 13, it can be observed that DP2RPQ$_{opt}$ outperforms DP2RPQ for all queries. The query time of DP2RPQ$_{opt}$ is on average 41.59% of that of DP2RPQ. Due to the diversified predicates in WatDiv, DP2RPQ$_{opt}$ can reduce the times of traversals and filter out more useless edges in comparison with the evaluation on the LUBM datasets.

### 6.2.2 Scalability of the algorithms

I. **Varying site number.** In order to evaluate the scalability of DP2RPQ and DP2RPQ$_{opt}$, we use the LUBM100 and DBpedia datasets, with four representative queries ($Q_2$, $Q_4$, $Q_5$, and $Q_{11}$) selected from Table 5.

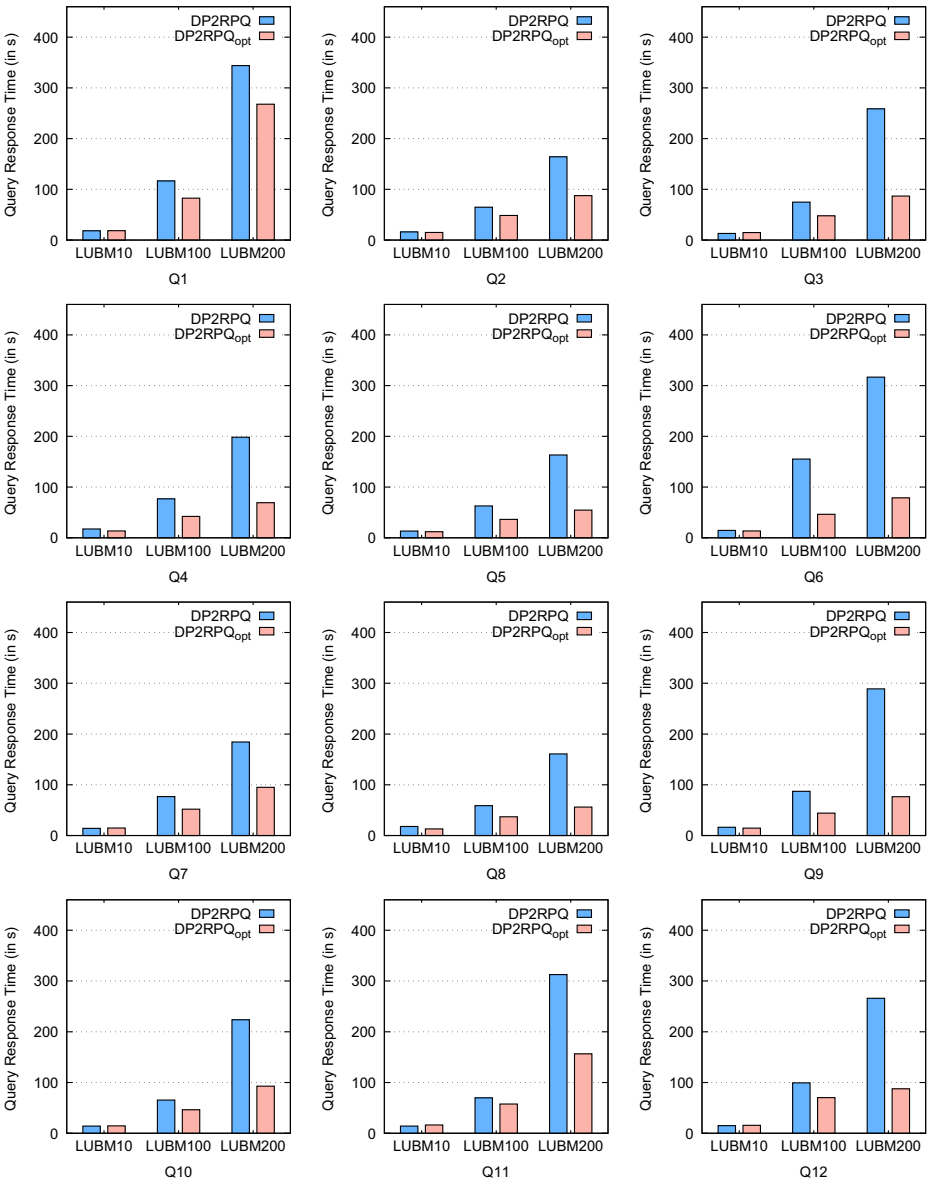



**Figure 12** The experimental results of efficiency on LUBM datasets

The query time on the different number of sites, varying from 4 to 10, is shown in Figure 14. The query time of DP2RPQ and DP2RPQ$_{opt}$ decreases with the number of sites increasing, which confirms that our algorithms can take full advantage of the vertex-centric Pregel framework for graph parallel computing. Moreover, the average speedup ratio of DP2RPQ is 1.21 times of DP2RPQ$_{opt}$.

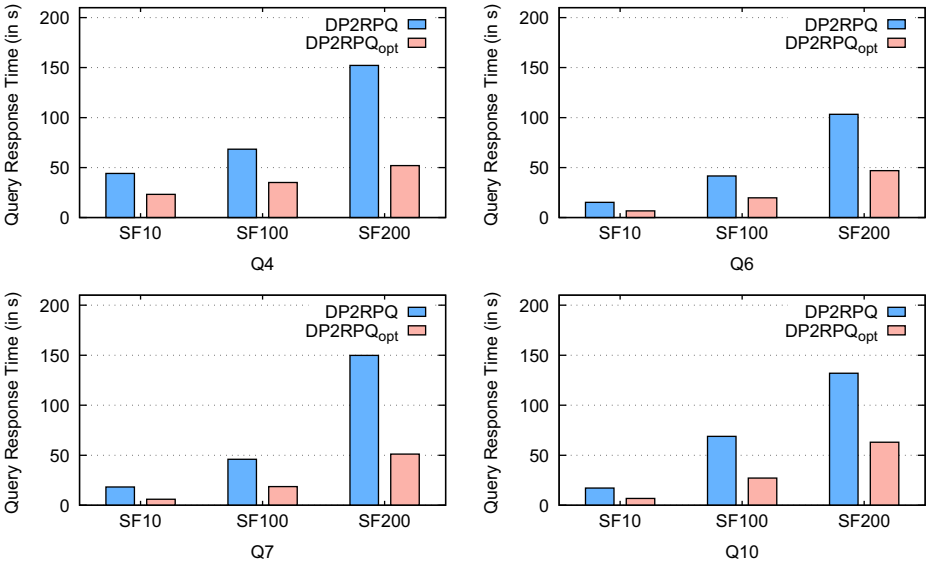
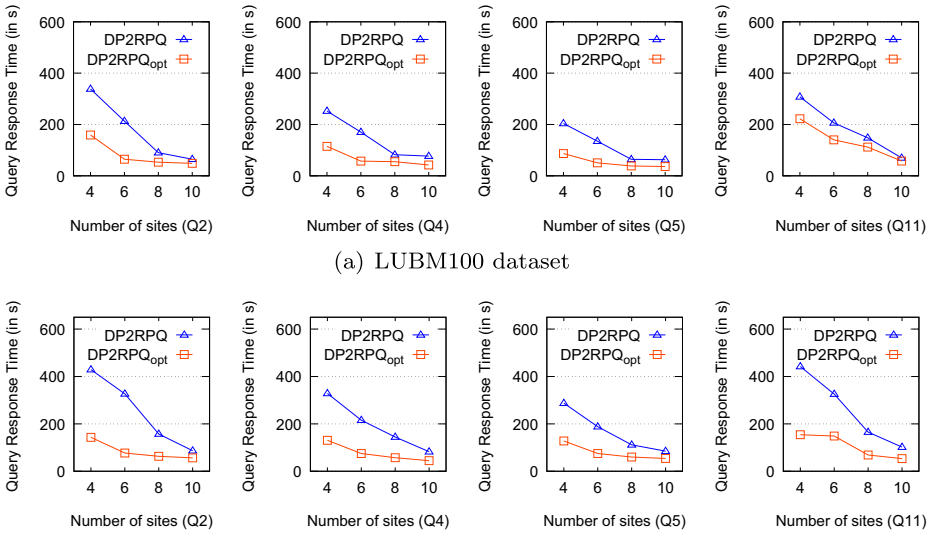



**Figure 13** The experimental results of efficiency on WatDiv datasets

II. **Maximum time in vertex computation.** It is obvious that the maximum execution time in each vertex computation is the dominant cost except the message passing cost. We selected four queries ($Q_2$, $Q_3$, $Q_6$ and $Q_8$) and evaluted them over LUBM datasets with different sizes (LUBM10, LUBM100, LUBM200).




(a) LUBM100 dataset

(b) DBpedia dataset

**Figure 14** Scalability by varying the number of sites

The optimized techniques, i.e., edges-filtering and candidate-states strategies, reduce the maximum execution time in vertex computation, whose results are shown in Figure 15. It can be observed that the maximum time of vertex computation scales linearly with the size of the data. Generally, DP2RPQ$_{opt}$ performs better than DP2RPQ in terms of maximum time in the vertex computation.

### 6.2.3 Efficiency of the message-communication optimization

Based on the algorithm of DP2RPQ$_{opt}$, the optimization strategies including sending message pruning and variable-length-byte encoding are further implemented for reducing the communication cost, which is called DP2RPQ$_{msg}$. To verify the impact of sending message pruning on reducing the sending message size, we compare the maximum length of messages to be sent in all supersteps between DP2RPQ and DP2RPQ$_{msg}$. Eight queries ($Q_1$ to $Q_7$, $Q_9$) are selected from Table 5 to evaluate on the LUBM datasets with different sizes, i.e., LUBM10, LUBM100, and LUBM200, respectively.

It presents the maximum length of sending messages in Table 6, where $R_{msg}$ is the reduction rate of the message length after message-communication optimization. As can be seen, the maximum length of sending messages is reduced significantly by the pruning-message strategy. Moreover, we notice that the reduction rate is more than 50% in all cases, which indicates that DP2RPQ$_{msg}$ reduce the number of messages efficiently.

### 6.2.4 Effectiveness of the counting-paths alleviation strategy

Another optimized algorithm, called DP2RPQ$_{cnt}$, was implemented with the message-compression and message-selection techniques to partly address the counting-paths problem.

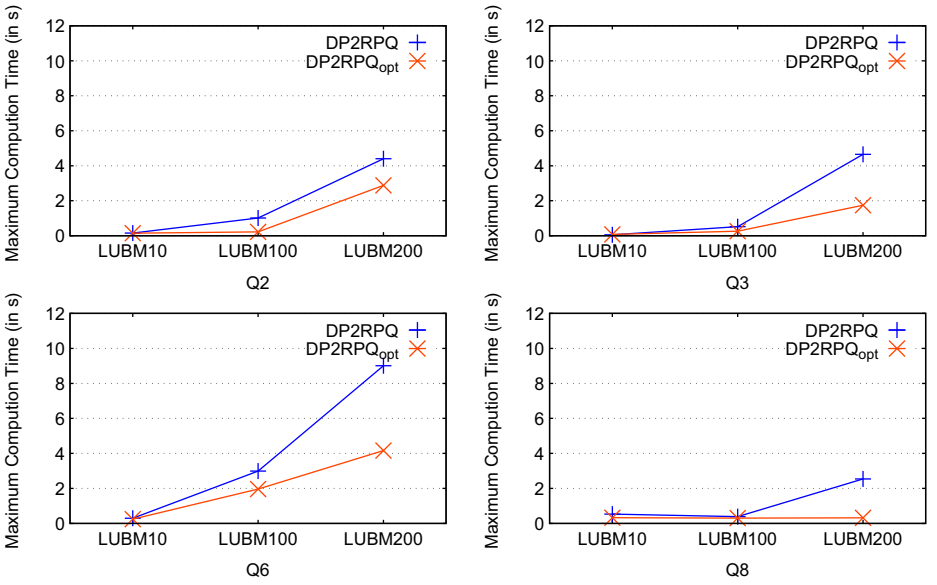



**Figure 15** The experimental results of the maximum time in vertex computation

**Table 6** The experimental results of the maximum length of sending messages

| The maximum length of sending messages | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | LUBM10 | | | LUBM100 | | | LUBM200 | | |
| | DP2RPQ$_{msg}$ | DP2RPQ | $R_{msg}$ | DP2RPQ$_{msg}$ | DP2RPQ | $R_{msg}$ | DP2RPQ$_{msg}$ | DP2RPQ | $R_{msg}$ |
| $Q_1$ | 20 | 54 | 63.0% | 20 | 60 | 66.7% | 23 | 69 | 66.7% |
| $Q_2$ | 44 | 955 | 95.4% | 190 | 1090 | 82.6% | 218 | 1090 | 80.0% |
| $Q_3$ | 12 | 64 | 81.3% | 19 | 72 | 73.6% | 23 | 72 | 55.6% |
| $Q_4$ | 20 | 54 | 63.0% | 20 | 60 | 66.7% | 23 | 66 | 65.2% |
| $Q_5$ | 19 | 60 | 68.3% | 20 | 60 | 66.7% | 20 | 60 | 66.7% |
| $Q_6$ | 1 | 170 | 99.4% | 1 | 1865 | 99.9% | 1 | 1865 | 99.9% |
| $Q_7$ | 1 | 2 | 50.0% | 1 | 2 | 50.0% | 1 | 2 | 50.0% |
| $Q_9$ | 13 | 39 | 66.7% | 19 | 56 | 66.1% | 23 | 66 | 65.2% |

Due to the limited lengths of the answers to the queries in Table 5, DP2RPQ$_{cnt}$ cannot reach its full potential. To this end, we generate RDF graphs w.r.t. the data model of WatDiv[7] by constructing structures like $T_4$ in Figure 7. Meanwhile, we design an RPQ $Q_c = x_1/x_2/\ldots/x_{10}$ by covering the predicates that may generate the Cartesian product. It is obvious that the lengths of the answer paths to $Q_c$ are much longer than that of the previous queries.

In Figure 16, it can be observed that DP2RPQ and DP2RPQ$_{opt}$ cannot finish within the time limit ($10^4$s), denoted by INF, while DP2RPQ$_{cnt}$ can return the answers in 78.39s and 377.56s over the RDF graphs that contain 1 million and 10 million triples, respectively. Thus, DP2RPQ$_{cnt}$ can effectively alleviate the counting-paths problem.

### 6.2.5 Performance comparison between DP2RPQ and RDFPath

The existing distributed method for provenance-aware regular path query is rare. To the best of our knowledge, RDFPath is the only method for answering RPQs using a distributed setting. Thus, in this section, we compare our approach DP2RPQ$_{msg}$ with RDFPath. Although RDFPath is designed based on the MapReduce framework, it is implemented using Spark. RDFPath is deployed on a 10-site cluster in the *Tencent Cloud* and executed on Hadoop 2.6.0 and Spark 1.6.1.

I. **Efficiency.** We first evaluated the efficiency of DP2RPQ$_{msg}$ and RDFPath, with four simple queries ($Q_1$ to $Q_4$) and four complex queries ($Q_5$, $Q_6$, $Q_7$, and $Q_9$). In RDFPath, a path query is composed by a sequence of basic navigational component denoted as *location steps*, it is very suitable for handling connection operations rather than the closure operators * and/or $^+$. In fact, the regular path query is converted into an SQL in the processing of RDFPath, which leads to the number of join operation is proportional to the length of path in the query. In particular, the semantics of the queries that RDFPath can handle are limited. If a query is not supported by RDFPath, its query response time is empty.

[7] http://dsg.uwaterloo.ca/watdiv/watdiv-data-model.txt

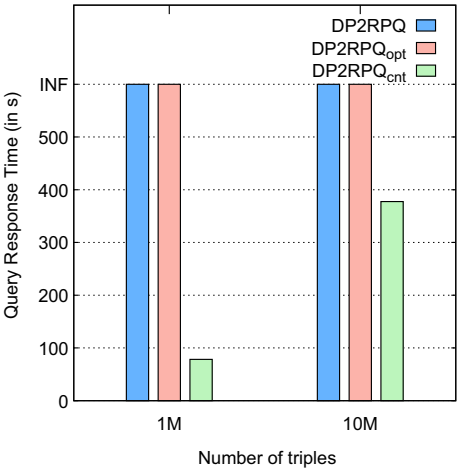



**Figure 16** The experimental results of the counting-paths alleviation

(1) *LUBM dataset.* We evaluated DP2RPQ$_{msg}$ and RDFPath with four simple queries over LUBM datasets, which is shown in Figure 17. It can be observed that the experimental results of $Q_2$ indicate that RDFPath performs better than DP2RPQ$_{msg}$, since $Q_2$ only contain the concatenation operator that RDFPath is suitable for handing. However, RDFPath only supports $Q_2$ since the other queries all contain the alternation operator not supported by RDFPath.

The number of edges can be specified in RDFPath query, so a fixed-length join operation can be used to simulate approximately the closure operations. For $Q_7 = a^+$, we convert it to an approximate query in RDFPath, denoted as $a$ <$k$>, where $k$ is the number of

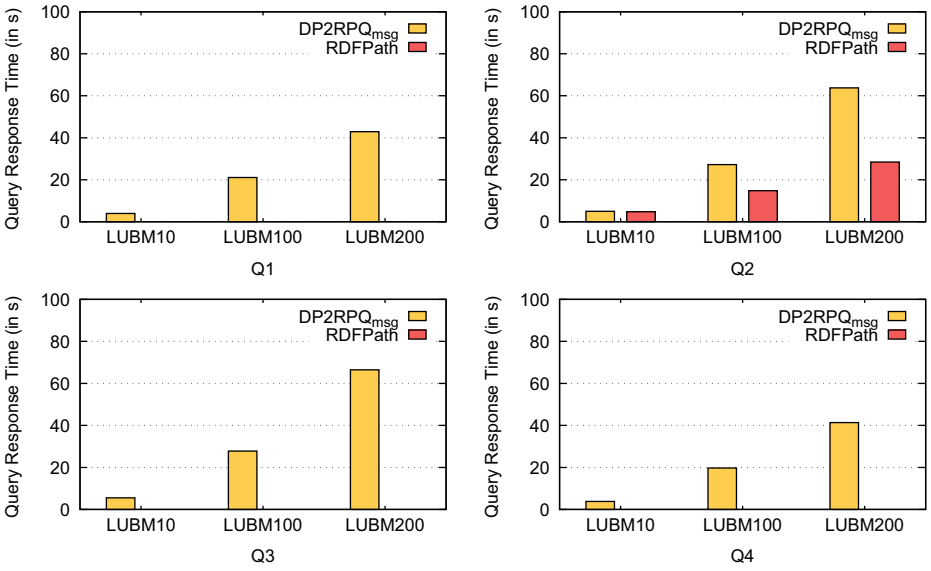



**Figure 17** The experimental contrast results of simple queries on LUBM datasets

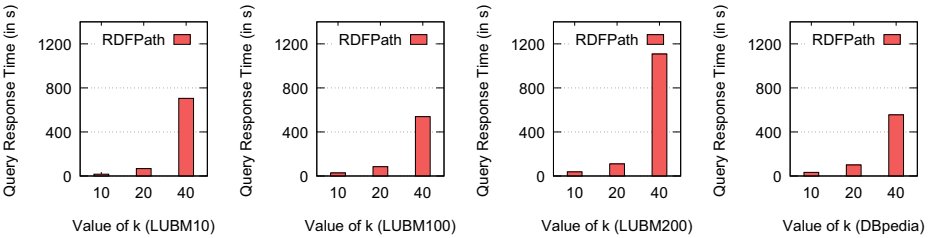



**Figure 18** Influence of different $k$ values on query performance of RDFPath

occurrences of the label $a$. Obviously, the expression power of $a$ $<k>$is weaker than $a^+$ in RPQ. Moreover, with the value of $k$ increasing, the query time increases rapidly, as shown in Figure 18. When $k = 15$, we compare the complex query response time of DP2RPQ$_{msg}$ and RDFPath. It can be observed that DP2RPQ$_{msg}$ has better performance than RDFPath, as shown in Figure 19.

(2) *DBpedia dataset.* The real-world dataset DBpedia has richer predicates and more complex structure, which can be used to design RPQs with a long length of path. Since RDFPath can only perform two queries of $Q_2$ and $Q_7$, in order to compare the performance of DP2RPQ$_{msg}$ and RDFPath, we design two additional queries $Q_{13} = a/b/\ldots/k/l$ and $Q_{14} = a/b/\ldots/o/p$ with lengths of 12 and 16, respectively. It shows that the RDFPath can only execute $Q_2$, $Q_7$, $Q_{13}$, and $Q_{14}$, and our method is more effective than RDFPath in most cases in Figure 20. Although the query time of DP2RPQ$_{msg}$ on $Q_2$ is longer than RDFPath, the semantics of RPQs are far richer than RDFPath on the whole.

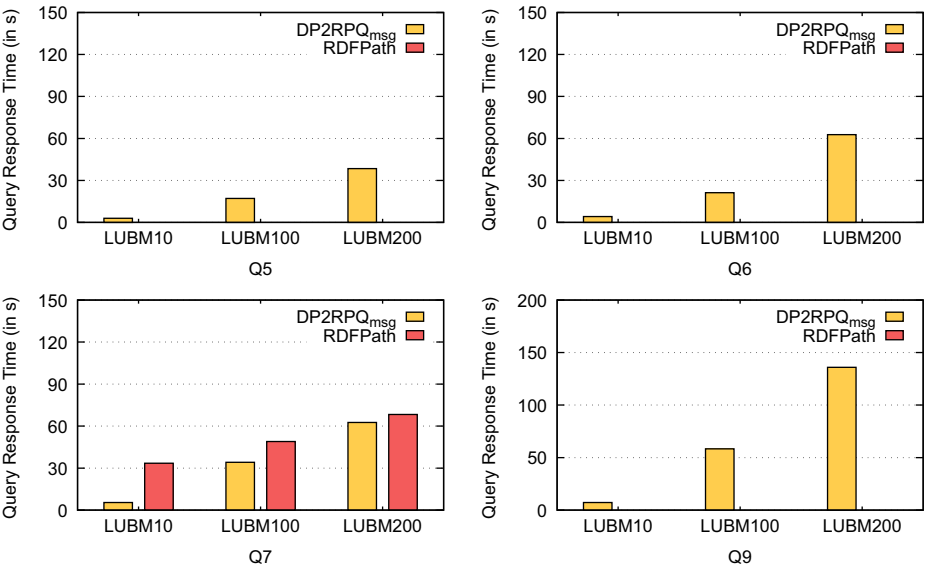



**Figure 19** The experimental contrast results of complex queries on LUBM datasets

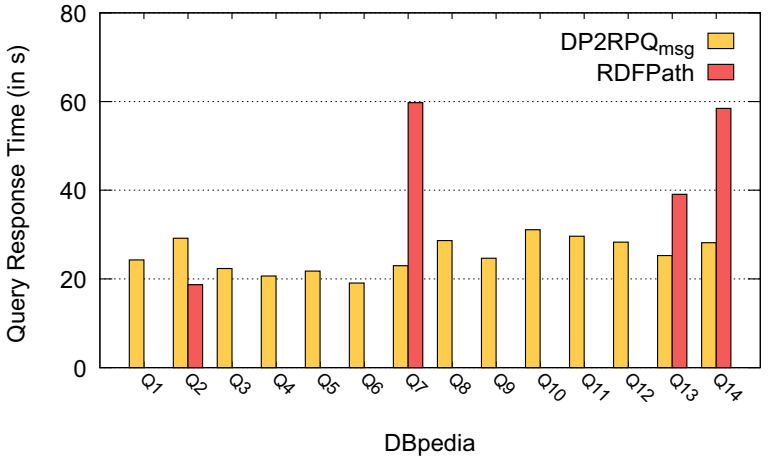
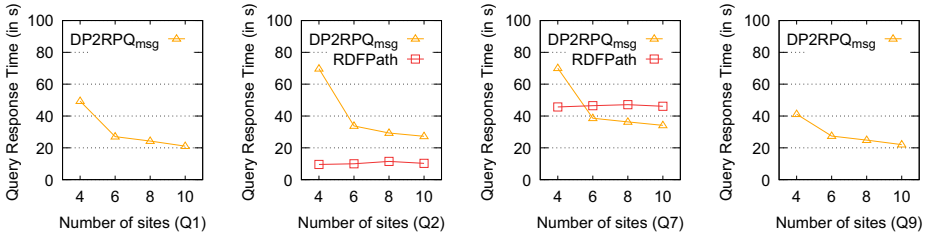
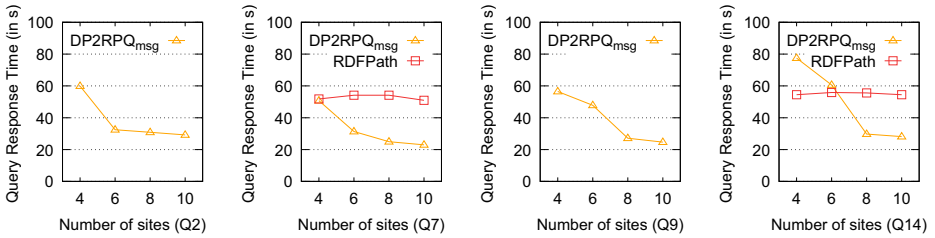



**Figure 20** The experimental contrast results on DBpedia datasets

II. **Scalability.** To evaluate the scalability with different number of sites, we used LUBM100 and DBpedia as the datasets and varied the number of sites from 4 to 10. The experimental results of four queries ($Q_1$, $Q_2$, $Q_7$, and $Q_9$) on LUBM100 are shown in Figure 21a, it can be seen that the query time of DP2RPQ$_{opt}$ decreases with the number of sites increasing. However, the change of query time is not obvious on the two queries ($Q_2$ and $Q_7$) in RDFPath. Figure 21b depicts the scalability evaluation of DP2RPQ$_{msg}$ and RDFPath for DBpedia dataset on queries ($Q_2$, $Q_7$, $Q_9$, and $Q_{14}$), in which the query time of DP2RPQ$_{msg}$ decreases more significantly than RDFPath with




(a) LUBM100 dataset




(b) DBpedia dataset

**Figure 21** Scalability by varying the number of sites

the number of sites increasing. It confirms that our algorithm can take full advantage of the graph parallel computing model in comparison with `RDFPath`.

## 7 Conclusion

In this paper, we propose a novel method for answering provenance-aware RPQs over large RDF knowledge graphs by using the Pregel parallel graph computing framework. We also devise three optimization techniques, among which the edge-filtering and candidate-states techniques can significantly improve the performance of RPQs, the sending message pruning and variable-length-byte encoding can reduce intermediate results and communication overhead greatly, and the message-compression and message-selection strategies are employed to alleviate the counting-paths problem. The extensive experiments were conducted on both synthetic and real-world datasets, which have verified the effectiveness, efficiency, and scalability of our method.

**Acknowledgements** This work is supported by the National Natural Science Foundation of China (61572353), the Natural Science Foundation of Tianjin (17JCYBJC15400).

## References

1. Arenas, M., Conca, S., Pérez, J.: Counting beyond a yottabyte, or how sparql 1.1 property paths will prevent adoption of the standard. In: Proceedings of the 21st International Conference on World Wide Web, pp. 629–638. ACM (2012)
2. Avery, C.: Giraph: Large-scale graph processing infrastructure on hadoop. Proc. Hadoop Summit Santa Clara **11**(3), 5–9 (2011)
3. Bai, Y., Wang, C., Ning, Y., Wu, H., Wang, H.: G-path: Flexible path pattern query on large graphs. In: Proceedings of the 22nd International Conference on World Wide Web, pp. 333–336. ACM (2013)
4. Bai, Y., Wang, C., Ying, X., Wang, M., Gong, Y.: Path pattern query processing on large graphs. In: IEEE Fourth International Conference on Big Data & Cloud Computing (2014)
5. Bai, Y., Wang, C., Ying, X.: Para-G: Path pattern query processing on large graphs. World Wide Web **20**(3), 515–541 (2017)
6. Barceló, P., Libkin, L., Lin, A.W., Wood, P.T.: Expressive languages for path queries over graph-structured data. ACM Trans. Database Syst. (TODS) **37**(4), 31 (2012)
7. Brüggemann-Klein, A.: Regular expressions into finite automata. Theor. Comput. Sci. **120**(2), 197–213 (1993)
8. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM (JACM) **11**(4), 481–494 (1964)
9. Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M.Y.: Answering regular path queries using views. In: 16th International Conference on Data Engineering, 2000. Proceedings, pp. 389–398. IEEE (2000)
10. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Commun ACM **51**(1), 107–113 (2008)
11. Dey, S., Cuevas-Vicenttín, V., Köhler, S., Gribkoff, E., Wang, M., Ludäscher, B.: On implementing provenance-aware regular path queries with relational query engines. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops, pp. 214–223. ACM (2013)
12. Gerbessiotis, A.V., Valiant, L.G.: Direct bulk-synchronous parallel algorithms. J. Parallel Distrib. Comput. **22**(2), 251–267 (1994)
13. Harris, S., Seaborne, A., Prud'hommeaux, E.: Sparql 1.1 query language. W3C Recommend., **21**(10) (2013)
14. Jupp, S., Malone, J., Bolleman, J., Brandizi, M., Davies, M., Garcia, L., Gaulton, A., Gehant, S., Laibe, C., Redaschi, N., et al.: The ebi rdf platform: linked open data for the life sciences. Bioinformatics **30**(9), 1338–1339 (2014)
15. Koschmieder, A., Leser, U.: Regular path queries on large graphs. In: International Conference on Scientific and Statistical Database Management, pp. 177–194. Springer (2012)

16. Kostylev, E.V., Reutter, J.L., Romero, M., Vrgoč, D.: Sparql with property paths. In: International Semantic Web Conference, pp. 3–18. Springer (2015)
17. Lehmann, J., Isele, R., Jakob, M., Jentzsch, A., Kontokostas, D., Mendes, P.N., Hellmann, S., Morsey, M., Van Kleef, P., Auer, S., et al.: Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web **6**(2), 167–195 (2015)
18. Libkin, L., Martens, W., Vrgoč, D.: Querying graph databases with XPath. In: Proceedings of the 16th International Conference on Database Theory, pp. 129–140. ACM (2013)
19. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 135–146. ACM (2010)
20. Nolé, M., Sartiani, C.: Regular path queries on massive graphs. In: Proceedings of the 28th International Conference on Scientific and Statistical Database Management, p. 13. ACM (2016)
21. Nolé, M., Sartiani, C.: A distributed implementation of GXPath. In: EDBT/ICDT Workshops (2016)
22. Przyjaciel-Zablocki, M., Schätzle, A., Hornung, T., Lausen, G.: Rdfpath: Path query processing on large rdf graphs with mapreduce. In: Extended Semantic Web Conference, pp. 50–64. Springer (2011)
23. Tong, Y., She, J., Meng, R.: Bottleneck-aware arrangement over event-based social networks: The max-min approach. World Wide Web **19**(6), 1151–1177 (2016)
24. Wang, X., Ling, J., Wang, J., Wang, K., Feng, Z.: Answering provenance-aware regular path queries on rdf graphs using an automata-based algorithm. In: Proceedings of the 23rd International Conference on World Wide Web, pp. 395–396. ACM (2014)
25. Wang, X., Wang, J.: Provrpq: An interactive tool for provenance-aware regular path queries on rdf graphs. In: Australasian Database Conference, pp. 480–484. Springer (2016)
26. Wang, X., Wang, J., Zhang, X.: Efficient distributed regular path queries on rdf graphs using partial evaluation. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, pp. 1933–1936. ACM (2016)
27. Wang, M., Zhang, J., Liu, J., Hu, W., Wang, S., Li, X., Liu, W.: Pdd graph: Bridging electronic medical records and biomedical knowledge graphs via entity linking. In: International Semantic Web Conference, pp. 219–227. Springer (2017)