Data Science and Engineering (2019) 4:24–43
https://doi.org/10.1007/s41019-019-0090-z

Efficient Subgraph Matching on Large RDF Graphs Using MapReduce

Xin Wang[1,2] · Lele Chai[1] · Qiang Xu[1] · Yajun Yang[1,2] · Jianxin Li[3] · Junhu Wang[4] · Yunpeng Chai[5]

Received: 15 January 2019 / Revised: 20 March 2019 / Accepted: 23 March 2019 / Published online: 4 April 2019

Abstract
With the popularity of knowledge graphs growing rapidly, large amounts of RDF graphs have been released, which raises the need for addressing the challenge of distributed subgraph matching queries. In this paper, we propose an efficient distributed method to answer subgraph matching queries on big RDF graphs using MapReduce. In our method, query graphs are decomposed into a set of *stars* that utilize the semantic and structural information embedded RDF graphs as heuristics. Two optimization techniques are proposed to further improve the efficiency of our algorithms. One algorithm, called *RDF property filtering*, filters out invalid input data to reduce intermediate results; the other is to improve the query performance by postponing the Cartesian product operations. The extensive experiments on both synthetic and real-world datasets show that our method outperforms the close competitors S2X and SHARD by an order of magnitude on average.

Keywords Star decomposition · Subgraph matching · MapReduce · RDF graphs

Abbreviations

CQ	Conjunctive query
BGP	Basic graph patterns
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
HDFS	Hadoop Distributed File System
LUBM	Lehigh University Benchmark
WatDiv	Waterloo SPARQL Diversity Test Suite

1 Introduction

More than one decade ago, the Semantic Web was proposed by Berners-Lee et al. [3], which now has become a series of W3C standards[1] in order to realize the machine understandable World Wide Web. The semantic links among resources on the traditional Web can be explicitly represented on the Semantic Web. In the meanwhile, the graph data model has been more and more popular to manage graph and network data in various domains. Compared with the relational model, the graph model can more naturally characterize relationships among entities in the real world. In particular, the *Resource Description Framework* (RDF) [16] is a mainstream graph model, which has become the de-facto standard for representing and exchanging data on the Semantic Web. In recent years, with the campaign of the Linked Open Data [4] initiative, the scale of RDF graph data has grown exponentially. Hence, it is essential to develop efficient storage and query mechanism for large-scale RDF graphs.

The Resource Description Framework, a graph-based data model, is commonly used to represent and organize resources in *knowledge graphs* because of its flexibility. An RDF data are a collection of triples (s, p, o), each of which represents a statement of a predicate p between a subject s and an object o. An RDF triple can be naturally viewed as an edge with s and o as vertices. Thus, an RDF graph can be represented as a labeled directed graph, e.g., the example RDF graph G_1 excerpted from DBpedia dataset in Fig. 1. It describes some information about philosophers. Due to the flexibility of RDF data, they are widely applied in various fields, such as science, bioinformatics, business intelligence,

✉ Yajun Yang
yjyang@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, Melbourne, Australia

4 School of Information and Communication Technology, Griffith University, Gold Coast, Australia

5 School of Information, Renmin University of China, Beijing, China

1 https://www.w3.org/standards/semanticweb/.

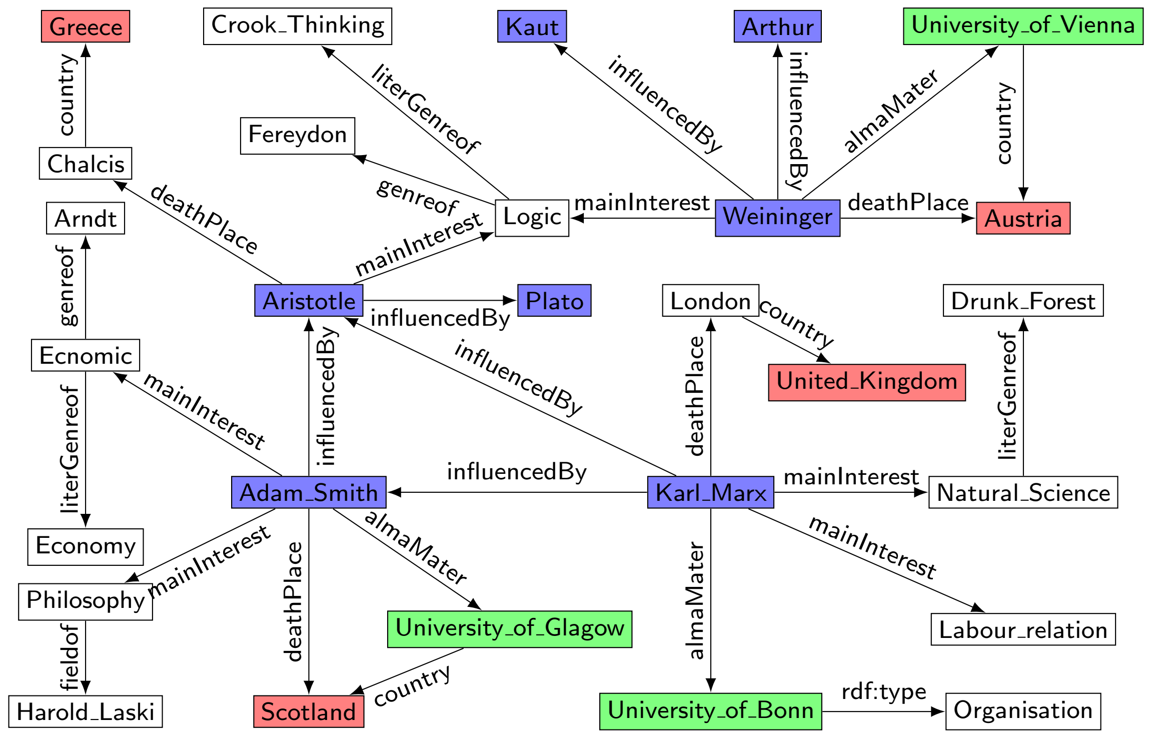


Fig. 1 An example RDF graph G_1 excerpted from DBpedia dataset

and social networks [12]. In real world, the size of RDF data often reaches hundreds of millions of triples.

Subgraph matching is widely considered as one of the fundamental mechanisms for querying large-scale graph data. SPARQL is the standard query language for RDF graphs endorsed by W3C [5], in which basic graph patterns (BGP) are realization of subgraph matching. Theoretically, the semantics of SPARQL BGP is equivalent to the problem of subgraph homomorphism [18], whose evaluation complexity is known to be NP-complete [7]. Therefore, how to efficiently answer subgraph matching queries (i.e., BGP) over big RDF graphs has been broadly recognized as a challenging problem.

Subgraph matching aims at finding all the satisfying matching subgraphs over the large data graph. More specifically, given a data graph, i.e., an RDF graph G and a query graph Q, subgraph matching will fetch all the subgraphs over G that satisfying all the triples contained in Q, which is a *conjunctive query* (CQ) on G. For instance, the following *CQ* Q_1 consists of two triple patterns over G_1.

$$Q_1(?x, ?y) \leftarrow (\texttt{Karl\_Marx}, \texttt{influenceBy}, ?x) \land (\texttt{Karl\_Marx}, \texttt{mainInterest}, ?y) \quad (1)$$

Currently, there has been some research works on subgraph matching queries over RDF data in a ***distributed*** environment. One category of methods is based on the relational schema [8, 11, 14, 19, 22], in which RDF data are modeled as a set of triples and stored in relational tables or a variant relational schema. All of these methods do not consider inherent graph-like structures of RDF data. When processing complex subgraph matching queries, excessive join operations over relational tables are needed, which may incur expensive cost. In contrast, the other category of methods manages RDF data in native graph formats [17, 20, 28] and represents subgraph matching queries as query graphs, which typically employs adjacency lists to store RDF data. Thus, for a subgraph matching problem, how to reduce the enormous intermediate results is crucial.

In [24], query graphs are decomposed into stars (trees of depth 1). Lai et al. pointed out that the star-join algorithm in [24] suffers from scalability problems due to the generation of a large number of matches when evaluating a star with multiple edges [15]. The reason for this issue is that in unlabeled, undirected graphs, they focused on it is very likely that the large combination of intermediate results is generated due to the lack of distinguishable information on vertices and edges. Thus, they proposed the so-called TwinTwigJoin MapReduce [6] algorithm, where a TwinTwig is either a single edge or two incident edges of a vertex. Unlike unlabeled and undirected graphs in [15, 24], RDF graphs have URIs as the unique vertex labels and directed edges. Thus, the problem concerned in [15] does not exist over RDF graphs. Therefore, it is reasonably safe to exploit the more holistic star-shaped structures other than just twin twigs as decomposition units of query graphs to minimize the amount of intermediate results.

To this end, we propose a new star-based query decomposition strategy, in which the star retains more holistic graph structures of query graphs than the TwinTwig method [15]. Thus, our approach can be completed in fewer MapReduce rounds. In our method, in order to evaluate subgraph matching queries more efficiently, query graphs are decomposed into a set of stars by using the semantic and structural information embedded in RDF graph as heuristics (i.e., h values defined in this paper), to evaluate subgraph matching queries in MapReduce. In addition, in order to reduce the intermediate results, the matching order of stars is determined by a greedy strategy.

Our main contributions include: (1) we propose an efficient and scalable distributed algorithm based on star decomposition, called **StarMR**, for answering subgraph matching queries on RDF graphs; (2) two optimization strategies of **StarMR** are devised, one of which employing the properties in RDF graphs to filter out invalid input data in MapReduce iterations, the other postponing part of Cartesian product operations to the final step of MapReduce to reduce a part of unpromising Cartesian product operations; and (3) extensive experiments on both synthetic and real-world RDF graphs have been conducted to verify the efficiency and scalability of our method. On average, the experimental results show that **StarMR** outperforms the state-of-the-art method by an order of magnitude.

The rest of this paper is organized as follows. Section 2 briefly reviews related work. In Sect. 3, we introduce preliminary definitions on RDF graphs and subgraph matching queries. In Sect. 4, we describe in detail how to decompose *CQ* queries, determine the matching order of stars, and match *CQ* queries using MapReduce. We then present two optimization strategies in Sect. 5. Section 6 shows our extensive experimental results, and we conclude in Sect. 7.

2 Related Work

The existing research work on ***distributed/parallel*** SPARQL queries over large-scale RDF graphs can be classified as follows:

Relational Schema Approach In the context of the urgent need for Web-scale distributed query systems [29], SHARD [19] is designed and developed using the MapReduce framework to address the scalability limitation issue. In terms of data persistence, the metadata of the system is persisted in the Hadoop Distributed File System [23]. In that case, the query graph is decomposed into the triple sets. More specifically, SHARD handles SPARQL queries over RDF data for triple stores which need to iterate over query statements to bind variables to vertices in data graphs while satisfying all of the query constraints. Meanwhile, to accelerate processing the subsequent similar queries, certain relevant intermediate results might not be removed immediately. Each round of MapReduce only adds one query clause with the join operation in [19]. Although SHARD has a significant improvement in enhancing the datasets scalability with the aid of Hadoop, due to no plans for query processing, a large number of Hadoop jobs are required to execute the whole procedure.

Similarly, HadoopRDF [14] features efficiency and scalability in managing large amounts of RDF data. For the data stored in the Hadoop cluster, the framework utilizes a schema to convert various format RDF data to N-triples. The standard data conversion can bring great benefits for the later processing. Moreover, HadoopRDF divides RDF triples based on the predicates into multiple smaller files. In this way, for a user query, if the predicate position is not a variable, the corresponding file can be matched directly; otherwise, because the predicate is a variable, HadoopRDF cannot make sure which type of the object belongs to. To avoid searching all the files, another file organization category named object splitting is exploited. The object splitting method further classifies files according to the object type. Meanwhile, by combining the predicate and object splitting approach, the query processing can speed up. Specifically, the query retrieval involves three phases. First, regarding the subgraph matching query clause as the input and passing it to the first component named input selector. Second, making use of the proposed greedy algorithm to guarantee the generated query plan as the optimal one. Finally, joining the relevant intermediate items together and feeding the final results back to the user. Moreover, a triple pattern in SPARQL queries cannot simultaneously take part in more than one join in a single Hadoop job by using the MapReduce framework.

The abovementioned two methods do not employ any structural information of query graphs, thus a large number of join operations may incur expensive costs. Furthermore, Virtuoso [8], supporting RDF in a native RDBMS, also model RDF data as a set of triples. TriAD [11], using a custom MPI protocol, employs six SPO permutation indexes, partitions RDF triples into those indexes, and uses a locality-based summary graph to speed up queries. Many current RDF query approaches are extremely dependent on the query pattern shapes, i.e., for certain query pattern shapes, the query processing can execute quite well. While the query performance drops for other query shapes. Hence, Schätzle et al. [22] proposed a relatively efficient query processing system named S2RDF, which does not depend on the query pattern shapes anymore. In addition, this approach extends the vertical partitioning [1] methods and Join Indices [25] to preprocess the original RDF data. More specifically, S2RDF introduces the relational partitioning model ExtVP to store

RDF data over the Spark parallel framework, by which it can effectively minimize the query input size. Nevertheless, as for modification operations, the deletion operation of the triples might result in a decline in query performance and stability. In addition, the cost of the semi-join preprocessing in [22] is prohibitively expensive.

Native Graph Approach The star-decomposition-based searching methods proposed by Yang et al. [26] is about approximate matching, which is devoted to $top-k$ star query. In [26], the query decomposition phase is to decompose the subgraph query to a set of star queries, and for each decomposed star query, the matches in decreasing order of matching score and the best match can be picked out. In [28], RDF data are modeled in its native graph form, a key-value store which saves node identifiers as the keys, and the adjacency lists of nodes as the values. Trinity.RDF [28] leverages graph exploration to reduce the volume of intermediate results, while the final results need to be enumerated at the single master node using a single thread. S2X [20] builds on GraphX [9], a distributed graph processing framework in top of Spark [27], to implement query graph matching of SPARQL. In S2X, a query graph is also decomposed into triple patterns which is similar to the methods in [14, 19]. All of these triple patterns are matched first; then, intermediate results are gradually discarded by iterative computation; finally, the remaining matching results are joined, which may lead to potentially large intermediate results. In addition, Peng et al. adopt a partial evaluation and assembly framework to perform SPARQL queries based on gStore [30], a graph-based SPARQL query engine using VS*-tree indexes [17]. In their method, each slave machine evaluates the query in the partial computation phase, and then, in the assembly phase, a large number of local partial matches are sent to the coordinator and joined together to obtain the final results, which may become a performance bottleneck when the amount of partial matches are large.

Distributed Systems In the era of Big Data, the distributed/parallel technique has become an indispensable tool for large-scale knowledge data management. In recent years, plentiful distributed systems and frameworks for large-scale graph data have been proposed. For instance, YARS2 [13] is a representative knowledge graph managing system based on MapReduce. YARS2 is a distributed semantic web search engine, which integrates data retrieving, collecting, indexing, and browsing together. It plays a pivotal role in managing large-scale graph data models and enabling interactive query answering. The system consists of several components: crawler, indexer, object consolidator, index manager, query processor, ranker, and user interface. The crawler is a pipelined architecture for crawling diverse source data into a uniform schema, and the indexer is a general framework for managing keyword indices and statement indices; then, the query processor will generate the optimal query plan for answering the queries. Then, the corresponding results are retrieved to users in the descending orders. Another research work Sempala [21], which is an RDF graph data query engine based on distributed SQL-on-Hadoop database Impala and Parquet, distributed file format, which provides interactive-time SPARQL query processing efficiently. In addition, Lai et al. proposed a MapReduce-based distributed efficient subgraph enumeration algorithm based on TwinTwig structure decomposition, but the algorithm is only used for undirected unlabeled graphs.

In this paper, we focus on the analytical processing scenario of RDF graphs using MapReduce which does not take advantage of any prebuilt indexes. Though building indexes can definitely accelerate lookups with high selectivity, it will not benefit analytical processing in which almost all data are accessed. So, it is unfair to compare our approach with those based on intensive indexes, such as S2RDF [22], the distributed gStore system [17]. In our method, (1) we store RDF triples using the adjacency list scheme; (2) a star-decomposition strategy with heuristic information is proposed, which is able to keep more holistic structures of query graphs; (3) as to optimization strategies, we employ RDF properties to filter out unpromising input data and postpone Cartesian product operations.

3 Preliminaries

In this section, we introduce several basic background definitions about RDF graphs and subgraph matching queries which are used in our algorithms.

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*, and o is the *object*. A finite set of RDF triples is called an *RDF graph*.

Given an RDF graph G, let V, E, Σ denote the set of vertices, edges, and edge labels, respectively. Formally, $V = \{s \mid (s, p, o) \in G\} \cup \{o \mid (s, p, o) \in G\}$, $E \subseteq V \times V$, and $\Sigma = \{p \mid (s, p, o) \in G\}$. The function **lab**: $E \rightarrow \Sigma$ returns the labels of edges in G.

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

Let $V(Q)$ and $E(Q)$ be the set of vertices and edges in G_Q, respectively. For each vertex $u \in V(Q)$, if $u \in Var$, then u can match any vertex $v \in V$; otherwise, u only can match the vertex $v \in V$ whose label is the same as that of u.

Definition 3 *(Subgraph matching)* The semantics of a *CQ* Q over an RDF graph G is defined as: (1) μ is a mapping from vertices in $\bar{x}$ and $\bar{y}$ to vertices in V, where $\bar{x} = (x_1, \dots, x_m)$, $\bar{y} = (y_1, \dots, y_m)$; (2) $(G, \mu) \vDash Q$ iff $(\mu(x_i), \mu(a_i), \mu(y_i)) \in E$ and the labels of x_i, a_i and y_i are the same as that of $\mu(x_i)$, $\mu(a_i)$ and $\mu(y_i)$, respectively, if x_i, a_i, $y_i \notin Var$; and (3) $\Omega(Q)$ is the set of $\mu(\bar{z})$, where $(\bar{z}) = (z_1, \dots, z_n)$, such that $(G, \mu) \vDash Q$. Ω_Q is the answer set of the subgraph matching query G_Q over G.

Some definitions about *mapping*s are needed. Two mappings μ_1 and μ_2 are called *compatible* denoted as $\mu_1 \sim \mu_2$, iff every element $v \in dom(\mu_1) \cap dom(\mu_2)$ satisfies $\mu_1(v) = \mu_2(v)$, where $dom(\mu_i)$ is the domain of μ_i. Furthermore, the set union of two compatible mappings, i.e., $\mu_1 \cup \mu_2$, is also a mapping.

4 The StarMR Algorithm

In this section, the distributed adjacency list storage strategy for an RDF graph will be first introduced. Then, we present how to decompose the query graph into a set of stars and determine the matching order of these stars. Finally, we describe in detail how to implement the subgraph matching query using MapReduce in a left-deep-join framework.

4.1 Storage Schema

In this paper, the RDF graph G is stored in a distributed adjacency list. For each vertex $v \in V$, we use $N(v)$ to denote the neighbor information of vertex v, where $N(v) = \{(p_i, v'_i) \mid (v, p_i, v'_i) \in G\}$. For example, the adjacency list storage schema of the RDF example graph G_1 is given in Table 1.

Taking the RDF graph G_1 in Sect. 1 as an example, all the vertices appeared in the subject positions are stored in the first column, all the neighbor vertices of each subject vertex are stored in the set of $N(v)$. For example, the entity London has one outgoing edges, and its neighbor set is { ⟨ country,United_Kingdom⟩ }.

Table 1 The adjacency list of RDF graph G_1

v	$N(v)$
Karl_Marx	{⟨mainInterest,Natural_Science⟩, ⟨mainInterest,Labour_relation⟩, ⟨influencedBy,Aristotle⟩, ⟨almaMater,University_of_Bonn⟩, ⟨influencedBy,Adam_Smith⟩, ⟨deathPlace,London⟩ }
...	...
London	{ ⟨country,United_Kingdom⟩ }

4.2 Star Matching

In this paper, the minimum matching unit is a star in our method, and the RDF graph G is stored in adjacency lists. Next, we give the definition of star.

Definition 4 *(Star)* A star is a tree of height one, denoted by $T = (r, L)$, where (1) r is the root of T; and (2) L is a set of 2 tuples (p_i, l_i), i.e., l_i is a leaf of T and (r, p_i, l_i) is an edge from r to l_i. Let $V(T)$ and $E(T)$ be the set of nodes and edges in T, respectively.

When matching a star T on the adjacency list of RDF graph, if the root vertex of star T can be well matched on one of the subject vertices of the adjacency list first, the matching process will not be terminated. Then, the star T will continue matching the leaf vertices l_i on $N(v)$ and once matched, we can obtain the sets of all the matching vertices, which are defined as the candidate sets $S(l_i)$ in this paper. Next, we will present the detailed process of a star matching on an adjacency list. And the star matching algorithm is listed as follows.

Algorithm 1: StarMatch$(T, N(v))$

```
  Input  : Star: $T = (r, L)$, where $L = \{(p_1, l_1), \dots, (p_t, l_t)\}$, $N(v), v \in V$
  Output: Matching results of $T$ over $N(v)$: $\Omega_v(T) = \{\mu_1, \mu_2, ..., \mu_n\}$
1 $\Omega_v(T) \leftarrow \emptyset$;
2 if $T.r$ matches vertex $v$ then
3     foreach $(p_i, l_i) \in T.L$ do            // the candidate set $S(l_i)$ of leaf $l_i$
4         if $p_i \notin Var$ then
5             $S(l_i) \leftarrow \{v' \mid (p_i, v') \in N(v) \wedge l_i$ matches $v'\}$ ;
6         else                                              // _ is a wildcard
7             $S(l_i) \leftarrow \{v' \mid (\_, v') \in N(v) \wedge l_i$ matches $v'\}$ ;
      // do Cartesian product operation $\{v\} \times S(l_1) \dots S(l_t)$ to get $\Omega_v(T)$
8     $\Omega_v(T) \leftarrow \{\mu \cup \mu_1 \dots \mu_t \mid \mu = \{(T.r, v)\} \wedge \mu_i = \{(l_i, v')\}, l_i \in T.L \wedge v' \in S(l_i)\}$;
9 return $\Omega_v(T)$;
```

Algorithm 1 will be run in the following steps: (1) first matches the root $T.r$ with the vertex v (line 2); (2) then obtains the candidate matching set of every leaf (lines 3–7); (3) next does the Cartesian product operations on the candidate matching sets of vertices in the star T to get matching results (line 8). Finally, StarMatch(T, $N(v)$) returns the matching results of the star T over $N(v)$ (line 9), and $\Omega(T)$ is the union of $\Omega_v(T)$, where $v \in V$.

4.3 Star Decomposition of Query Graphs

Before matching the subgraphs in an RDF graph, it is necessary to decompose the query graph into the minimum matching unit stars. Next, we give the definition of star decomposition and explain why the matching orders are crucial. In addition, we propose an effective approach to reduce the number of intermediate results, which leverages the user-defined heuristic information h value.

Definition 5 *(Star decomposition)* The *star decomposition* of a *CQ* $Q = \{tp_1, \dots, tp_n\}$ is denoted as $D = \{T_1, \dots, T_m\}$, where (1) T_i is a star; (2) $T_i.r \neq T_j.r, T_i, T_j \in D \wedge i \neq j$; (3) $E(T_i) \cap E(T_j) = \emptyset, T_i, T_j \in D \wedge i \neq j$; and (4) $\bigcup_{1 \le i \le m} E(T_i) = E(Q)$.

Example 1 Consider the example query Q_1 over the RDF graph G_1 in Sect. 1, the query graph G_{Q_1} of Q_1 is shown in Fig. 2. Moreover, D is the star decomposition of G_{Q_1} which contains three stars, T_1, T_2, and T_3. □

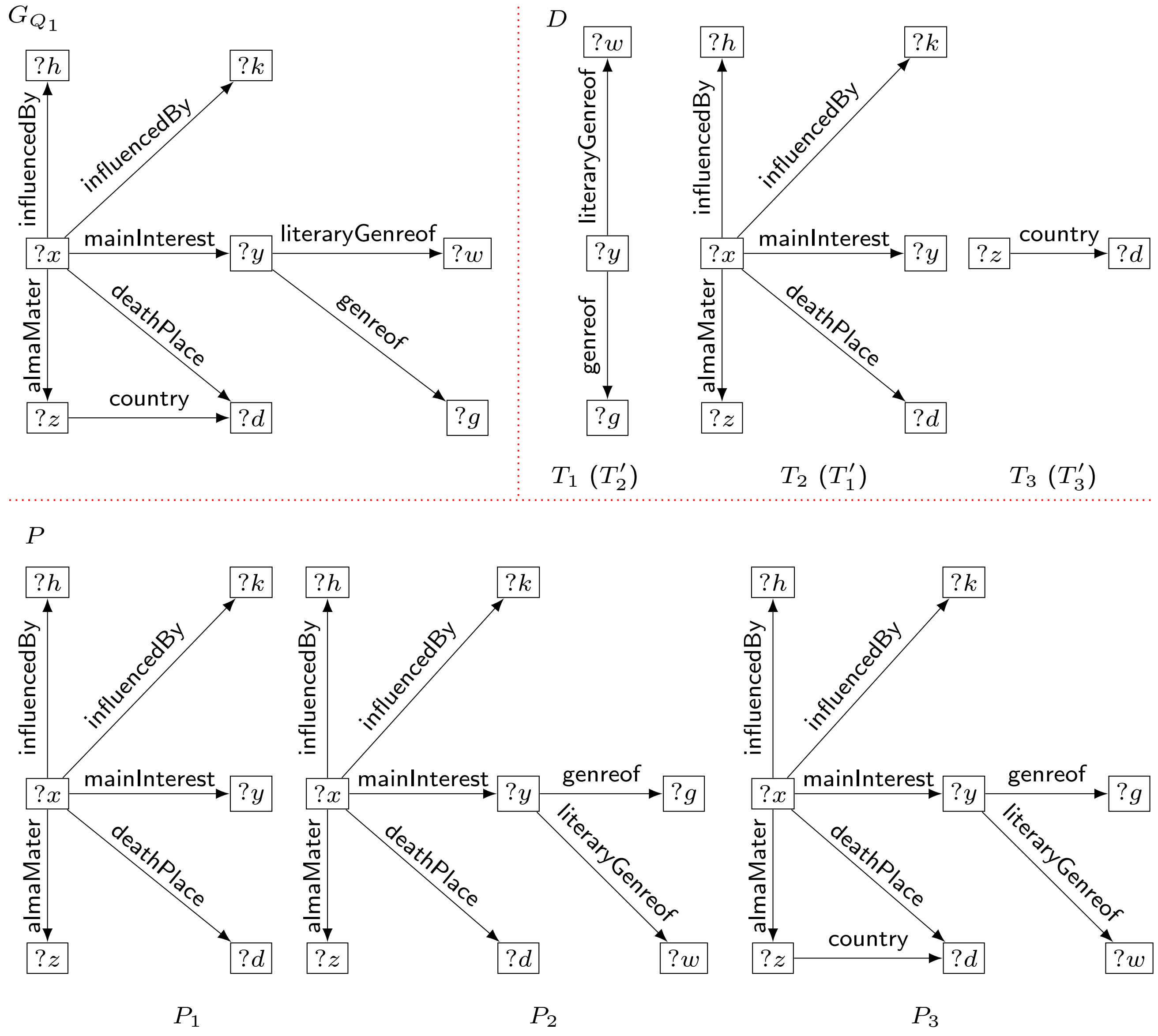


Fig. 2 The query graph and star decomposition of query Q_1

After obtaining the query decomposition D of Q_1, there exist six matching orders. According to Algorithm 1, stars T_1, T_2, and T_3 over G_1 have 2, 2, and 4 matching results, respectively. Consider the matching order $T_1T_3T_2$, there exists eight intermediate results by joining the matching results of star T_1 and T_3, because these two stars do not share any common vertex. However, another matching order $T_2T_1T_3$ only generates one intermediate result. In other words, the matching order of stars has a significant effect on the performance of queries.

We leverage the structure information and semantics in RDF graphs to decompose the query graph into stars and give a matching order to reduce the number of intermediate results using a greedy strategy. In particular, we define h value as the heuristic information. The function **fre**: $\Sigma \rightarrow \mathbb{N}$ gets the frequency of a predicate p in an RDF graph G, where $\mathbb{N}$ is the set of natural numbers and $\mathsf{fre}(p) = |\{(s_i, p, o_i) \mid (s_i, p, o_i) \in G\}|$. Then, for a query Q over G, let $P(u)$ be the set of properties (a.k.a., predicates) of vertex u in Q, i.e., $P(u) = \{p_i \mid (u, p_i, u'_i) \in Q\}$. The h value of each vertex $u \in V(Q)$ is defined as follows:

$$\mathsf{h}(u) = \frac{|outDeg|}{\min_{p \in P(u)} \mathsf{fre}(p)} \tag{2}$$

where $outDeg$ is the out degree of vertex u. The h value is determined by two factors: (1) the more out degrees a vertex u has, the more variables may be bound when the star rooted at u is matched; (2) the smaller $\mathsf{fre}(p), p \in P(u)$ is, the higher selectivity of vertex u has. If all properties of vertex u are variables, $\mathsf{h}(u) = 0$. Our star-decomposition algorithm guided by h values is shown in Algorithm 2.

In Algorithm 2, a constant vertex in Q_c having the maximum h value is selected as the root of the first star (lines 3–4). If Q_c is an empty set, the algorithm picks up a vertex in $Sub(Q)$ whose h value is the maximum (lines 4–5). The star rooted at the selected vertex is generated (line 7) by calling the function genStar (lines 13–17). Then, we use M_v to denote the candidate set of root nodes which can guarantee that the star to be generated and the stars that have been generated share at least one common vertex (line 9). Similarly, after obtaining the vertex r with respect to the h value, a new star is generated (lines 10–11). This process (lines 8–11) terminates until the set Q is empty.

For a subgraph matching query Q, Algorithm 2 can produce a star decomposition D and determine an order of these stars, $T_1 \ldots T_m$, such that $\bigcup_{1 \le i < j} V(T_i) \cap V(T_j) \neq \emptyset, 1 \le j \le m$. Based on this matching order, we further introduce the concept of the partial query graph.

Definition 6 *(Partial query graph)* The *partial query graph* $P_j, 1 \le j \le m$ is a subgraph of G_Q, where (1) $V(P_j) = \bigcup_{1 \le i \le j} V(T_i)$ and (2) $E(P_j) = \bigcup_{1 \le i \le j} E(T_i)$. Obviously, $P_1 = T_1$ and $P_m = G_Q$. Let $\Omega(T_i)$ and $\Omega(P_i)$ be the set of matching results for star T_i and partial query graph P_i, respectively. We have $\Omega(P_1) = \Omega(T_1)$, $\Omega(P_t) = \Omega(P_{t-1}) \bowtie \Omega(T_t)$, and $\Omega(P_m) = \Omega(Q)$.

Example 2 Consider the query Q_1 over G_1, where $\mathsf{h}(?y) = \frac{2}{3}$, $\mathsf{h}(?x) = \frac{4}{2}$, and $\mathsf{h}(?z) = \frac{1}{4}$. According to Algorithm 2, the first selected vertex is $?x$ and the corresponding star is T_2 (T'_1) in Fig. 2. Then, stars T_1 (T'_2) and T_3 (T'_3) are generated. Based on this order, P_1, P_2, and P_3 in Fig. 2 are the partial query

Algorithm 2: StarDecompose(Q)

```
   Input : A Query graph $Q$: $\{tp_1, tp_2, ..., tp_n\}$
   Output: A Queue of stars $D$: $\{T_1, ..., T_m\}$
1  $D \leftarrow \emptyset$;            // $D$: the queue of stars, $V(D)$: the set of vertices in $D$
2  $Q_c \leftarrow \{s \mid s \in Sub(Q) \wedge s \notin Var\}$;        // $Sub(Q)$: the set of subjects in $Q$
3  if $Q_c \neq \emptyset$ then
4      $r \leftarrow \arg\max_{v \in Q_c} \mathsf{h}(v)$
5  else
6      $r \leftarrow \arg\max_{v \in Sub(Q)} \mathsf{h}(v)$
7  genStar($r, Q, D$);                  // generate the star rooted at vertex $r$
8  while $Q \neq \emptyset$ do
9      $M_v \leftarrow \{s \mid s \in Sub(Q) \wedge s \in V(D)\} \cup \{s \mid (s,p,o) \in Q \wedge o \in V(D)\}$;
10     $r \leftarrow \arg\max_{v \in M_v} \mathsf{h}(v)$;
11     genStar($r, Q, D$);
12 return $D : \{T_1, ..., T_m\}$;
13 Function genStar ($r, Q, D$)                              // generate a star
14     $T.r \leftarrow r$;
15     $T.L \leftarrow \{(p_i, l_i) \mid (r, p_i, l_i) \in Q\}$;
16     $D$.enqueue($T$);
17     $Q \leftarrow Q \setminus \{(r, p_i, l_i) \mid (p_i, l_i) \in T.L\}$;
```

graphs of Q_1. Obviously, P_3 is exactly the original query graph G_{Q_1}. □

4.4 Subgraph Matching Algorithm Using MapReduce

Next, we show how to use MapReduce to answer a subgraph matching query in a left-deep-join framework and demonstrate the pseudocode of our StarMR algorithm.

We can get $P_1 = T_1$ and $P_m = G_Q$ according to the partial query graph definition. In addition, the notation $\Omega(T_i)$ represents the matching results of the star T_i. The notation $\Omega(P_i)$ represents the matching results of the partial query graph P_i. The partial query graph P_i can be obtained by joining the star T_i and partial query graph P_{i-1} together. Thus, the intersection of P_{i-1} and T_i will serve as the joining key. To process the subgraph matching query efficiently, we present Algorithm 3 to answer the queries, which uses MapReduce.

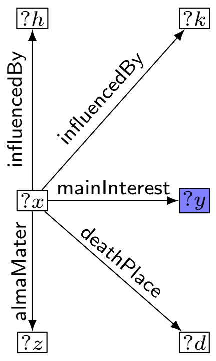
Algorithm 3 decomposes the query Q into a queue K of stars (line 1) and matches these stars in MapReduce iterations (lines 3–9). Each round of the MapReduce iteration joins one star with the partial results until all stars are matched. Map function consists of two parts: (1) when the input value is $N(v)$ (lines 12–20), the function matches the star T_t over every neighbor information $N(v)$ in RDF graph G *in parallel* (line 13), then let the matching results of intersection of vertex sets of star T_t and partial query graph P_{t-1}, i.e., μ_{key}, be keys (line 22); and (2) when the input value is a mapping in $\Omega(P_{t-1})$ (lines 22–23), similarly, let μ_{key} be keys (line 20). Every mapping μ in $\Omega_v(T_t)$ and $\Omega(P_{t-1})$ is transformed into a key-value pair (μ_{key}, μ). Note that when $t = 1$, the output of map is $(\emptyset, \mu)$ (lines 14–16). reduce function joins the matching results of T_t and P_{t-1} to generate the matching results of P_t with μ_{key} as the keys (lines 24–26). Figure 3 shows the matching process of StarMR.

Theorem 1 *Given an RDF graph G and a query graph G_Q, we assume that Algorithm 2 decomposes G_Q into a queue of*

Algorithm 3: StarMR

```
   Input  : RDF graph $G = (V, E, \Sigma)$, A query graph $Q$: $\{tp_1, tp_2, ..., tp_n\}$
   Output: The answer set: $\Omega(Q)$
1  $K \leftarrow$ StarDecompose($Q$);                    // decompose query graph
2  $\Omega(Q) \leftarrow \emptyset$, $t \leftarrow 1$;
3  while $K$ is not empty do                              // MapReduce iterations
4      $T_t \leftarrow K$.dequeue();
5      map($\emptyset, N(v)$);
6      if $t > 1$ then
7          map($\emptyset, \mu$) s.t. $\mu \in \Omega(P_{t-1})$;
8          reduce($\mu_{key}, (\Omega_1, \Omega_2)$ s.t. $\Omega_1 \subseteq \Omega(T_t) \wedge \Omega_2 \subseteq \Omega(P_{t-1})$);
9      $t \leftarrow t + 1$;
10 return $\Omega(P_m)$;
11 Function map ($\emptyset$, $N(v)$ or $\mu$)
12     if value is an adjacency list $N(v)$ in $G$ then           // match the star $T_t$
13         $\Omega_v(T_t) \leftarrow starMatch(T_t, N(v))$ ;
14         if $t = 1$ then
15             foreach $\mu \in \Omega_v(T_1)$ do
16                 return $(\emptyset, \mu)$ ;
17         else
18             foreach $\mu \in \Omega_v(T_t)$ do          // get the mapping $\mu_{key}$ as key
19                 $\mu_{key} \leftarrow \{(u_k, \mu(u_k)) \mid u_k \in V(P_{t-1}) \cap V(T_t)\}$;
20                 return $(\mu_{key}, \mu)$;
21     else                                                       // value is a mapping
22         $\mu_{key} \leftarrow \{(u_k, \mu(u_k)) \mid u_k \in V(P_{t-1}) \cap V(T_t)\}$ ;
23         return $(\mu_{key}, \mu)$ ;
24 Function reduce ($\mu_{key}, (\Omega_1, \Omega_2)$)
25     foreach $(\mu, \mu') \in \Omega_1 \times \Omega_2$ do
26         return $(\emptyset, \mu \cup \mu')$ ;                          // $\mu \cup \mu' \in \Omega(P_t)$
```

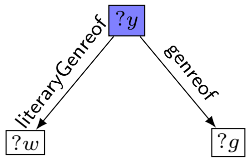
Fig. 3 Matching stars with the basic algorithm **StarMR**



(a) Decomposed Star T_2 (T_1')

?x	?y	?k	?h	?d	?z
Karl_Marx	Labour_relation	Adam_Smith	Aristotle	London	University_of_Bonn
Karl_Marx	Labour_relation	Aristotle	Adam_Smith	London	University_of_Bonn
Karl_Marx	Natural_Science	Adam_Smith	Aristotle	London	University_of_Bonn
Karl_Marx	Natural_Science	Aristotle	Adam_Smith	London	University_of_Bonn
Weininger	Logic	Kaut	Arthur	Austria	University_of_Vienna
Weininger	Logic	Arthur	Kaut	Austria	University_of_Vienna

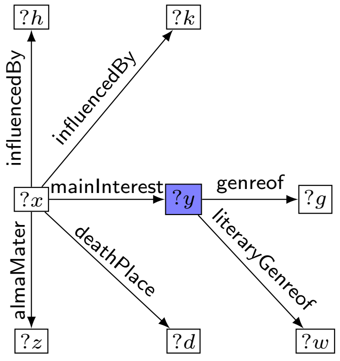
(b) Matching T_2 (T_1') with StarMR



(c) Decomposed Star T_1 (T_2')

?y	?c	?p
Ecnomic	Arndt	Economy
Logic	Fereydoon	Cokked_Thinking

(d) Matching T_1 (T_2') with StarMR



(e) $T_1' \ltimes T_2'$

?x	?y	?k	?h	?d	?z	?g	?w
Weininger	Logic	Arthur	Kaut	Austria	University_of_Vienna	Arndt	Economy
Weininger	Logic	Kaut	Arthur	Austria	University_of_Vienna	Fereydoon	Cokked_Thinking

(f) Reduce $T_1' \ltimes T_2'$

stars $D = \{T_1, \ldots, T_m\}$. Algorithm 3 gives the correct query results, and the number of MapReduce iterations is m.

Proof (Sketch) The algorithm correctness can be proved as follows: (i) In the first MapReduce iteration of Algorithm 3, Algorithm 1 is invoked by the map function to match the star T_1; then, the matching set $\Omega(T_1)$, i.e., $\Omega(P_1)$, can be obtained. From the second round of the MapReduce iteration on, Algorithm 3 matches a new star T_t in each round according to the matching orders computed by Algorithm 2. In this way, the star matching result $\Omega(T_t)$ can be given out. (ii) In the reduce function, the matching results $\Omega(P_{t-1})$ and $\Omega(T_t)$ are joined, and the joining results $\Omega(P_t)$ can be obtained, where $\Omega(P_{t-1})$ is the matching results of partial query graph P_{t-1}, which is obtained in the last round of the MapReduce iteration. Consequently, the answer set of query Q, i.e., $\Omega(P_m)$, can be obtained in the m round of the MapReduce iteration. $\square$

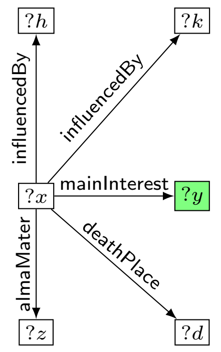
Theorem 2 *The time complexity of Algorithm 3 is bounded by $O(|V|^m \cdot |N_{max}|^{m \cdot |L_{max}|})$, where $|V|$ is the size of G, $|N_{max}|$ is the largest out degree in G,* and *$|L_{max}|$ is the largest out degree in G_Q.*

Proof (Sketch) In Algorithm 3, each round of the MapReduce iteration matches one star; therefore, it can evaluate the query Q in m rounds. The time complexity of Algorithm 3 consists of two parts: (1) the time complexity of star matching is $\sum_{1 \le t \le m} \sum_{v \in V} (|N(v)| + |\Omega(T_t)|)$; (2) the time complexity of join operations is $\sum_{1 < t \le m} |\Omega(P_{t-1})| \times |\Omega(T_t)|$. In the worst case, every leaf in T_t can match all neighboring vertices of a vertex v, $v \in V$, i.e., $|\Omega_v(T_t)| = |N(v)|^{|T_t.L|}$. Thus, the time complexity of Algorithm 3 is bounded by $O(|V|^m \cdot |N_{max}|^{m \cdot |L_{max}|})$ $\square$

5 Two Optimization Strategies

In this section, two optimization strategies are proposed to improve the efficiency of the StarMR algorithm. The first one leverages the inherent semantics of the RDF graph to reduce the overhead expense; the other technique improves the query performance by postponing the Cartesian product operations.

Fig. 4 Matching stars with the optimization algorithm **StarMR$_{\text{opt}}$**



(a) Decomposed Star T_2 (T_1')

$?x$	$?\mathbf{y}$

Karl_Marx
Weininger
	{Labour_relation,Natural_Science}
	{Logic}

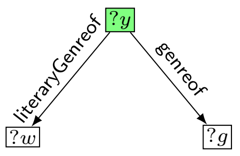
$?k$	$?\mathbf{h}$

{Adam_Smith,Aristotle}
{Kaut,Arthur}
	{Aristotle,Adam_Smith}
	{Arthur,Kaut}

$?\mathbf{d}$	$?\mathbf{z}$

{London}
{Austria}
	{University_of_Bonn}
	{University_of_Vienna}

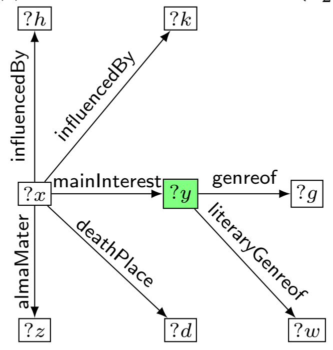
(b) Matching Results to Star T_2 (T_1')



(c) Decomposed Star T_1 (T_2')

$?y$	$?c$	$?p$
Ecnomic	{Arndt}	{Economy}
Logic	{Fereydoon}	{Cokked_Thinking}

(d) Matching Results to Star T_1 (T_2')



(e) $T_1' \ltimes T_2'$

$?x$	$?\mathbf{y}$
Weininger	Logic
$?\mathbf{k}$	$?h$
{Arthur,Kaut}	{Kaut,Arthur}
$?d$	$?\mathbf{z}$
{Austria}	{University_of_Vienna}
$?g$	$?\mathbf{w}$
Fereydoon	Cokked_Thinking

(f) Reduce $T_1' \ltimes T_2'$

5.1 RDF Property Filtering

We take advantage of the inherent semantics embedded in RDF graphs to filter out unpromising computations. As mentioned in Sect. 1, the RDF describes resources by defining classes and properties. In addition, the RDF Schema (RDFS)[2] is an extended version of RDF, which is regarded as a framework to define classes of resources. For instance, the RDF triple $(s, \texttt{rdf : type}, C)$ declares that resource s is an instance of the class C, denoted by $s \in C$. We assume that for each subject s in an RDF graph G, there exists at least a triple $(s, \texttt{rdf : type}, C) \in G$. We believe that this assumption is reasonable since every resource should belong to at least one type in the real world.

Example 3 As shown in Fig. 1, the triple (`University_of_Bonn`, `rdf : type`, `Organisation`) denotes that `University_of_Bonn` is an instance of the class `Organization`. Other such triples in Fig. 1 are omitted. Moreover, all instances of class `Organisation` are highlighted in green. Similarly, there exist other classes in G_1, e.g., `Person` in blue and `Country` in red. □

Given an RDF graph $G = (V, E, \Sigma)$, let $P'(C) = \{p \mid (s, p, o) \in G \wedge s \in C\}$ be the set of RDF properties of class C. Note that the size of classes in an RDF graph is much less than vertices in the corresponding RDF graph. When matching a star T in the function Map$(T, N(v))$, RDF properties of C can be used to filter out input data as follows: if $v \in C \wedge P(T.r) \nsubseteq P'(C)$, the procedure StarMatch$(T, N(v))$ in map function can be pruned.

Example 4 Consider matching star T_2 rooted at $?x$ in Fig. 2 over G_1 in Fig. 1, we have $P(?x) = \{\texttt{influencedBy}, \texttt{mainInterest}, \texttt{deathPlace}, \texttt{almaMater}\}$ and $P'(\texttt{Organisation}) = \{\texttt{country}\}$. Due to `University_of_Glagow` ∈ `Orga–nisation` $\wedge\, P(?x) \nsubseteq P'(\texttt{Organisation})$, the computation of StarMatch$(T_2$, $N(\texttt{University\_of\_Glagow}))$ can be pruned. □

5.2 Postponing Cartesian Product Operations

In this section, we first demonstrate the expensive cost introduced by our basic algorithm and then illustrate an

[2] https://www.w3.org/TR/rdf-schema/.

efficient solution. Finally, the optimization method will be analyzed.

In the initial star matching phase of our basic algorithm StarMR, the function map is invoked to match the star T_1, where $T_1 = (r, L)$ and $L = \{(p_1, l_1), \dots, (p_t, l_t)\}$; then, we can obtain the candidate matching set $S(l_i)$ of every leaf vertex l_i. The matching set $S(l_i)$ is generated according to the label of l_i matched with leaf vertices over $N(v)$. Next, we can get the matching results by doing the Cartesian product operations on the candidate matching sets of vertices in the star T_1. Unfortunately, a majority of matching results cannot contribute to the final results. In other words, the Cartesian product operation is not necessarily done during the star matching phase. Even worse, that operation can incur expensive cost in addition. Since the Cartesian product operation leads to expensive costs, after executing subgraph matching with our basic algorithm StarMR, an improvement is proposed to postpone the Cartesian product operation.

Let f be a mapping from vertices in a star to the candidate matching sets of the corresponding vertices. We use $\Omega'(T)$ and $\Omega'(P)$ to denote the matching results of star T and partial query graph P, respectively. In order to reduce the intermediate results cost. We develop an efficient optimization algorithm, denoted by StarMR$_{opt}$. In our optimization method, the initial star matching phase only needs to calculate the candidate matching sets of leaves of the star T_1.

Algorithm 4: StarMR$_{opt}$

```
   Input  : RDF graph $G = (V, E, \Sigma)$, A subgraph matching $Q$: $\{tp_1, tp_2, ..., tp_n\}$
   Output: The answer set: $\Omega(Q)$
1  $K \leftarrow$ StarDecompose($Q$);                                   // decompose query graph
2  $\Omega(Q) \leftarrow \emptyset$, $t \leftarrow 1$;
3  while $K$ is not empty do                                           // MapReduce iterations
4      $T_t \leftarrow K$.dequeue();
5      map($\emptyset, N(v)$);
6      if $t > 1$ then
7          map($\emptyset, \mu$) s.t. $\mu \in \Omega(P_{t-1})$;
8          reduce($\mu_{key}, (\Omega_1, \Omega_2)$ s.t. $\Omega_1 \subseteq \Omega(T_t) \wedge \Omega_2 \subseteq \Omega(P_{t-1})$);
9      $t \leftarrow t + 1$;
10 return $\Omega(P_m)$;
11 Function $starMatch_{opt}(T, N(v))$              // Cartesian product can be postponed.
12     if $T.r$ matches vertex $v$ then
13         foreach $(p_i, l_i) \in T.L$ do
14             if $p_i \notin Var$ then
15                 $S(l_i) \leftarrow \{v' \mid (p_i, v') \in N(v) \wedge l_i$ matches $v'\}$;
16             else
17                 $S(l_i) \leftarrow \{v' \mid (\_, v') \in N(v) \wedge l_i$ matches $v'\}$;
18         $f \leftarrow \{(u, S(u)) \mid u \in V(T)\}$;
19     return $f$;
20 Function map ($\emptyset$, $N(v)$ or $f'$ s.t. $t > 1$)  // RDF property filtering is applied.
21     if $t = 1$ then
22         return ($\emptyset, starMatch_{opt}(T_1, N(v))$);
23     else
24         $V_{key} : \{u_1, \dots, u_k\} \leftarrow V(P_{t-1}) \cap V(T_t)$;
25         if value is an adjacency list $N(v)$ in $G$ then
26             $f \leftarrow starMatch_{opt}(T_t, N(v))$;
               // do $f(u_1) \times \cdots \times f(u_k)$ to get $\Omega_{key}$
27             $\Omega_{key} \leftarrow \{f_1 \cup \cdots \cup f_k \mid f_i = \{(u_i, \{v\})\} \wedge u_i \in V_{key} \wedge v \in f(u_i)\}$;
28             foreach $f_{key} \in \Omega_{key}$ do                          // $f \in \Omega_f$
29                 return ($f_{key}, f - V_{key}$);
30         else
31             $\Omega_{key} \leftarrow \{f'_1 \cup \cdots \cup f'_k \mid f'_i = \{(u_i, \{v\})\} \wedge u_i \in V_{key} \wedge v \in f'(u_i)\}$;
32             foreach $f_{key} \in \Omega_{key}$ do                          // $f' \in \Omega_{f'}$
33                 return ($f_{key}, f' - V_{key}$);
34 Function reduce ($f_{key}, (\Omega_f, \Omega_{f'})$)
35     foreach $(f, f') \in \Omega_f \times \Omega_{f'}$ do
36         return ($\emptyset, f_{key} \cup f \cup f'$);
```

In Algorithm 1, for each star $T = (r, L), L = \{(p_1, l_1), \ldots, (p_t, l_t)\}$, we do the Cartesian product operation $\{v\} \times S(l_1) \ldots S(l_t)$ to get the matching results of the star T over $N(v)$, which is $\Omega_v(T)$. However, unlike Algorithm 1 enumerating the matching results of each leaf node in detail, the starMatch$_{opt}(T, N(v))$ method adds $(u, S(u))$ to a mapping f (line 18). Similarly, the functions map and reduce are also changed. In the first iteration, the function map invokes the starMatch$_{opt}(T, N(v))$ method and return the mapping f. Meanwhile, in the t-th MapReduce iteration, first joining the t-th star T_t with the partial query graph P_{t-1}; then, the common vertices key sets $V_{key} : \{u_1, \ldots, u_k\}$ can be obtained. (line 19). The input of the function map can be classified into two categories. One is an adjacency list $N(v)$ in G; the other is a mapping f. The map function does the Cartesian product operation $f(u_1) \times \cdots \times f(u_k)$, where $u_k \in V(P_{t-1}) \cap V(T_t)$ (lines 22–33). The reduce function takes (k, v) pairs as the input, and for those (k, v) pairs with the same key, the reduce function can join them together and emerge a new (k, v) pair. For every $f \in \Omega'(P_m)$, we do the remaining Cartesian product operation $f(u'_1) \times \cdots \times f(u'_k)$ to get the final matching results $\Omega(Q)$, where $u'_k \in V(Q) \setminus \bigcup_{1<t\leq m}(P_{t-1} \cap V(T_t))$ (lines 19–20). Although the strategy does not change the complexity of our algorithm, they can improve query efficiency significantly. We will take an example to further illustrate the optimization algorithm.

Example 5 When answering Q_1 over G_1, starMatch$(T'_1, N(\texttt{Karl_Marx}))$ obtains the *candidate set* $S(l_i)$ of every leaf l_i in T'_1, e.g., $S(?y) = \{\texttt{Natural\_Sci- ence}\}$. We have $V(P_1) \cap V(T'_2) = \{?y\}$. Next, star T'_2 is matched, but the candidate set of $?y$ in all $N(v)$ in G_1 does not contain vertex `Natural_Science`. Thus, we do not need to do the Cartesian product operation $S(?z) \times S(?y) \times S(?w)$ to get $\Omega_{\texttt{Karl\_Marx}}(T'_1)$, whose cost is prohibitively expensive. The more detailed explanation could refer to Figs. 3 and 4. □

6 Experiments

We have carried out extensive experiments on both synthetic and real-world RDF graphs to verify the efficiency and scalability of StarMR and compared it with the optimization method StarMR$_{opt}$, the state-of-the-art S2X, and SHARD. Our algorithm is orthogonal to the graph partitioning and placement strategies in the cluster environment. In particular, for the implementation of StarMR, we use the default partitioner employed by the Hadoop Distributed File System (HDFS).

6.1 Settings

The prototype program, which is implemented in Scala using Spark, is deployed on an 8-site cluster connected by a gigabit Ethernet. Each site has a Intel(R) Core(TM) i7-7700 CPU with four cores of 3.60GHz, 16GB memory, and 500GB disk. We used Hadoop 2.7.4 and Spark 2.2.0. All the experiments are carried out on Linux (64-bit CentOS) operating systems.

We used three RDF datasets, including synthetic dataset WatDiv,[3] LUBM,[4] and real-world dataset DBpedia[5] in our experiments. (1) WatDiv is an RDF benchmark, which allows users to define their own datasets and generate test datasets with different sizes; (2)LUBM is also a standard RDF synthetic benchmark, which is used to generate datasets of different scales; (3) DBpedia is a real-world dataset extracted from Wikipedia. As listed in Table 2, we summarize the statistics of these datasets. For RDF queries, we group them into four categories according to their shapes, including *linear* queries (L), *star* queries (S), *snowflake* queries (F), and *complex* queries (C), which are listed in Table 3. Regarding WatDiv, it gives 20 basic query templates. We leveraged 14 testing queries provided by the LUBM benchmark. Due to the absence of query templates on DBpedia, we designed eight queries, covering the four query categories mentioned above.

6.2 Experiments on WatDiv Datasets

In this section, we verify the efficiency and scalability of our methods on WatDiv datasets. This section is organized as the following orders: (1) brief introduction on WatDiv dataset; (2) validate the efficiency of the optimization algorithm StarMR$_{opt}$ and analyze the experimental results; (3) examine the scalability of these methods.

Waterloo SPARQL Diversity Test Suite [2], denoted by WatDiv, was developed to measure the performance of an RDF data management system, i.e., the RDF data scale and the SPARQL queries spectrum can be defined by the user themselves. More specifically, WatDiv is composed of a data generator and a query generator. We can generate our own datasets through a designated dataset description language using the data generator. Our experiments were carried out on WatDiv1M, WatDiv10M, and WatDiv100M datasets, where "1M" indicates the number of triples is one million.

[3] http://dsg.uwaterloo.ca/watdiv/.

[4] http://swat.cse.lehigh.edu/projects/lubm/.

[5] http://wiki.dbpedia.org/downloads-2016-10.

Table 2 Datasets

Datasets	|V|	|E|
LUBM4	78,595	493,844
LUBM40	864,238	5,495,742
LUBM400	8,675,133	55,256,074
WatDiv1M	158,118	1,109,678
WatDiv10M	1,052,571	10,916,457
WatDiv100M	10,250,947	108,997,714
DBpedia	6,060,648	23,509,250

Table 3 Queries

Query	L	S	F	C
LUBM	$Q_1, Q_3, Q_5, Q_6, Q_{10}, Q_{11}, Q_{13}, Q_{14}$	Q_4	Q_7, Q_8, Q_{12}	Q_2, Q_9
WatDiv	L_1, L_2, L_3, L_4, L_5	$S_1, S_2, S_3, S_4, S_5, S_6, S_7$	F_1, F_2, F_3, F_4, F_5	C_1, C_2, C_3
DBpedia	L_1, L_2	S_1, S_2	F_1, F_2	C_1, C_2

6.2.1 Efficiency on WatDiv

Experiments were conducted on WatDiv100M to verify the query efficiency of our method. As shown in Fig. 5, our optimization method StarMR$_{opt}$ has the best query efficiency on all 20 queries. The basic method StarMR is also much better than S2X and SHARD. The query execution times of these 20 queries are given in Table 4. For the query F_3 (resp. C_2), it can be observed that S2X cannot finish in the time limit (1×10^4s), denoted by INF, while StarMR$_{opt}$ and StarMR can return the answers within 20s (resp. 28s) and 65s (resp. 94s), respectively. The average execution speed of the remaining 18 queries in StarMR$_{opt}$ is about 11 times faster than of that in S2X. Furthermore, the average query execution time of StarMR$_{opt}$ covering all the query categories is about 17 seconds, which is up to 47 times and on average 26 times faster than SHARD, i.e., our optimization method on average, outperforms S2X and SHARD by an order of magnitude over WatDiv100M.

In addition, compared with StarMR, the time of StarMR$_{opt}$ is reduced from 44.86% to 74.94%, as listed in Table 4. Thus, StarMR$_{opt}$ is able to evaluate the query more efficiently. We can observe that StarMR$_{opt}$ tends to be stable over all the queries. In contrast, S2X and SHARD fluctuate dramatically.

The experimental result demonstrates that the effect of optimization strategies in StarMR$_{opt}$ is significant. We analyze that the reasons include: (1) S2X and SHARD joined the intermediate matching results of all triple patterns in subgraph and did not leverage any structure and semantic information, leading to expensive cost; (2) StarMR did Cartesian product operations in the star matching phase, which may lead to expensive cost; and (3) in StarMR$_{opt}$, a part of invalid input data were pruned by utilizing RDF properties embedded in RDF graphs, and a large number of Cartesian operations were postponed and reduced.

6.2.2 Scalability on WatDiv

We compared StarMR$_{opt}$ with StarMR, S2X, and SHARD. The scalability comparison experiments were carried out on various scale WatDiv datasets and different experimental cluster sites.

Different Size of Datasets For the reason that S2X cannot finish in the time limit (1×10^4 s) on query F_3 and C_2, we conducted experiments on WatDiv datasets over the other 18 queries except them. Moreover, the average times of

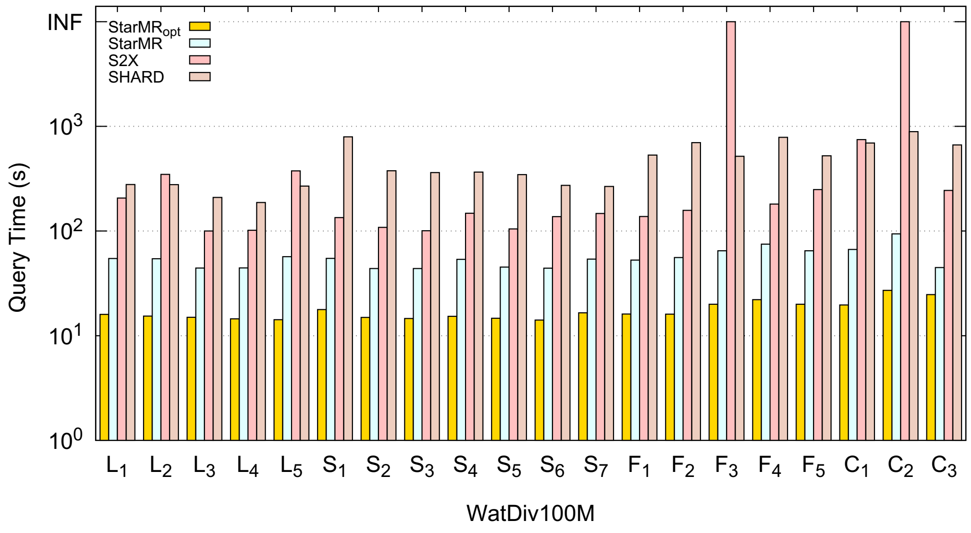


Fig. 5 The results on different queries over WatDiv100M

Table 4 The query times (in s) of StarMR$_{opt}$, StarMR, S2X, and SHARD on WatDiv100M

Query	*S*1	*S*2	*S*3	*S*4	*S*5
StarMR$_{opt}$	17.773	14.973	14.606	15.331	14.698
StarMR	54.741	43.882	43.884	53.696	45.313
S2X	134.380	108.325	100.568	147.862	104.754
SHARD	794.303	376.891	361.489	365.994	346.176
Query	*S*6	*S*7	*C*1	*C*2	*C*3
StarMR$_{opt}$	14.096	16.565	19.667	27.144	24.712
StarMR	44.165	53.858	66.647	93.888	44.820
S2X	137.311	147.112	747.289	INF	244.216
SHARD	272.885	266.494	692.901	890.009	664.440
Query	*L*1	*L*2	*L*3	*L*4	*L*5
StarMR$_{opt}$	15.987	15.403	14.998	14.497	14.240
StarMR	54.636	54.379	44.281	44.476	56.826
S2X	206.564	347.468	100.222	101.839	375.845
SHARD	278.2577	277.2127	209.261	187.220	268.5197
Query	*F*1	*F*2	*F*3	*F*4	*F*5
StarMR$_{opt}$	16.092	16.081	19.954	22.118	19.987
StarMR	52.733	55.855	64.719	75.001	64.818
S2X	137.796	157.659	INF	180.965	248.616
SHARD	531.6827	698.4937	517.9577	785.1687	522.8997

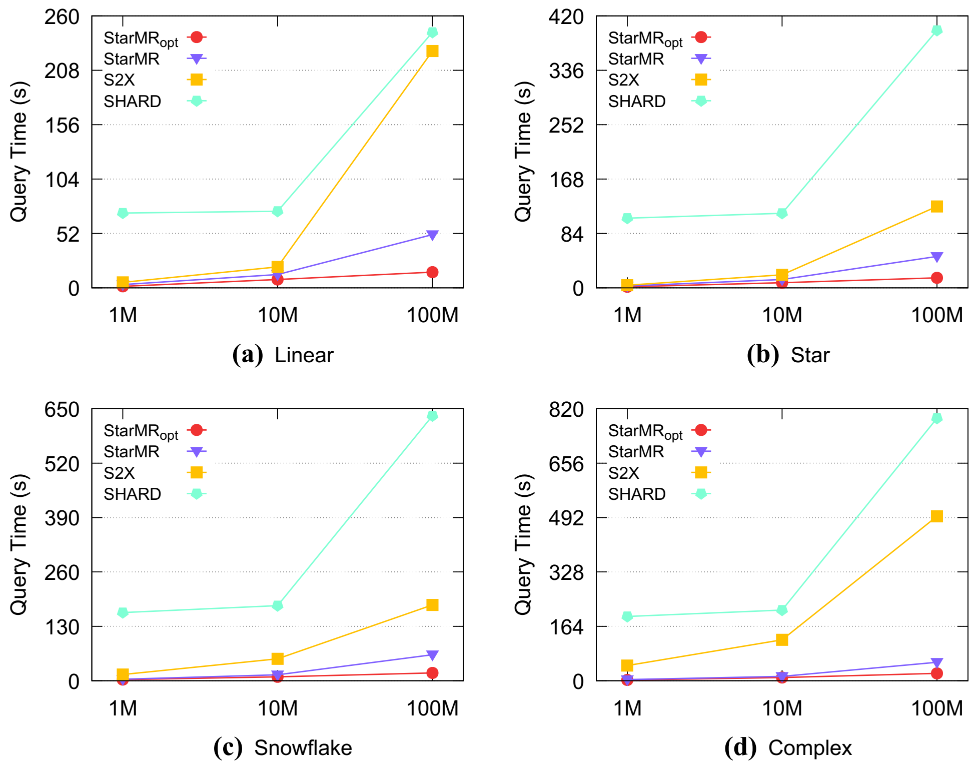
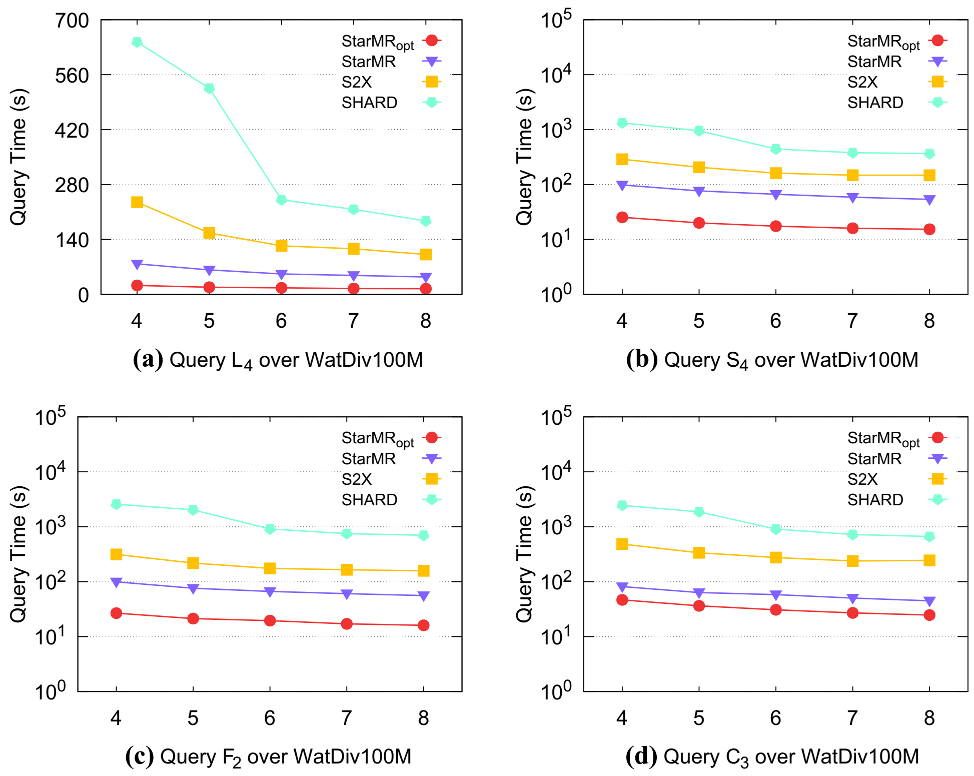


Fig. 6 Scalability on WatDiv datasets

Fig. 7 Scalability on cluster sites

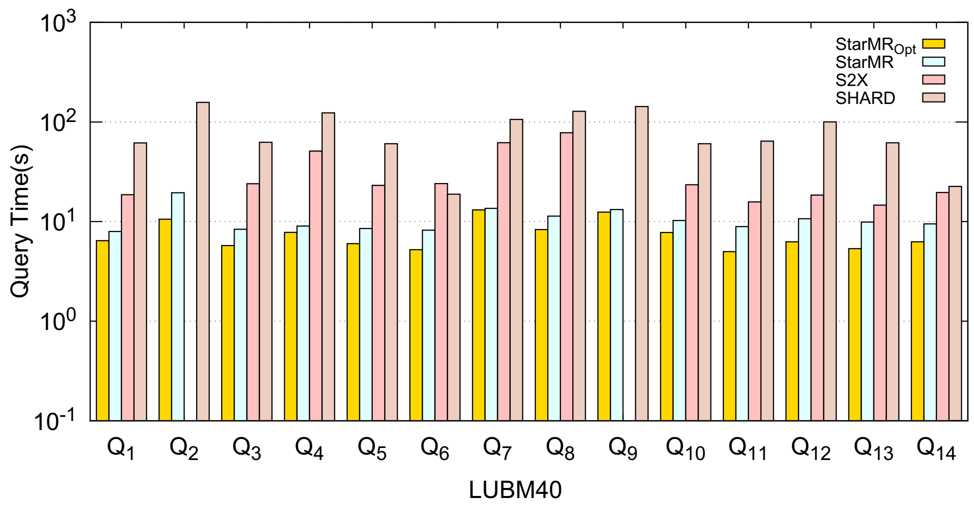


each query category were calculated, as shown in Fig. 6. When changing the size of datasets from WatDiv1M to WatDiv100M, query times of all four methods increased and StarMR$_{opt}$ was always the best one. We can observe that with the scale of the datasets increasing, the query times of S2X and SHARD increased dramatically. More specifically, the average growth rate of S2X and SHARD were 95.8% and 72.7%, respectively. In contrast, for the two methods StarMR$_{opt}$ and StarMR, the query time growth rate changed slightly. We analyzed the low performance of S2X and SHARD was that the expensive cost incurred by abundant intermediate results.

Compared with StarMR$_{opt}$, the growth rate of query times in StarMR was higher than that of StarMR$_{opt}$. As shown in Fig. 6, the performance of SHARD and S2X dropped significantly with the size of datasets increasing, especially SHARD.

Different Numbers of Cluster Sites Extensive experiments were carried out on the WatDiv100M dataset with the number of cluster sites varying from 4 to 8. During these experiments, we randomly selected one query from each of the four

Fig. 8 Efficiency on LUBM40



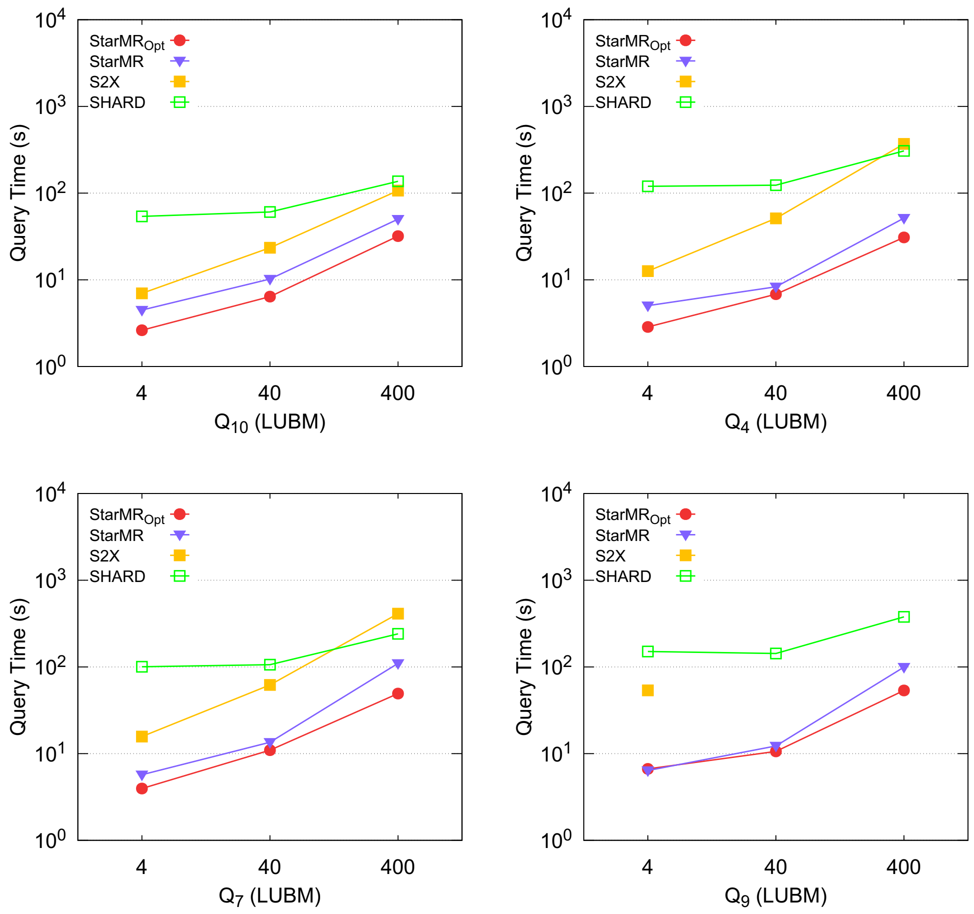


Fig. 9 Scalability on LUBM datasets

query categories, i.e., L_4, S_4, F_2, and C_3. As shown in Fig. 7, the experimental results verified our intuition that query times of all these four methods decreased as the number of cluster sites increased. This is because when the number of sites increased, the degree of parallelism also increased. Although with the number of sites increasing, the speedup ratios of S2X and StarMR$_{opt}$ are comparative; the performance of StarMR$_{opt}$ is stable for selective queries and the query times of StarMR$_{opt}$ are far less than that of S2X. For SHARD, we can observe that the query times of category F and S are around 1000 s, and in all the query categories, the query time of SHARD dropped dramatically from site 4 to site 5. It demonstrated that the performance of SHARD was extremely dependent on the experimental environment.

6.3 Experiments on LUBM

The Lehigh University Benchmark [10], denoted by LUBM, is a synthetic dataset, which aims at evaluating the performance and capability of various knowledge base systems. To verify the stability of StarMR$_{opt}$, extensive experiments were conducted not only on the WatDiv datasets but also on the standard benchmark LUBM. In this paper, we use the LUBM datasets scaling from LUBM4 to LUBM400. The following experiments focus on two aspects: the efficiency and scalability of these methods.

6.3.1 Efficiency on LUBM

The efficiency validating experiments were conducted on the LUBM40 dataset. Figure 8 compared our proposed methods StarMR$_{opt}$ and StarMR with the other two methods.

We can observe that StarMR$_{opt}$ outperformed StarMR, S2X, and SHARD for all query categories. The performance of S2X was better than SHARD for most queries, for which we analyzed the reason was that SHARD cannot process several triple patterns in a single MapReduce job. However, when answering the query Q_2 (resp. Q_9), S2X terminated with errors. It was because that the query process produced too many intermediate results, thus incurring expensive overhead for S2X. While StarMR$_{opt}$ and StarMR can return the answers within 10.6s (resp. 19.5s) and 12.5s (resp. 13.2s), respectively. The average runtimes of the remaining 12 queries in StarMR$_{opt}$ were about 5 times faster than of that in S2X, and about 11 times faster than of that in SHARD.

In addition, the average query runtime of StarMR$_{opt}$ covering all the query categories was about 7 seconds, which was up to 29 times and in average 12 times faster than

SHARD. Overall, the experimental results clearly demonstrated the superior performance and stability of our optimization method StarMR$_{opt}$, which was an order of magnitude faster than other methods for all queries.

6.3.2 Scalability on LUBM

Similar to the previous scalability-examining experiments on the WatDiv datasets, we carried out extensive experiments on various size of LUBM datasets, i.e., LUBM4, LUBM40, and LUBM400. In addition, we verified the scalability of our methods by changing the number of sites. Furthermore, we selected four queries, Q_4, Q_7, Q_9, Q_{10}, which are covering all the query categories. The intuitive experimental results are shown in Figs. 9 and 10.

Different Size of Datasets We evaluated the scalability of the StarMR$_{opt}$ by changing the scale of LUBM datasets. As shown in Fig. 9, the experimental results showed that the method we proposed significantly outperforms S2X and SHARD. We can observe that the query performance of all the queries decreased with the scale of datasets increasing. When S2X answers query Q_9 over LUBM40 and LUBM400, some errors occurred, which revealed that S2X cannot complete the query and often aborted. In contrast, the runtimes of our methods were significantly less than S2X and SHARD covering all queries. The average runtime of StarMR$_{opt}$ covering all the query categories was about 18 seconds, which was up to 23 times faster than that of S2X, and 13 times faster than that of SHARD. We analyze the reason can be that our optimization method exploited the structure features, semantic information, and heuristic algorithms, which can reduce the expensive overhead incurred by the intermediate results.

Different Numbers of Cluster Sites The scalability of each method was evaluated on LUBM400 dataset with the number of cluster sites varying from 4 to 8, as shown in Fig. 10. It was intuitive that for most queries, the runtimes decreased with the cluster sites increasing. We analyzed that

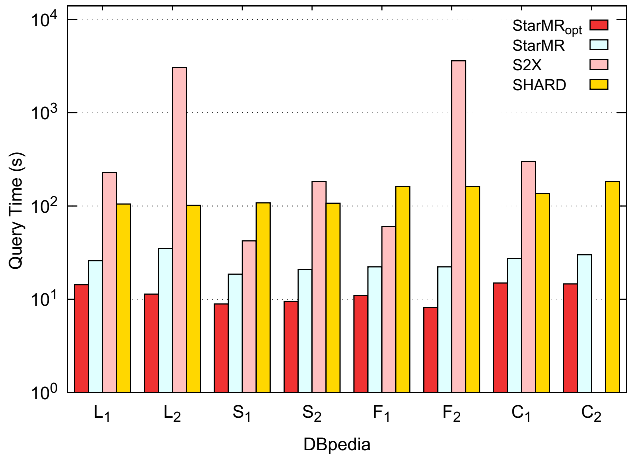


Fig. 11 Efficiency on DBpedia

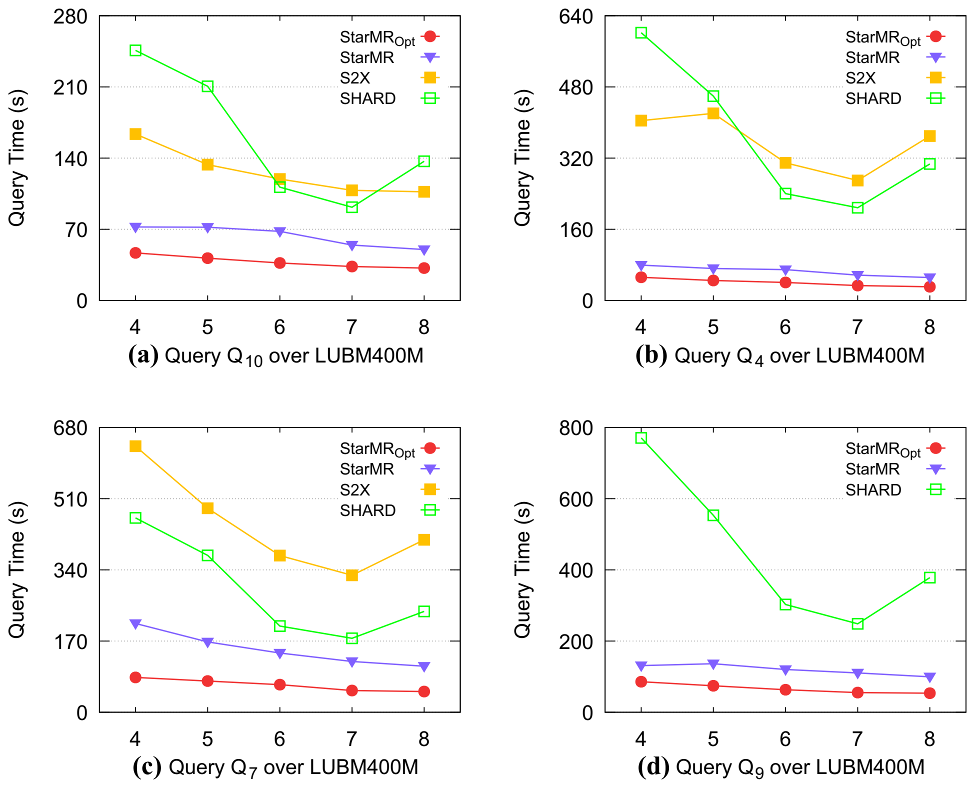
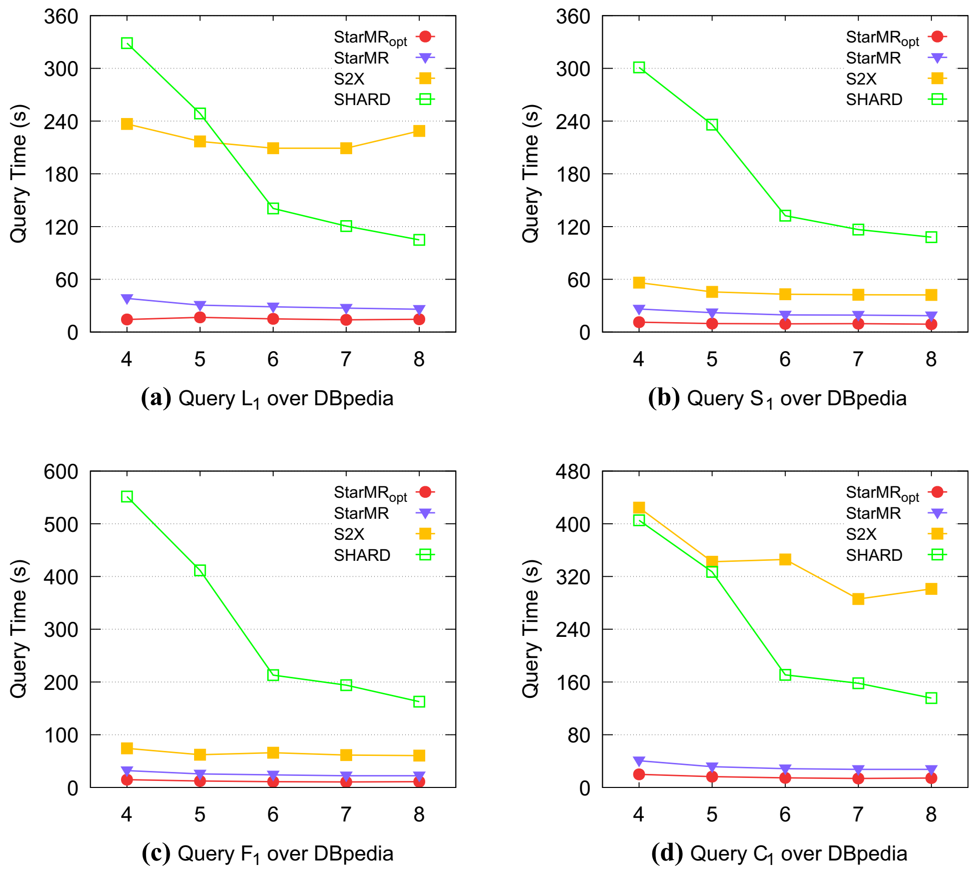


Fig. 10 Scalability on cluster cites

Fig. 12 Scalability on DBpedia



more cluster sites generate more degree of parallelism so that there were more threads to execute query tasks. However, the runtimes of S2X and SHARD decreased from 4 to 7, but increased from 7 to 8 instead. After an in-depth analysis, we concluded that when the parallelism number increased to a certain extent, the expensive communication cost became the bottleneck.

In addition, S2X cannot complete queries Q_9 on all the number of cluster sites, the result validated the conclusion that the performance of S2X cannot beat our methods. To summarize, our proposed methods achieved better performance than S2X and SHARD.

6.4 Experiments on the Real-World Dataset

To evaluate the efficiency and scalability of StarMR$_{opt}$ on a real-world dataset DBpedia, extensive experiments were carried out. Our experimental results are shown in Figs. 11 and 12. More specifically, we compared our proposed methods with the close competitors, i.e., S2X and SHARD. On the one hand, we used the average runtimes to verify the efficiency; on the other hand, we changed the number of cluster sites to examine the scalability of these methods.

6.4.1 Efficiency on DBpedia

Extensive experiments were carried out to verify the query efficiency of our method on the real-world datasets DBpedia. Figure 11 compared the different algorithms over DBpedia covering all query categories.

We can observe that SHARD showed a good performance for all the query types over DBpedia dataset, for the reason that the query time of it tended to be more stable than S2X. But SHARD was not able to beat our methods for any query. As shown in Fig. 11, StarMR$_{opt}$ also demonstrated the best query efficiency on all queries over DBpedia, and StarMR performed much better than the other two methods, i.e., S2X and SHARD. When answering C_2, i.e., the query Q_1 mentioned in Sect. 1, S2X terminated with errors. Thus, S2X cannot efficiently evaluate the complex query involving a large number of intermediate results. For the remaining seven queries, the execution speeds in StarMR$_{opt}$ was about 4 to 469 times faster than that in S2X, and about 7 to 20 times faster than that in SHARD. Compared with StarMR, the time of StarMR$_{opt}$ was reduced from 44.19% to 67.48%, i.e., the optimization effect on DBpedia was prominent. So in summary, the experimental results in Fig. 11 demonstrated that StarMR$_{opt}$ reduced invalid input data and postponed Cartesian product operations by a large margin.

From the experimental results, we can observe that the number of matching subgraphs does have an effect on the matching efficiency. If the selectivity is low, then the number of matching subgraphs is high; the execution time was long, such as the complex query F_2; for the high selectivity, the number of matching subgraphs is low, and it takes less execution time for the query, such as the query L_2.

6.4.2 Scalability on DBpedia

We conducted experiments on DBpedia to compare the scalability of these methods, with the number of cluster sites varying from 4 to 8; the experimental result is shown in Fig. 12.

The scalability experiments were conducted on four queries, i.e., L_1, S_1, F_1, and C_1 over DBpedia. Similarly, query times of these four methods decreased with the number of cluster sites varying from 4 to 8, as shown in Fig. 12. Furthermore, the optimization method StarMR_{opt} was an order of magnitude faster than SHARD on average for all four query categories, even with repeated query execution. An interesting observation was that the queries L_1 and C_1 of S2X did not decrease with the number of site increasing, which counters our intuition. According to our analysis, the reason can be that when the degree of parallelism increased to a certain extent, the cost of communication increased and gradually became the main factor. In addition, the speedup ratio of StarMR was about 1.1 times of S2X.

7 Conclusion

In this paper, we proposed the StarMR star-decomposition-based query processor for efficiently answering subgraph matching queries on big RDF graph data using MapReduce. Moreover, we also developed two optimization strategies, including RDF property filtering and postponing Cartesian product operations, to improve the basic StarMR algorithm. Our extensive experimental results on both synthetic and real-world datasets have verified the efficiency and scalability of our method, which outperforms S2X and SHARD by one order of magnitude. In the future, we will investigate how our approach can be adapted to the top-k SPARQL BGP queries. Meanwhile, we will investigate how the multi-SPARQL query settings can be improved by using our approach.

Acknowledgements The preliminary version of this article has been published in APWeb-WAIM 2018.

Author Contributions XW, JL, JW, and YC proposed the key idea and provided the professional guidance of this research work. QX and LC conducted the experiments and were the major contributors in writing the manuscript. All authors read and approved the final manuscript.

Funding This work is supported by the National Natural Science Foundation of China (61572353, 61402323, and 61472427), the Natural Science Foundation of Tianjin (17JCYBJC15400), and the Natural Science Foundation of Beijing (4172031).

Compliance with Ethical Standards

Conflict of interest The authors declare that they have no competing interests.

Availability of Data and Materials The datasets generated and analyzed during the current study are available in the WatDiv, LUBM, and DBpedia repository, WatDiv (http://dsg.uwaterloo.ca/watdiv/), LUBM (http://swat.cse.lehigh.edu/projects/lubm/), and DBpedia (http://wiki.dbpedia.org/downloads-2016-10).

References

1. Abadi DJ, Marcus A, Madden SR, Hollenbach K (2007) Scalable semantic web data management using vertical partitioning. In: Proceedings of the 33rd international conference on Very large data bases. VLDB Endowment, pp 411–422
2. Aluç G, Hartig O, Özsu MT, Daudjee K (2014) Diversified stress testing of RDF data management systems. In: International Semantic Web Conference. Springer, pp 197–212
3. Berners-Lee T, Hendler J, Lassila O (2001) The semantic web. Sci Am 284(5):34–43
4. Bizer C, Heath T, Berners-Lee T (2011) Linked data: The story so far. In: Semantic services, interoperability and web applications: emerging concepts. IGI Global, pp 205–227
5. Harris S, Seaborne A (2013) SPARQL 1.1 query language. W3C recommendation, W3C
6. Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun ACM 51(1):107–113
7. Dyer M, Greenhill C (2000) The complexity of counting graph homomorphisms. Random Struct Algorithms 17(3–4):260–289
8. Erling O, Mikhailov I (2010) Virtuoso: Rdf support in a native rdbms. In: de Virgilio R, Giunchiglia F, Tanca L (eds) Semantic web information management. Springer, Berlin, Heidelberg, pp 501–519
9. Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I (2014) Graphx: Graph processing in a distributed dataflow framework. In: 11th USENIX symposium on operating systems design and implementation (OSDI 14), pp 599–613
10. Guo Y, Pan Z, Heflin J (2005) Lubm: A benchmark for owl knowledge base systems. Web Semant Sci Serv Agents World Wide Web 3(2–3):158–182
11. Gurajada S, Seufert S, Miliaraki I, Theobald M (2014) Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, pp 289–300
12. Hammoud M, Rabbou DA, Nouri R, Beheshti SMR, Sakr S (2015) Dream: distributed rdf engine with adaptive query planner and minimal communication. Proc VLDB Endow 8(6):654–665

13. Harth A, Umbrich J, Hogan A, Decker S (2007) Yars2: A federated repository for querying graph structured data from the web. In: Aberer K, Choi K-S, Noy N, Allemang D, Lee K-Il, Nixon L, Golbeck J, Mika P, Maynard D, Mizoguchi R, Schreiber G, Cudré-Mauroux P (eds) The semantic web. Springer, Berlin, Heidelberg, pp 211–224
14. Husain M, McGlothlin J, Masud MM, Khan L, Thuraisingham BM (2011) Heuristics-based query processing for large rdf graphs using cloud computing. IEEE Trans Knowl Data Eng 23(9):1312–1327
15. Lai L, Qin L, Lin X, Chang L (2015) Scalable subgraph enumeration in mapreduce. Proc VLDB Endow 8(10):974–985
16. Cyganiak R, Wood D, Lanthaler M (2014) RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C
17. Peng P, Zou L, Özsu MT, Chen L, Zhao D (2016) Processing sparql queries over distributed rdf graphs. VLDB J 25(2):243–268
18. Pérez J, Arenas M, Gutierrez C (2006) Semantics and complexity of sparql. In: Cruz I, Decker S, Allemang D, Preist C, Schwabe D, Mika P, Uschold M, Aroyo LM (eds) International semantic web conference. Springer, Berlin, Heidelberg, pp 30–43
19. Rohloff K, Schantz RE (2010) High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In: Programming support innovations for emerging distributed applications. ACM, p 4
20. Schätzle A, Przyjaciel-Zablocki M, Berberich T, Lausen G (2015) S2x: Graph-parallel querying of rdf with graphx. In: Wang F, Luo G, Weng C, Khan A, Mitra P, Yu C (eds) VLDB workshop on big graphs online querying. Springer, pp 155–168
21. Schätzle A, Przyjaciel-Zablocki M, Neu A, Lausen G (2014) Sempala: interactive sparql query processing on hadoop. In: International semantic web conference. Springer, pp 164–179
22. Schätzle A, Przyjaciel-Zablocki M, Skilevic S, Lausen G (2016) S2rdf: Rdf querying with sparql on spark. Proc VLDB Endow 9(10):804–815
23. Shvachko K, Kuang H, Radia S, Chansler R (2010) The hadoop distributed file system. In: 2010 IEEE 26th symposium on mass storage systems and technologies (MSST). pp 1–10. https://doi.org/10.1109/MSST.2010.5496972
24. Sun Z, Wang H, Wang H, Shao B, Li J (2012) Efficient subgraph matching on billion node graphs. Proc VLDB Endow 5(9):788–799
25. Valduriez P (1987) Join indices. ACM Trans Database Syst (TODS) 12(2):218–246
26. Yang S, Han F, Wu Y, Yan X (2016) Fast top-k search in knowledge graphs. In: 2016 IEEE 32nd international conference on data engineering (ICDE), pp 990–1001
27. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I (2010) Spark: Cluster computing with working sets. HotCloud 10(10–10):95
28. Zeng K, Yang J, Wang H, Shao B, Wang Z (2013) A distributed graph engine for web scale rdf data. In: Proceedings of the VLDB endowment, vol 6. VLDB Endowment, pp 265–276
29. Zhang X, Chen L (2017) Distance-aware selective online query processing over large distributed graphs. Data Sci Eng 2(1):2–21
30. Zou L, Özsu MT, Chen L, Shen X, Huang R, Zhao D (2014) gstore: A graph-based sparql query engine. VLDB J 23(4):565–590