TraPath: Fast Regular Path Query Evaluation on Large-Scale RDF Graphs

Xin Wang[1,2], Guozheng Rao[1,2,⋆], Longxiang Jiang[1,2], Xuedong Lyu[1,2], Yajun Yang[1,2], and Zhiyong Feng[1,2]

[1] School of Computer Science and Technology, Tianjin University, Tianjin, China
[2] Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin, China
{wangx,rgz,yjyang,zyfeng}@tju.edu.cn, {lxjiang2012,lxd.1990}@gmail.com

Abstract. Regular path queries, or RPQs, are basic querying mechanisms on graphs that play an increasingly important role over the past decade. In recent years, large amounts of RDF data are published on the Web since the development of Linked Data. Such a large-scale of data has posed serious challenges to the efficiency of RPQs. In this paper, we devise a double-layer bi-directional index structure that has a linear space complexity, and propose a novel traversal-based algorithm TraPath that achieves the fast evaluation of RPQs by using the index structure. We conduct extensive experiments to evaluate and compare the performance of our prototype system and the Sesame RDF repository with a real-world RDF dataset from DBpedia. The experimental results show that TraPath significantly outperforms the state-of-the-art methods.

Keywords: regular path query, RDF graph, large-scale, index structure.

1 Introduction

The Semantic Web is considered as the next-generation of the current Web, on which information is machine-understandable. The standard data model of the Semantic Web is the Resource Description Framework (RDF) [1], which describes resources with triples of the form (s, p, o) where s is the subject, p the predicate, and o the object. Since each triple states a relation from its subject to object, an RDF dataset, consisting of a set of triples, is represented as a directed labeled graph. Queries on RDF graphs belong to subgraph matching queries or path queries, rather than relational queries. Therefore, traditional RDBMS is not applicable for the management of large-scale RDF data. As a basic querying mechanism in RDF databases, RPQs are recognized as an essential operation to explore more complex relationships between recourses in RDF graphs. In particular, researchers in some areas that have been equipped with relatively rich RDF datasets, such as bioinformatics [2] and social networking [3], have tried to use different forms of RPQs to gain new knowledge from large RDF graphs. In addition, as the current standard query language for RDF, SPARQL 1.1 [4]

⋆ Corresponding author.

F. Li et al. (Eds.): WAIM 2014, LNCS 8485, pp. 372–383, 2014.

has introduced a new feature called *property paths* to realize the functionality of RPQs. Therefore, the efficiency of evaluating RPQs on large RDF graphs is of great importance.

Mendelzon and Wood [5] have proven that the evaluation of RPQs by simple paths on graph is NP-complete, which indicates regular path queries have a high complexity. In addition, the RDF graphs have proliferated significantly with the development of Linked Data [6], which posed serious challenges to graph data management. As a consequence, traditional methods based on triple indexes and sort-merge joins, which need to load large amounts of triples into memory, exhibit low performance and cannot be adapted to the big data scenario.

In this paper, we propose an efficient method for answering RPQs on large-scale RDF data. The contributions of our paper are summarized as follows:

- We devise a *double-layer bi-directional* index structure that covers all regular path queries and has a linear space complexity.
- Based on this index structure, we propose a novel *traversal-based* algorithm, named TraPath, for searching paths on large RDF graphs, which achieves the efficient evaluation of RPQs.
- We perform extensive experiments to evaluate and compare the performance of our method and Sesame. The experimental results show that TraPath significantly outperforms the state-of-the-art methods.

The rest of the paper is organized as follows. After a review of related work in Section 2, we introduce the necessary definitions and formalize the RPQ problem in Section 3. In Section 4, we present the double-layer bi-directional index structure that covers all regular path queries and has a linear space complexity. In Section 5, we describe our algorithms which traverse RDF graphs bi-directionally in parallel. In Section 6 we evaluate our work by a series of experiments. Finally, we conclude the paper in Section 7.

2 Related Work

We focus on regular path queries on large-scale RDF graphs and review related work from two aspects separately, i.e., RDF indexes and approaches to RPQs.

Indexes for RDF data can be divided into two categories, B^+-tree-based indexes and Bigtable-based indexes. RDF-3X [7] introduces the concept of sextuple indexing based on B^+-tree, and processes triple pattern queries efficiently. However, RDF-3X also employees the multi-way join operations to implement more complex SPARQL queries, which may incur the high time overhead due to the large number of intermediate results. CumulusRDF [8] and Jingwei+ [9] both implement triple indexes based upon Bigtable, but their performance is limited since the mechanism of the super-column may reduce the efficiency of the operations on triple indexes.

SPARQL is the W3C recommended query language for RDF. However, the functionality of RPQs has not been proposed until the *property paths* are introduced in SPARQL 1.1. In the past few years, researchers had concentrated

on studying SPARQL including implementations, speeding up queries, and extensions to support RPQs. Sesame and Jena are two state-of-the-art single-machine implementations of SPARQL, while they provide weak support for RPQs. PSPARQL [10] is an RPQ extension to SPARQL, but it does not provide the implementation method for the language. SPARQLeR [11] also extends SPARQL with regular paths, and Koschmieder and Leser [12] propose to use rare labels for answering RPQs. However, both of these approaches are implemented as a bi-directional breadth-first search, which is obviously different from our approach. Besides, both approaches employ the *counting paths* semantics that has a PSPACE complexity.

Our approach differs from the above work significantly. On one hand, we use the flat structure to construct indexes on Bigtable, which achieves both high performance and scalability. The *double-layer bi-directional* index structure is built specifically for RPQs, which has superior performance for joining triples. On the other hand, our approach to answering RPQs is implemented as a *depth-first* search that has the tremendous performance advantages for finding paths on large-scale RDF graphs. In addition, we simplify the problem by restricting the semantics of queries, which can be tackled in polynomial time.

3 Definitions

Before introducing our work in detail, we give several definitions of RDF graphs, in/out-degree nodes of a predicate, and the syntax of our RPQ language. In this paper, we define an RDF graph as a set of triples that can be mapped to a general graph of the form $G = (V, E)$.

Definition 1. *An RDF graph is defined as* $T = \{(s, p, o) \mid s \in S, p \in P, o \in O\}$, *in which we define the set of subjects as* $S = \{s \mid s = lab(v), v \in V, \forall v_i \in V, \langle v, v_i\rangle \in E\}$, *the set of predicates as* $P = \{p \mid p = lab(\langle v_i, v_j\rangle), \forall v_i, v_j \in V, \langle v_i, v_j\rangle \in E\}$ *and the set of objects as* $O = \{o \mid o = lab(v), v \in V, \forall v_i \in V, \langle v_i, v\rangle \in E\}$. $lab()$ *is the function that returns the label of a vertex or an edge.*

Definition 1 describes the logical model of an RDF graph that differs from a general graph. If we define an RDF graph in the form of both T and G, the following properties hold: (1) $lab(V) = S \cup O$, and $lab(E) = P$; (2) $|V| = |S \cup O|$, but $|E| \geqslant |P|$, since a subject s or an object o of a triple in T corresponds to a unique label of a vertex in G, while a predicate p may correspond to more than one edges in G.

In an RDF graph, a path is composed of consecutive edges, each edge is represented as a triple (s, p, o). If we consider the predicate p as a vertex, then it has both in-degree (subjects) nodes and out-degree (objects) nodes.

Definition 2. *We define in-degree subjects* $PS_{p'} = \{s \mid \forall s \in S, \forall o \in O, (s, p', o) \in T\}$, *in-degree subjects with specified object* $POS_{p'o'} = \{s \mid \forall s \in S, (s, p', o') \in T\}$, *out-degree objects* $PO_{p'} = \{o \mid \forall s \in S, \forall o \in O, (s, p', o) \in T\}$, *and out-degree objects with specified subject* $PSO_{p's'} = \{o \mid \forall o \in O, (s', p', o) \in T\}$.

The formulas in Definition 2 describe the in/out-degree nodes of a specified predicate p, from which we obtain: (1) For a predicate $p' \in P$, $PSO_{p's}$, $POS_{p'o}$, $PS_{p'}$ and $PO_{p'}$ are all not empty, iff $\exists s \in S, \exists o \in O, (s, p', o) \in T$; (2) $POS_{p'o} \subseteq PS_{p'}$, $PSO_{p's} \subseteq PO_{p'}$; and (3) $POS_{p'o} = PS_{p'}$, iff $PO_{p'} = \{o\}$, and similarly, $PSO_{p's} = PO_{p'}$, iff $PS_{p'} = \{s\}$.

Definition 3. *The regular expression of path queries is defined as:* $r = p \mid -r \mid (r/r) \mid (r|r) \mid r^*$, $p \in P$. *We define the syntax of our RPQ queries as* $Q = (?x|s, r, ?y|o)$, $s \in PS_{p_1}, o \in PO_{p_n}$, p_1 *and* p_n *is the first and last edge of* r *respectively.*

Definition 3 gives a recursive definition to the regular paths, which covers all the possible forms of RPQs. However, the complexity of RPQs with counting paths semantics is PSPACE-complete [5], which is considered infeasible for large-scale RDF graphs. To simplify the problem, in our RPQ semantics, we just find one satisfiable path for an RPQ (i.e., not counting paths), and we also allow non-simple paths as the answers. Definition 3 also introduces the syntax of our RPQ language. For example, if Tom wants to find that whether there exists a person who is a friend of his friends and that person also has a pet, then the query is expressed as: $(Tom, isFriend/isFriend/hasPet, ?y)$.

4 Index Structures

This section presents the double-layer bi-directional index structure that is abbreviated as DB-Index. First we give a detailed introduction to DB-Index. Then we describe the procedure for the index construction.

4.1 DB-Index

Definition 2 describes our new perspective of the primitive path edge. Under normal circumstances, there is no need to specify the in/out-degree nodes, while in the context of big data, the scale of triples under the same predicate p might be extremely large. Therefore, we specify these nodes and separate them into smaller units, which is also called the subject/object refinement. However, for a triple in RPQs, we do not know s or o in most cases, as a consequence, POS_{po} and PSO_{ps} do not seem to work. For example, we want to access all objects in PSO_{ps}, but we get nothing if only p is specified.

To compensate for this defect, we define PS_p and PO_p to co-work with POS_{po} and PSO_{ps}. As a matter of fact, DB-Index is constructed on the basis of the formulas in Definition 3, in which we regard POS_{po} and PSO_{ps} as primary indexes, PS_p and PO_p as secondary indexes. The structure of DB-Index is shown in Fig. 1, of which the space complexity is $O(|T|)$, and $|T|$ is the size of the RDF graph T. We separate RPQs into atomic units, as mentioned before, which is actually the subset of triple pattern query of the form $(?s|s, ?p|p, ?o|o)$. However, the predicates of RPQs are never variables, as a consequence, there are four possibilities for a primitive edge of RPQs $(?s|s, p, ?o|o)$, all of which DB-Index

covers. For example, $(s, p, ?o)$ can be obtained by PSO_{ps}, $(?s, p, o)$ by POS_{po}, and $(?s, p, ?o)$ by PSO_{ps} with PS_p. There is no need to query (s, p, o), since we have already obtained all terms of this triple. However, RPQs are much more complex than triple pattern queries, and later in Section 5, we will present the algorithms.

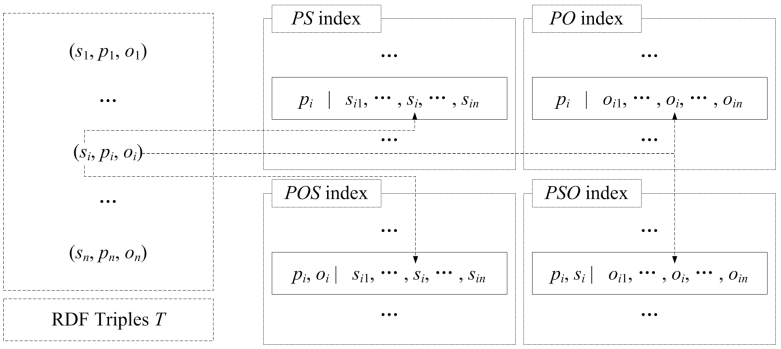

Fig. 1. The structure and construction of DB-Index

4.2 Index Construction

In order to construct DB-Index, for each triple $(s, p, o) \in T$, s is inserted into POS_{po} and PS_p, o into PSO_{ps} and PO_p, as demonstrated in Fig. 1 and Algorithm 1. If s already exists in POS_{po} and PS_p, we skip this step, the same as o. The complexity of Algorithm 1 is $O(|T|)$, in which $|T|$ is the size of the RDF graph T. As shown in Fig. 1, we ensure the indexes are ordered by inserting elements into the appropriate positions, which are to be used in the sort-merge join operations.

Algorithm 1. Constructing DB-Index

Input: An RDF graph T
Output: DB-Index

```
1. constructIndex(T)
2. for each triple (s, p, o) ∈ T do
3.    if s does not exist in PS_p and POS_po then
4.       Insert s into PS_p and POS_po;
5.    end if
6.    if o does not exist in PO_p and PSO_ps then
7.       Insert o into PO_p and PSO_ps;
8.    end if
9. end for
10. return
```

5 TraPath Algorithms

In this section, we introduce a series of maintenance algorithms, called TraPath, which are based upon DB-Index, which is composed of three parts, the traversal-based search algorithm, the parallel evaluation algorithm and the scheduling algorithm. First, we present the traversal-based search algorithm, which is the core algorithm of this paper. Then, we introduce the parallel evaluation algorithm that accelerates our query. Finally, we describe the scheduling algorithm. The following theorems lay the theoretical foundation for the TraPath algorithms.

Theorem 1. *For two arbitrary edges $p_1, p_2 \in P$. There exists a path on them, if $PO_{p_1} \cap PS_{p_2} \neq \emptyset$.*

Proof. There exists a path on two arbitrary edges, iff there exist a common vertex between them, i.e., the union of out-degree objects of p_1 and in-degree subjects of p_2 is not an empty set.

Theorem 2. *For two arbitrary edges $p_1, p_2 \in P$, the in-degree of p_1 is specified with s. There exists a path on them, if $\exists e \in PSO_{p_1 s'}$, s.t $PSO_{p_2 e} \neq \emptyset$.*

Proof. If $PSO_{p_2 e} \neq \emptyset$, then $\exists (e, p_2, o') \in T$, $e \in PS_{p_2}$, we know $PSO_{p_1 s'} \subseteq PO_{p_1}$, and $e \in PSO_{p_1 s'}$, then $e \in PO_{p_1}$ obviously. $PO_{p_1} \cap PS_{p_2} \neq \emptyset$, according to Theorem 1, we have a path on p_1 and p_2.

The traversal-based search is devised on the basis of Theorem 2, and Theorem 1 plays an important role in the parallel evaluation algorithm.

5.1 Traversal Based Search

The traversal based search algorithm (abbreviated as TBS) takes advantage of the depth-first traversal on the basis of Theorem 2, in which nested-loops join is used to bridge various edges. For each step of TBS of a forward search, two terms of the triple are known of which the form is $(s, p, ?o)$. s is obtained in different ways, from the previous step, PS_p, or specified by users. We access objects from PSO_{ps}, and then push an o into the next step, in which o is treated as s', and $PSO_{p's'}$ is invoked again with the next predicate p'. We save the intermediate state, and push a next o if nothing obtained from $PSO_{p's'}$. The algorithm prints a result when a path is found, and exits when all elements of PSO_{ps} have been traversed, in which p is the label of the first edge of RPQs.

The pseudo code of the traversal based search is shown in Algorithm 2, in which we traverse recursively on an RDF graph. The stacks are chosen to store regular path expressions due to their belonging to sequential sets. Line 4-8 makes it possible for us to search bi-directionally. For each step, the algorithm traverses once at least, and $|list|$ times in the worse case. If we traverse $|r|$ steps, then the complexity of Algorithm 1 is $O(n^{|r|})$, where n is the max size of PSO_{ps} or POS_{ps}.

Sort-merge joins can be also taken into account in the TBS. For example, there are two sequential predicates p_1 and p_2, as described in Theorem 1, PO_{p_1}

Algorithm 2. Traversal based search

Input: $StackC$ with sequential predicates, $StackP \leftarrow \emptyset$, and start node s
Output: Path matching the given regular expression

```
1. traversalBasedSearch(StackC, StackP, s)
2. p ← pop(StackC);
3. push(StackP, p)
4. if flag is forward then
5.    list ← PSO_ps;
6. else
7.    list ← POS_ps;
8. end if;
9. if list is not empty then
10.   for each element e ∈ list do
11.      if C is ∅ then
12.         Put the result into the queue Queue;
13.         Push(StackC, pop(StackP));
14.         return
15.      else
16.         traversalBasedSearch(StackC, StackP, e);
17.      end if
18.   end for
19. end if
20. push(StackC, pop(StackP));
21. return
```

is joined with PS_{p_2}. This the complexity of TBS is $O(m+n)$, which is better than the nested-loop join's $O(m \cdot n)$. However, for a predicate p in a large-scale RDF graph T, the size of PS_p appears to be extremely large. All of the terms in both PO_{p_1} and PS_{p_2} need to be loaded into the memory, which exhibits a inferior performance. Our investigation and analysis of the real-world RDF graph, which will be introduced in detail in Section 6, indicates $|PSO_{ps}| \ll |PS_p|$ for the same predicate p and arbitrary s. As a consequence, for both methods of constituting a path as represented in Theorem 1 and 2, nested-loops join achieves superior performance than sort-merge join on large-scale RDF graphs, which prompts us to choose the nested-loops join as our basic method.

5.2 Parallel Evaluation

The parallel evaluation algorithm parallelizes TBS to expedite the execution by separating paths. In the parallel evaluation process, there are several threads to process sub-paths invoking the TBS algorithm, cooperating with the master thread to collect partial results and bridge them together. For each TBS, counting paths are printed into the result queues, nevertheless, we search for an existing path in the master. Algorithm 3 and 4 gives the pseudo code of our parallelization strategies. Each *subPathProcessor* processes a sub-path on the basis of TBS, the *pathGenerator* gathers partial results and generates the

path matching the regular expression r. The complexity of *pathGenerator* is $O(m \cdot n)$, in which m is the max size of $Queue[i]$. The *pathGenerator* achieves superior performance, on one hand, the max $|Queue[i]|$ could be quite small if the path is divided appropriately, on the other hand, *pathGenerator* starts with *subPathProcessors* simultaneously, which means they work in parallel.

Algorithm 3. Parallel producer

Input: $StackC$ with sequential predicates, $StackP \leftarrow \emptyset$
Output: Partial results $Queue$, $n \geq 2$

1. **subPathProcessor**($StackC$, $StackP$)
2. $p \leftarrow$ get($StackC$);
3. **if** $flag$ is $forward$ **then**
4. $list \leftarrow PS_p$;
5. **else**
6. $list \leftarrow PO_p$;
7. **end if**;
8. **if** $list$ is not empty **then**
9. **for** each element $e \in list$ **do**
10. **traversalBasedSearch**($StackC$, $StackP$, e);
11. **end for**
12. **end if**
13. **return**

Algorithm 4. Parallel comsumer

Input: partial results $Queue$, $n \geq 2$
Output: Path matching the given regular expression

1. **pathGenerator**($StackQ$)
2. **for** $Queue[i] \in Queue$ **do**
3. $Path[i] \leftarrow$ pop($Queue[i]$);
4. **end for**
5. **while** $true$ **do**
6. **if** a path $Path$ is found **then**
7. **return** $Path$;
8. **end if**
9. find the smallest $Path[i] \in Path$;
10. $Path[i] \leftarrow$ pop($Queue[i]$);
11. **end while**
12. **return**

Here we have a *trade-off* between parallel algorithms with higher complexity and serial algorithms with lower complexity. We choose the former since parallel algorithms cannot only withstand the pressure of large-scale data, but also have a good scalability, of which we could take advantage to expand our index and algorithm schemes.

5.3 Scheduling

The parallel evaluation algorithm only works when no starting or ending node is initialized, of which the query expression is $(?x, r, ?y)$. It is worth mentioning that, if s or o is specified by users, it is more efficient to invoke the TBS algorithm directly. Therefore, the algorithms need to be scheduled in accordance to the form of the query syntax. The pseudo code of the scheduling algorithm is shown in Algorithm 5, which schedules different algorithms under various conditions.

Algorithm 5. Scheduling

Input: Query $Q = (?x|s, r, ?y|o)$
Output: Path matching given regular expression

```
1.  queryScheduling(Q)
2.  if Q ∈ (?x, r, ?y) then
3.     Separate r into sub-paths SubPaths
4.     for each sub-path sp ∈ SubPaths do
5.        Push each edge of sp into StackC; StackP ← ∅;
6.        subPathProcessor(StackC, StackP);
7.     end for
8.  else if Q ∈ (?x, r, o) then
9.     flag ← inverse;
10.    Push each edge of r into StackC; StackP ← ∅;
11.    traversalBasedSearch(StackC, StackP, o);
12. else
13.    flag ← forward;
14.    Push each edge of r into StackC; StackP ← ∅;
15.    traversalBasedSearch(StackC, StackP, s);
16. end if;
```

5.4 Alternation and Kleene Star

The alternation and Kleene star operators with the existing path semantics can be both transformed to the basic paths of which the regular expressions are of the forms $r = p \mid (r/r)$. We search an alternation path by permutation and combination. For example, for a query $Q = (?s, (p_1|p_2)/p_3, ?o)$, the path p_1/p_3 and p_2/p_3 are both taken into account. For a path contains a Kleene star, the search begins with an empty path of the Kleene star, extends the path by repeating predicates, and ends with either a path obtained or no path found with a circle formed.

6 Experiments

In this section, we carry out a series of experiments to evaluate our index structures and algorithms. All experiments are conducted on a Dell OptiPlex 990 PC with a 3.10 GHz Intel i5-2400 quad-core CPU. We use Cassandra 1.1.6 as our underlying repository, and implement the algorithms with Java.

6.1 Storage Performance

We estimate the storage performance from two aspects including execution time of data loading and size of index structures, and use Lehigh University Benchmark (LUBM) as the datasets of our storage performance experiments.

The Sesame repository can be deployed only in a stand-alone environment, for the sake of fairness, we load data in a single node to compare with Sesame. As shown in Fig. 2, our approach and Sesame are equally matched in terms of loading performance. Fig. 3 displays the size of our index structure and Sesame. The space of primary indexes is larger relatively, which carries most of the information of paths. As speculated, the storage performance experiments indicates that DB-Index has a linear space complexity.

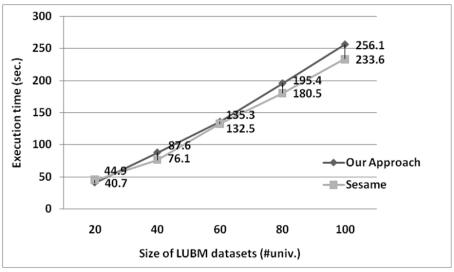

Fig. 2. Load time

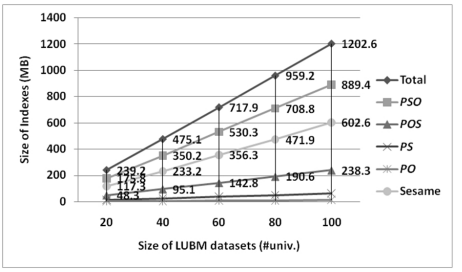

Fig. 3. Index size

6.2 Query Performance

We use DBpedia 3.6 as the dataset for query performance experiments since it is a real-world RDF data extracted from Wikipedia. For the in-depth understanding of the internal structure of DBpedia, we have analysed the experimental dataset, and the in/out-degree size distribution of predicates is demonstrated in Fig. 4. As is shown in the figure, the sizes of primary indexes mostly distribute in the range of e^0 to e^2. It means almost all of POS_{po} and PSO_{ps} have a small size, which verifies our assumption. If we describe an instance with RDF triples as the form $(s, p, ?o)$, there will not be too many objects. For example, if we have a query $(Aristotle, name, ?o)$, and we can only obtain one object, since Aristotle has only one name. It is similar that we change the predicate as long as the subject is specified. As opposed to the primary indexes, the sizes of secondary indexes distribute more evenly, from e^0 to e^{12}, which signifies that a lot of PS_p and PO_p have a large scale. For the above example, if we replace $Aristotle$ with a variable, then the query becomes $(?s, name, ?o)$, and PS_{name} is extremely large, since almost all of the instances in DBpedia have the predicate $name$.

We have devised several RPQ test cases, among which Q_2 is generated by random walking, Q_3 by predicates random selecting, and the rest by artificial design. As shown in Table 1, Q_2, Q_4 and Q_6 have the specified in/out-degree

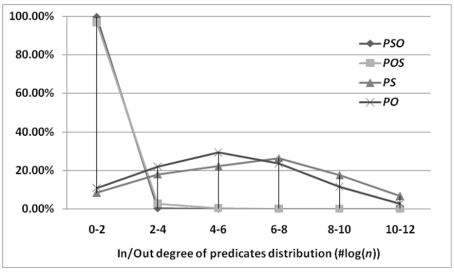

Fig. 4. Distribution of DB-Index size

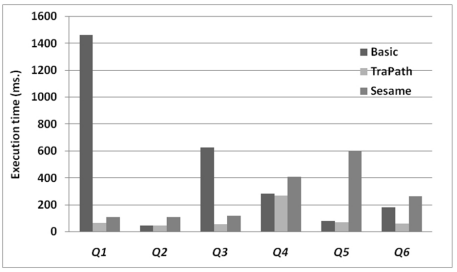

Fig. 5. Query performance experiments

Table 1. RPQ test cases

$\#Q$	RPQ test cases
Q_1	$(?x, country/largestCity/name, ?y)$
Q_2	$(Aristotle, influenced/beatifiedPlace/leaderName/birthPlace/name, ?y)$
Q_3	$(?x, purpose/(birthPlace\|deathPlace)/honours/largestCity/birthDate, ?y)$
Q_4	$(Alabama\_Army\_National\_Guard, (country/largestCity)^{100}/name, ?y)$
Q_5	$(?x, (lowestPlace/depth)^{100}, ?y)$
Q_6	$(?x, (govenment/govenmentElevation)^{100}/area, Kentucky)$

node, Q_4-Q_6 belong to long paths. As mentioned before, the Sesame repository provides weak support for RPQs, for the fairness of the evaluations, we apply long paths instead of Kleene star.

We have implemented TraPath based upon DB-Index, in which there are two *subPathProcessor*s. For the purpose of comparison, we have removed the parallel algorithms so that there is only a serial algorithm with TBS. Besides, we have deployed Sesame 2.6.0 with its own native repository in our experimental environment. The experimental results are shown in Fig. 5. As shown in the figure, our parallel approach exhibits better performance than Sesame, especially for longer RPQ paths.

7 Conclusion

In this paper, we devise a double-layer bi-directional index structure, called DB-Index, which covers all possible forms of RPQs, and propose a novel traversal-based algorithm, named TraPath, which achieves the efficient execution of RPQs on large-scale RDF graphs by using the DB-Index. The experiment results show that DB-Index has a linear complexity and TraPath outperforms the state-of-the-art methods.

Acknowledgments. This work is supported by the National Natural Science Foundation of China (Grant No. 61100049, 61373165) and the National High-tech R&D Program of China (863 Program) (Grant No. 2013AA013204).

References

1. Klyne, G., Carroll, J.J., McBride, B.: RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation (2014)
2. Jupp, S., Malone, J., Bolleman, J., et al.: The EBI RDF platform: Linked Open Data for the Life Sciences. Bioinformatics (2014)
3. Breslin, J., Decker, S.: The Future of Social Networks on the Internet: the Need for Semantics. IEEE Internet Computing 11(6), 86–90 (2007)
4. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Recommendation (2013)
5. Mendelzon, A.O., Wood, P.T.: Finding Regular Simple Paths in Graph Databases. SIAM Journal of Computing 24(6), 1235–1258 (1995)
6. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data - The Story So Far. International Journal on Semantic Web and Information Systems 5(3), 1–22 (2009)
7. Neumann, T., Weikum, G.: RDF-3X: A RISC-Style Engine for RDF. In: Proceedings of the VLDB Endowment, vol. 1(1), pp. 647–659 (2008)
8. Ladwig, G., Harth, A.: CumulusRDF: Linked Data Management on Nested Key-Value Stores. In: 7th International Workshop on Scalable Semantic Web Knowledge Base Systems, pp. 30–42 (2011)
9. Wang, X., Jiang, L., Shi, H., Feng, Z., Du, P.: Jingwei+: A distributed large-scale RDF data server. In: Sheng, Q.Z., Wang, G., Jensen, C.S., Xu, G. (eds.) APWeb 2012. LNCS, vol. 7235, pp. 779–783. Springer, Heidelberg (2012)
10. Alkhateeb, F., Baget, J.F., Euzenat, J.: Extending SPARQL with Regular Expression Patterns (for Querying RDF). Journal of Web Semantics 7(2), 57–73 (2009)
11. Kochut, K.J., Janik, M.: SPARQLeR: Extended SPARQL for Semantic Association Discovery. In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007. LNCS, vol. 4519, pp. 145–159. Springer, Heidelberg (2007)
12. Koschmieder, A., Leser, U.: Regular Path Queries on Large Graphs. In: Ailamaki, A., Bowers, S. (eds.) SSDBM 2012. LNCS, vol. 7338, pp. 177–194. Springer, Heidelberg (2012)