GQA$_{\mathrm{RDF}}$: An Efficient SPARQL Query Answering Engine on RDF Graphs

Chu Huang [a], Qianzhen Zhang [a], Deke Guo [a,*,**] and Xiang Zhao [a] Xi Wang [a]
[a] *Science and Technology on Information Systems Engineering Laboratory, University of Defense Technology, China*
E-mails: huangchu313@163.com, 850806464@qq.com, dekeguo@nudt.edu.cn, xiangzhao@nudt.edu.cn, wangxi19@nudt.edu.cn

Abstract.
Due to the increasing use of RDF data, efficient processing of SPARQL queries over RDF datasets has become an important issue. In graph-based RDF data management solution, SPARQL queries are translated into subgraph patterns and evaluated over RDF graphs via graph matching. However, answering SPARQL queries requires handing *RDF reasoning* to model *implicit* triples in RDF data, which is largely overlooked by existing graph-based solutions. In this paper, we investigate to equip graph-based solution with the important RDF reasoning feature for supporting SPARQL query answering. In detail, we first propose an on-demand saturation strategy, which only selects an RDF fragment that may be potentially affected by the query. Then, we provide a filtering-and-verification framework to efficiently compute the answers of a given query. The framework groups the equivalent entity vertices in the RDF graph to form semantic abstracted graph as index, and further computes the matches according to the multi-grade pruning supported by the index. In order to drive and expedite query evaluation, we conceive an effective cost model for estimating the step-wise cost of query pattern matching. In addition, we show that the semantic abstracted graph and the graph saturation can be efficiently updated upon the changes to the data graph, enabling the framework to cope with dynamic RDF graphs. The results of extensive experiments on real-life and synthetic datasets demonstrate the effectiveness and efficiency of our algorithms.

Keywords: SPARQL BGP query, subgraph matching, RDFS entailment

1. Introduction

The Resource Description Framework (RDF) is a graph-based data model promoted by the W3C for modeling Web Objects as part of the prospective semantic web. Due to the simplicity and flexibility of RDF, it is now leveraged as a unified data model in a wide spectrum of applications, including bioinformatics [2], media data [3], Wikipedia [4] and social networks [5], etc. They enable machines to leverage the rich structured knowledge to better understand texts or provide intelligent services.

An RDF dataset is in essence a set of *triples*, each of the form $\langle s, p, o\rangle$ for $\langle \underline{\texttt{s}}\texttt{ubject}, \underline{p}roperty, \underline{\texttt{o}}\texttt{bject}\rangle$, where `subject` and `object` are entities or concepts and *property* is the relationship connecting them. Consequently, a collection of triples can be modelled as a *directed labeled* graph where the graph vertices denote subjects and objects while graph edges are used to denote properties, as shown in Figure 1. In order to query RDF data, the W3C recommends a formal language, namely, SPARQL. For example, to retrieve the actor in a science fiction film who won an America award, one may formulate the query using the following SPARQL:

[*] This article is an extension of our earlier published work [1] in Springer DASFAA 2020.
[**] Corresponding author. E-mail: dekeguo@nudt.edu.cn.

$$\text{SELECT } \texttt{?m} \text{ WHERE} \tag{1}$$
$$\{\texttt{?m}\ \langle won\rangle\ \texttt{?p}. \tag{2}$$
$$\texttt{?n}\ \langle hasActor\rangle\ \texttt{?m}. \tag{3}$$
$$\texttt{?m}\ \langle rdf : type\rangle\ \texttt{Actor}. \tag{4}$$
$$\texttt{?p}\ \langle rdf : type\rangle\ \texttt{America\_Award}. \tag{5}$$
$$\texttt{?n}\ \langle rdf : type\rangle\ \texttt{Science\_Fiction\_ Film}.\} \tag{6}$$

From the perspective of data management, there exist two types of solutions—relational and graph-based—to RDF data [6]. Using relational databases does not always offer an elegant solution towards efficiently RDF query evaluation, and still lacks best practices currently [7]. Recently, graph-based solution emerges, attributed to the fact that RDF is a universal graph model of data. In graph-based solution, a SPARQL query is translated into a graph pattern P, which is then evaluated over the RDF graph G. The query evaluation process is performed via matching the variables in P with the elements of G such that the returned graph is contained in G (pattern matching). To simplify presentation, we assume that the SPARQL queries originally issued at the control site are basic graph pattern (BGP) queries, since BGP queries are the building block of SPARQL queries [8]. Our solution is easily extensible to handle general SPARQL queries.

The major advantage of graph-based solution lies in that SPARQL query becomes easier to formulate without losing its modeling capability, and more importantly, graph pattern matching, without optimization strategies, is able to perform, if not better, as good as relational RDF query engines [9]. In succession, a few novel graph-based systems were put forward [9–11]. In particular, gStore [10] uses a carefully designed index VS*-tree to process RDF queries. TurboHom++ [9] transforms RDF graphs into labeled graphs and applies subgraph homomorphism methods to RDF query processing. AMbER [11] is a graph-based RDF engine that represents the RDF data and SPARQL queries into multigraphs and the query evaluation task is transformed to the problem of subgraph homomorphism.

All the aforementioned work can be summarized as graph-based efforts for RDF query evaluation (not answering) since they ignore the essential RDF feature called *entailment*, which allows model *implicit information*

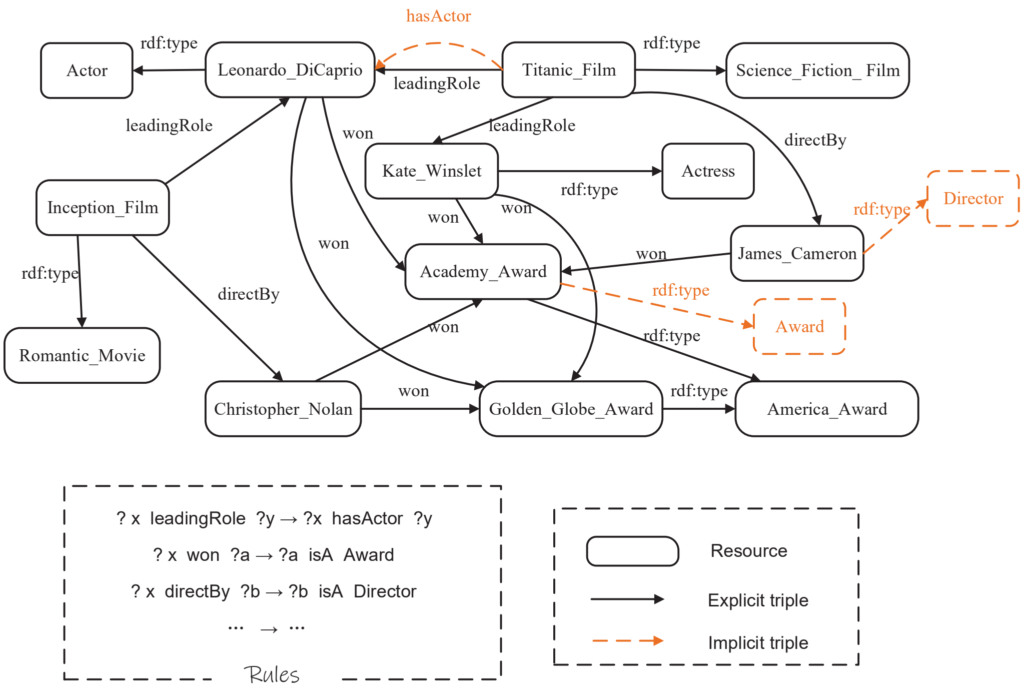


Fig. 1. Sample RDF graph

within RDF graph. Taking entailment into account is crucial, without which leads to incomplete answers [12]. For instance, assume the claim that "Titanic has an actor Leonardo DiCaprio" is not in the RDF data; nonetheless, we can derive ⟨`Titanic_Film`, *hasActor*, `Leonardo_DiCaprio`⟩, on the basis of the explicit triple ⟨`Titanic_Film`,*leadingRole*,`Leonardo_DiCaprio`⟩ and the description "`leadingRole` belongs to the subproperty of `hasActor`" in terms of RDFS Full. Here, RDFS represents an ontology language that can be used to enhance the description of RDF graphs. As a result, SPARQL query answering can be split into a reasoning step and a query evaluation step.

There are two disparate reasoning steps, i.e., *saturation* and *reformulation* [13, 14], in relational-based approaches. Saturation-based query answering exhaustively makes explicit all of the implicit information. Reformulation-based query answering performs rewriting a query into an equivalent large union of conjunctive queries and posing them against the original RDF data [15, 16]. While saturation leads to efficient query evaluation, it requires large amount of time to compute, space to store, and must be recomputed upon updates; query reformulation adversely affects query response times due to syntactically great complexity and subtle interplay between RDF and SPARQL dialects.

In this paper, we investigate to close the gap by supplementing reasoning mechanism to existing graph-based systems. Conceptually, we strike a balance between *saturation* and *reformulation*, and propose to deal with entailment by using an *on-demand saturation* strategy. That is, we need not make explicit all of the implicit data in the RDF graph, since most implicit information is irrelevant to the query; instead, we carefully choose only the RDF fragment that is revelent to the query, and then, saturate it accordingly. Based on the reasoning mechanism, we propose a filtering-and-verification framework, namely, $\mathsf{GQA}_{\mathsf{RDF}}$, for computing the answers of a given query.

Contributions. In short, the major contributions under the framework we have made are summarized below:

- We group the equivalent entity vertices in the RDF graph to form multi-grade abstracted graph as index.
- Using the index, we develop a filtering strategy, which extracts a small subgraph of G as a compact representation of the query results.
- We propose a new encoding method for further refining the candidates of each query vertex and conduct subgraph matching calculations.
- We conceive an effective cost model for estimating the stepwise cost of query pattern matching, which helps minimize the total number of intermediate results, and hence, generates a judicious matching order for driving query evaluation.
- We provide techniques to incrementally maintain the index and the graph saturation upon the changes to the RDF graph, enabling the framework to cope with dynamic data graphs.

Experiment results demonstrate that our techniques significantly outperform the state-of-the-art RDF data management system.

Organization. Section 2 formulates the problem, and an overview of the proposed framework follows in Section 3. Sections 4 introduces the offline process that build the semantic abstracted graph as index. Section 5 and 6 detail the major components of SPARQL query answering, for handling the entailment and pattern matching, respectively. Section 7 further enables the SPARQL query answering to cope with dynamic data graphs. Experimental results and analyses are presented in Section 8. Section 9 summarizes related work, and Section 10 concludes the paper.

2. Preliminaries

RDF data is a set of triples of the form $\langle s, p, o\rangle$, where s is an entity or a class, and p denotes one attribute associated to one entity or a class, and o is an entity, a class, or a literal value. We consider only well-formed triples, as per the W3C RDF standard, an entity or a class can be represented by URI (Uniform Resource Identifier). In this work, we will not distinguish between an"entity" and a "literal" since we have the same operations. As an alternative, RDF data is expressed as an RDF graph, formally defined as follows.

Definition 1 (RDF graph)**.** *An* RDF graph *is a directed labeled graph $G = (U_G, E_G, \Sigma_G, L_G)$, where U_G is a set of vertices that correspond to all subjects and objects in RDF data, $E_G \subseteq U_G \times U_G$ is the set of* directed *edges that connect the subjects and objects, Σ_G is a finite set of labels for vertices and edges, and the labeling function L_G maps each vertex or edge to a label in Σ_G. More precisely, a vertex of a subject has a label corresponding to its URI,* *while a vertex of an object can possess a label of either URI or literal. The label of an edge is its corresponding property.*

Definition 2 (RDF schema)**.** *RDF Schema (RDFS) is a valuable feature of RDF that allows enhancing the descriptions in RDF graphs. RDFS triples declare semantic constraints between the classes and the properties used in those graphs.*

Definition 3 (RDF entailment)**.** *The W3C named RDF entailment the mechanism, through which, implicit RDF triples can be derived, based on a set of explicit triples and some entailment rules.*

Table 1

Instance-level entailment

Constraints	Description	Entailment rules	Entailed triples
$\prec_{sc}$	*subclass*	$s' \prec_{sc} s''$; s *rdf:type* s'	s *rdf:type* s''
$\prec_{sp}$	*subproperty*	$p \prec_{sp} p'$; $s\ p\ s'$	$s\ p'\ s'$
$\leftharpoondown_d$	*domain*	$p \leftharpoondown_d s$; $s_1\ p\ o$	s_1 *rdf:type* s
$\rightharpoondown_r$	*range*	$p \rightharpoondown_r s$; $s_1\ p\ o$	o *rdf:type* s

In this research, we concentrate ourselves on the core entailment of RDFS regime. Using RDFS, we can recover a large amount of implicit information, part of which may be answers to queries. Specifically, Table 1 enumerates the possible RDFS constraints and the corresponding entailment rules. The first two columns show the allowed semantic constraints, and the notations to express them, where *domain* and *range* denote the first and second attribute of every property (edge label), respectively. The last two columns show the entailment rules to get the *entailed triples*. Since the overwhelming practical impact of querying only the instance-level (implicit and explicit) data, we focus on query answering only for instance-level queries (cf. Table 1).

We consider the most fundamental building block of SPARQL, which consists of (unions of) basic graph pattern (BGP) queries[1].

Definition 4 (Basic Graph Pattern)**.** *A basic graph pattern is modeled as a directed labeled query pattern graph $Q = (V_Q, E_Q, L_Q)$, where $V_Q \in V_G \cup V_{var}$ is a collection of vertices, where V_Q denotes vertices in RDF graph G and V_{var} is a set of variables; $E_Q \subseteq V_Q \times V_Q$ is a set of directed edges in Q; each edge e in E_Q has a property in L_Q. Figure 2 shows a BGP query, and the circles in it represent variable vertices.*

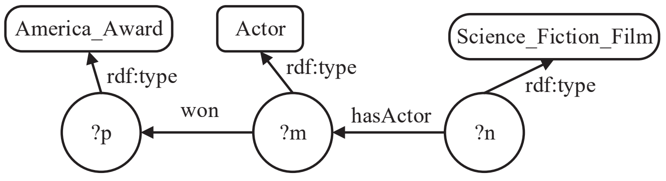


Fig. 2. An illustrate of query pattern graph

Definition 5 (BGP match over RDF graph)**.** *Consider an RDF graph G and a BGP query $Q = (V_Q, E_Q, L_Q)$ that has n vertices $\{v_1, \ldots, v_n\}$. A set of n vertices $\{u_1, \ldots, u_n\}$ in G is said to be a* BGP match*, or* embedding*, of Q, if and only if the following conditions hold:*

- *if v_i is a literal vertex, v_i and u_i have the same literal value;*
- *if v_i is an entity vertex, v_i and u_i have the same URI;*
- *if v_i is a variable vertex, there is no constraint on u_i; and*
- *if there is an edge $\langle v_i, v_j\rangle \in E_Q$ with the property p, there is an edge $\langle u_i, u_j\rangle \in E_G$ with the same property p.*

Definition 6 (Query answering)**.** *It is important to keep in mind the distinction between query evaluation and query answering. The evaluation of Q against G only use G's explicit triples to obtain the BGP matches, thus may lead to an incomplete answer set. Query answering is the evaluation of Q against G that takes the entailment into account, to obtain complete answer set. The answers of Q are constituted of returned* bindings *to query variables.*

Frequently used notations are summarized in Table 2.

[1] https://www.w3.org/TR/rdf-sparql-query/#BasicGraphPatterns

Table 2
Notations

Notations	Description
Q and G	RDF query and RDF graph
v and u	A vertex in Q and a vertex in G
$\mathbb{T}$ and $\mathbb{T}_s$	A class vertex in RDFS and a superclass vertex
G_c and $\mathcal{U}_c$	The concept graph of G and the node in G_c
G_{c_i} and $\mathcal{U}_{c_i}$	The i-th grade concept graph and the node in G_{c_i}
$\mathbb{T}_u$	The class vertex that is adjacent to u
$\mathsf{cand}(v)$	The candidates of a query vertex v
Q_T and v_r	The spanning tree of Q and the root vertex in Q_T

3. Framework

Recall that the SPARQL BGP query answering problem is a major challenge that is largely overlooked by existing graph-based efforts towards RDF data management. To this end, we provide a novel filtering-and-verification framework named $\mathsf{GQA}_{\mathsf{RDF}}$. Generally speaking, our approach consists of two stages: *offline index construction* and *online RDF query answering* (see Figure 3). We briefly review the two stages before we discuss them in details in upcoming sections.

Offline index construction. The offline process is used to build the semantic abstracted graph as index. We describe the main components. Firstly, we construct an auxiliary data structure, namely, STP, which is a series of sets that represent the semantic inclusion relation in RDFS. Then, based on STP, we merge the entity vertices in the RDF graph that is adjacent to equivalent class vertices (have equivalent type) to construct an abstracted graph as index. The index is precomputed once, and is dynamically maintained upon changes to G.

Online query processing. The online process is used to calculate the answers of a given query. Upon receiving an RDF query Q, the framework extracts a small subgraph as a compact representation of all the matches that are similar to Q, by visiting the abstracted graphs. If such a subgraph is empty, the framework determines that Q has no answers. Otherwise, we use the proposed on-demand saturation strategy to obtain the candidates of each variable vertex and conduct subgraph matching to calculate the answer. Specially, we propose a new encode module to encode the neighborhood structure around a vertex into a bitstring, and prune the candidates via "Bloom filter". Besides, in order to reduce the matching cost, which is proportional to the total number of comparisons, we propose a cost model to guide the pattern matching, and produce the answer bindings eventually.

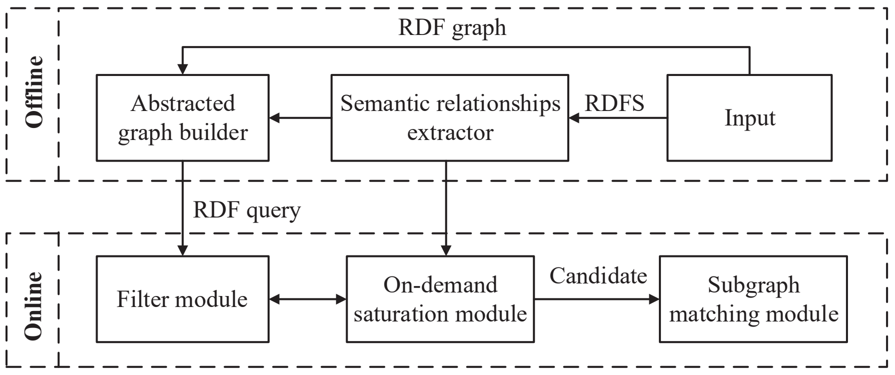


Fig. 3. Solution framework of $\mathsf{GQA}_{\mathsf{RDF}}$

4. Semantic Abstracted Graph

In this section, we propose an effective index to reduce the space cost and facilitate the query processing.

4.1. Semantic Relationships Extraction

In order to construct the *abstracted graph*, we need to group and merge the equivalent entity vertices in G, where two entity vertices are equivalent if they are adjacent to equivalent class vertex (have the equivalent type). To this end, we construct an auxiliary data structure named STP by using the semantic relation in RDFS, such that given a class vertex $\mathbb{T}$ and an entity vertex u, one can check whether u has type $\mathbb{T}$. STP is comprised of the following four sets.

- $\mathsf{SubPro}(\cdot)$: given an edge property p in RDFS, $\mathsf{SubPro}(p)$ is a set of edge properties that are the *subproperty* of p;
- $\mathsf{SubClass}(\cdot)$: given a class vertex $\mathbb{T}$ in RDFS, $\mathsf{SubClass}(\mathbb{T})$ is a set of class vertices that are the *subclass* of $\mathbb{T}$;
- $\mathsf{Domain}(\cdot)$: given a class vertex $\mathbb{T}$ in RDFS, $\mathsf{Domain}(\mathbb{T})$ is a set of edge properties that belong to the *domain* of $\mathbb{T}$; and
- $\mathsf{Range}(\cdot)$: given a class vertex $\mathbb{T}$ in RDFS, $\mathsf{Range}(\mathbb{T})$ is a set of edge properties that belong to the *range* of $\mathbb{T}$.

We extract all triples in RDFS with edge property "*rdfs:subPropertyOf*", i.e., $\langle p_1, \textit{rdfs:subPropertyOf}, p_2\rangle$ to obtain $\mathsf{SubPro}(\cdot)$,. Then, the vertex p_1 is extracted to form the set SubPro (p_2). The other three sets can be constructed in a similar flavor as the set $\mathsf{SubPro}(\cdot)$. Note that, in the STP construction process, we need to obtain corresponding superclass vertices for constructing the index.

Definition 7 (Superclass vertex)**.** *We say a class vertex $\mathbb{T}_s$ is a* ***superclass vertex*** *if there exists no other class vertex $\mathbb{T}$ such that $\mathbb{T}_s \in \mathsf{SubClass}(\mathbb{T})$.*

To achieve the superclass vertices, we use a counter $\mathsf{num}(\mathbb{T})$ (initialize to 0) for every class vertex $\mathbb{T}$ in RDFS to count the times of $\mathbb{T}$ that is extracted to construct $\mathsf{SubClass}(\cdot)$. For example, in processing a trip in RDFS with edge property "*rdfs:subClassOf*", i.e., $\langle \mathbb{T}_1, \textit{rdfs:subClassOf}, \mathbb{T}_2\rangle$, $\mathbb{T}_1$ is extracted to form the set $\mathsf{SubClass}$ $(\mathbb{T}_2)$. Then, we set $\mathsf{num}(\mathbb{T}_1) \leftarrow \mathsf{num}(\mathbb{T}_1) + 1$. Intuitively, we say a class vertex $\mathbb{T}_s$ is a superclass vertex if $\mathbb{T}_s$ has a 0 count (i.e., $\mathsf{num}(\mathbb{T}_s) = 0$). The class vertices $\{\mathbb{T}\}$ within $\mathsf{SubClass}(\cdot)$ are sorted in descending order of vertex weights $w(\mathbb{T})$ where $w(\mathbb{T}) = \frac{1}{\mathsf{num}(\mathbb{T})}$.

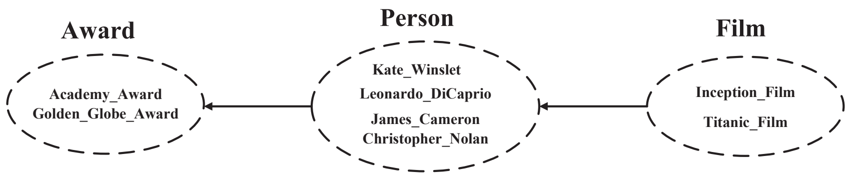
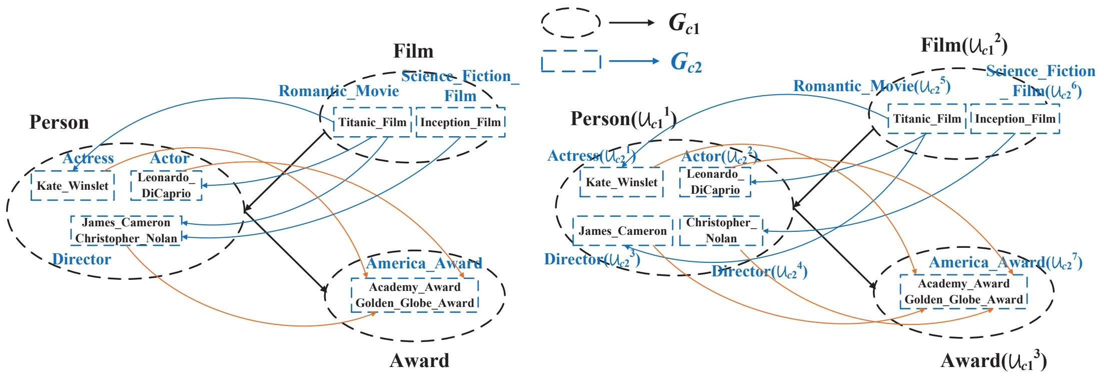


Fig. 4. Concept graph

4.2. Semantic Abstracted Graph

Relying on the semantic class constraint set in STP, we construct a semantic abstracted graph as index to reduce the space cost further.

Given an RDF graph $G = (U, E, L)$, a *concept graph* $G_c = (U_c, E_c, L_c)$ is a directed graph by ignoring edge labels. In detail, (1) U_c is a partition of U, where each $\mathcal{U}_c \in U_c$ is a set of entity vertices; (2) each $\mathcal{U}_c$ has a label $L_c(\mathcal{U}_c)$ from the superclass vertices obtained in STP, such that for any entity vertex $u \in \mathcal{U}_c$ of type $\mathbb{T}_u$, $\mathbb{T}_u \in \mathsf{SubClass}(L_c(\mathcal{U}_c))$; (3) $\langle \mathcal{U}_c^1, \mathcal{U}_c^2\rangle$ is an edge in E_c if and only if for each entity vertex u_1 in $\mathcal{U}_c^1$ (resp. u_2 in $\mathcal{U}_c^2$), there is an entity vertex u_2 in $\mathcal{U}_c^2$ (resp. u_1 in $\mathcal{U}_c^1$), such that $\langle u_1, u_2\rangle$ (resp. $\langle u_2, u_1\rangle$) is an edge in G. Here, a entity vertex u of type $\mathbb{T}_u$ means there is a class vertex $\mathbb{T}_u$ that is adjacent to u. Specially, if u has no type, we can use STP to derive corresponding type of u. To differentiate the vertices of the concept graph from the vertices of Q and G, we call vertices of the abstracted graph as ***nodes***.



G_{c1}
G_{c2}
Film
Science_Fiction_Film
Romantic_Movie
Person
Titanic_Film
Inception_Film
Actress
Actor
Kate_Winslet
Leonardo_DiCaprio
James_Cameron
Christopher_Nolan
Director
America_Award
Academy_Award
Golden_Globe_Award
Award
Film($\mathcal{U}_{c1}^2$)
Science_Fiction_Film($\mathcal{U}_{c2}^6$)
Romantic_Movie($\mathcal{U}_{c2}^5$)
Person($\mathcal{U}_{c1}^1$)
Actress($\mathcal{U}_{c2}^1$)
Actor($\mathcal{U}_{c2}^2$)
Director($\mathcal{U}_{c2}^3$)
Director($\mathcal{U}_{c2}^4$)
America_Award($\mathcal{U}_{c2}^7$)
Award($\mathcal{U}_{c1}^3$)

(a) Split-Merge process (b) Two-grade semantic abstracted graph

Fig. 5. Construction of abstracted graph

Example 1. *Figure 4 shows the concept graph G_c of the RDF graph in Figure 1. We see that Person, Film and Award are the superclass vertices. Each node $\mathcal{U}_c$ in G_c represents a set of entity vertices whose types belong to* $\mathsf{SubClass}(L_c(\mathcal{U}_c))$. *For example, in the node Film, the types of Inception_Film (i.e., $Science\_Fiction\_Film$) and $Titanic\_Film$ (i.e., Romantic_Movie) both belong to* $\mathsf{SubClass}(Film)$. *Then, consider the edge $\langle Award, Person\rangle$ in E_c. For each entity vertex u in the node Award (resp. Person), there is entity vertex u' in the node Person (resp. Award) such that $\langle u, u'\rangle$ is an edge in G.*

Definition 8 (Semantic abstracted graph)**.** *A semantic abstracted graph is multi-grade concept graph, where*

- *the first grade, $G_{c_1} = (U_{c_1}, E_{c_1}, L_{c_1})$, represents the initial concept graph constructed by using the superclass vertices;*
- *the i-th ($i \geqslant 2$) grade, $G_{c_i} = (U_{c_i}, E_{c_i}, L_{c_i})$, is a more detailed concept graph constructed from $G_{c_{i-1}}$ in the $(i-1)$-th grade by dividing each node $\mathcal{U}_{c_{i-1}}$ ($\mathcal{U}_{c_{i-1}} \in U_{c_{i-1}}$) into smaller partitions. In this case, (1) each $\mathcal{U}_{c_i}$ ($\mathcal{U}_{c_i}$ is in $\mathcal{U}_{c_{i-1}}$) has a label $L_{c_i}(\mathcal{U}_{c_i})$, which is the child-class of $L_{c_{i-1}}(\mathcal{U}_{c_{i-1}})$; (2) $\langle \mathcal{U}^1_{c_i}, \mathcal{U}^2_{c_i}\rangle$ ($\mathcal{U}^1_{c_i} \in \mathcal{U}^1_{c_{i-1}}$, $\mathcal{U}^2_{c_i} \in \mathcal{U}^2_{c_{i-1}}$) is an edge in E_{c_i} if and only if for each entity vertex u_1 in $\mathcal{U}^1_{c_i}$ (resp. u_2 in $\mathcal{U}^2_{c_i}$), there is a entity vertex u_2 in $\mathcal{U}^2_{c_i}$ (resp. u_1 in $\mathcal{U}^1_{c_i}$), such that $\langle u_1, u_2\rangle$ (resp. $\langle u_2, u_1\rangle$) is an edge in G.*

An important issue is to get the child-class vertices of a given class vertex $\mathbb{T}_u$. Recall that we can obtain the subclass vertices $\{\mathbb{T}^n_u\}$ of $\mathbb{T}_u$ based on $\mathsf{SubClass}(\mathbb{T}_u)$ in STP, each of which has a weight $w(\mathbb{T}^n_u)$. Note that, the closer $\mathbb{T}^n_u$ is to $\mathbb{T}_u$, the greater the value of $w(\mathbb{T}^n_u)$ is. As a result, we say $\{\mathbb{T}^1_u, \ldots, \mathbb{T}^i_u\}$ ($1 < i \leqslant n$) is the set of child-class vertices of t_u if they have the same and greatest value of weights in $\mathsf{SubClass}(\mathbb{T}_u)$. In specific, if $\mathsf{SubClass}(\mathbb{T}_u) = \emptyset$, we say the child-class vertex of $\mathbb{T}_u$ is itself. Figure 5(b) depicts a semantic abstracted graph of the RDF graph in Figure 1, which is also a two-grade concept graph.

Our empirical study showed that three-grade concept graph are enough for optimization. Thus, we set the grade as 3 in our experiments.

Semantic abstracted graph construction. In order to construct the semantic abstracted graph, we present an algorithm, namely, constructSAG. It first constructs the nodes set U_{c_1} as a vertex partition of G, where each node $\mathcal{U}_{c_1}$ of U_{c_1} consists of the entity vertices of type $L_{c_1}(\mathcal{U}_{c_1}) \in \mathsf{SubClass}(\mathbb{T}_s)$ (Line 1). The edge set E_{c_1} is also constructed accordingly (Line 2). It then checks the condition whether for each edge $\langle \mathcal{U}^1_{c_1}, \mathcal{U}^2_{c_1}\rangle$, each vertex u_1 (resp. u_2) in $\mathcal{U}^1_{c_1}$ (resp. $\mathcal{U}^2_{c_1}$) has a child in $\mathcal{U}^2_{c_1}$ (resp. parent in $\mathcal{U}^1_{c_1}$) (Line 4). If not, it refines U_{c_1} by splitting and merging the node $\mathcal{U}^1_{c_1}$ (resp. $\mathcal{U}^2_{c_1}$) to make the condition satisfied. The refinement process repeats until a fixpoint is reached (Line 5). G_{c_1} is updated accordingly with the new node and edge set (Line 6). In G_{c_i} ($i \geqslant 2$), it replaces the class vertices used in $G_{c_{i-1}}$ with corresponding child-class vertices and adopt the same procedure to construct G_{c_i} (Lines 7–9).

For example in Figure 5(a), nodes *Person* and *Film* are divided into a set of nodes $\{Actress, Actor, Director\}$ and $\{Romantic\_Movie, Seience\_Fiction\_Film\}$ in G_{c_2}, respectively. Since the entity vertex *Christopher_Nolan* in *Director* has no neighbor in node *Romantic_Movie*, we split the node *Director* into two nodes to produce G_{c_2} (Figure 5(b)) as the 2-*nd* grade concept graph.

Algorithm 1: constructSAG

```
Input   : $G$ is the RDF graph ; STP is the auxiliary data structure.
1 Construct partition $U_{c_1}$ of $U$ as $\{\mathcal{U}^1_{c_1}, \cdots, \mathcal{U}^n_{c_1}\}$, where $L_{c_1}(\mathcal{U}^i_{c_1}) := \mathbb{T}_{s_i}, \mathcal{U}^i_{c_1} = \{u_i | \mathbb{T}_{u_i} \in \mathsf{SubClass}(L_{c_1}(\mathcal{U}^i_{c_1}))\}$;
2 Set $E_{c_1} := \{\langle \mathcal{U}^1_{c_1}, \mathcal{U}^2_{c_1} \rangle | \langle u_1, u_2 \rangle \in E, u_1 \in \mathcal{U}^1_{c_1}, u_2 \in \mathcal{U}^2_{c_1}\}$
3 while there is a change in $U_{c_1}$ do
4     if there is an edge $\langle \mathcal{U}_{c_1}, \mathcal{U}'_{c_1} \rangle$ where $u \in \mathcal{U}_{c_1}$ has no child in $\mathcal{U}'_{c_1}$ (resp. $u' \in \mathcal{U}'_{c_1}$ has no parent in $\mathcal{U}_{c_1}$) then
5         SplitMerge $(\mathcal{U}_{c_1}, G_{c_1})$ (resp. SplitMerge $(\mathcal{U}'_{c_1}, G_{c_1})$)
6     update $G_{c_1}$;
7 foreach grade $i$ from 2 to $maxgrade$ do
8     Construct partition $U_{c_i}$ of $U$ as $\{\mathcal{U}^1_{c_i}, \cdots, \mathcal{U}^n_{c_i}\}$, where $L_{c_i}(\mathcal{U}^m_{c_i}) := \mathsf{ChildClass}(L_{c_{i-1}}(\mathcal{U}^m_{c_{i-1}}))$,
      $\mathcal{U}^m_{c_i} = \{u_m | \mathbb{T}_{u_m} \in \mathsf{SubClass}(L_{c_i}(\mathcal{U}^m_{c_i}))\}$;
9     Same as Lines 2–6;
```

Correctness and Complexity. The algorithm constructSAG correctly computes a set of concept graphs as the sematic abstracted graph. For the complexity, the time complexity of SplitMerge is $\mathcal{O}(|U_G| + |E_G|)$; and the complicity of constructing the i-th grade concept graph is $\mathcal{O}(|E_G|log|U_G|)$. Thus, the total time complexity of constructSAG is $\mathcal{O}(g * |E_G|log|U_G|)$ where g is maxgrade. As g is typically small comparing with $|U_G|$ and $|E_G|$, the overall complexity of constructSAG is thus $\mathcal{O}(|E_G|log|U_G|)$.

5. Query Pruning and Answering

In this section, we illustrate the filtering phase of the query answering framework based on the abstracted graph index, and then obtain the answers of the query by adding the on-demand saturation strategy.

5.1. Multi-grade Filtering

In order to retrieve the final answers, we need to obtain candidates for each variable vertex in the query. Instead of performing the subgraph matching directly over the RDF graph, we extract a (typically small) subgraph of G that contains all the matches of the query based on the abstracted graph.

We first search the query graph over G_{c_1}. For each variable v in Q, we can obtain the corresponding superclass $\mathbb{T}_s$ of v based on STP. Let cand(v) denote the candidates of v, which is initialized as the set of nodes labeled $\mathbb{T}_s$ in G_{c_1}. We conduct a fixpoint computation for each query edge $\langle v, v' \rangle$ (v' is not a class vertex) using cand(v) and cand(v'). Regarding each node $\mathcal{U}_{c_1} \in$ cand(v), we check if there is a node $\mathcal{U}'_{c_1}$ in cand(v') such that edge $\langle \mathcal{U}_{c_1}, \mathcal{U}'_{c_1} \rangle$ in G_{c_1} has the same direction as $\langle v, v' \rangle$. If not, $\mathcal{U}_{c_1}$ (and all the data vertices contained in it) is no longer a candidate for v, and will be removed from cand(v). Specially, if cand(v) is empty, then we can say the query Q has no answers over the RDF graph.

Multi-grade pruning. Since the semantic abstracted graph is a multi-grade concept graph, we can refine candidates by going through *i-th* ($i \geqslant 2$) grade concept graph one-by-one. For example, in the 2-*nd* grade, given a query edge $\langle v, v' \rangle$, let $\mathbb{T}_v$ and $\mathbb{T}_{v'}$ denote the types of v and v', respectively. For each node $\mathcal{U}_{c_2}$ contained in $\mathcal{U}_{c_1}$ ($\mathcal{U}_{c_1} \in$ cand(v)), we check if (1) $\mathbb{T}_v \in \mathsf{SubClass}(L_{c_2}(\mathcal{U}_{c_2}))$ (or $\mathbb{T}_v = L_{c_2}(\mathcal{U}_{c_2})$); (2) there is node $\mathcal{U}'_{c_2}$ contained in $\mathcal{U}'_{c_1}$ ($\mathcal{U}'_{c_1} \in$ cand(v')) that is adjacent to $\mathcal{U}_{c_2}$ and $\mathbb{T}_{v'} \in \mathsf{SubClass}(L_{c_2}(\mathcal{U}'_{c_2}))$ (or $\mathbb{T}_{v'} = L_{c_2}(\mathcal{U}'_{c_2})$). If not, $\mathcal{U}_{c_2}$ (and all the entity vertices contained in it) can be pruned. Note that, if the type $\mathbb{T}_v$ of v is equal to $L_{c_2}(\mathcal{U}_{c_2})$, then we will not check the query edges adjacent to v any more in larger grades concept graphs. To differentiate v from other query vertices, we use a *flag* for each query vertex (initialize to **false**) and set $flag[v] =$ **true**. Similarly, one may further refine the candidates by going through larger grades concept graphs.

Example 2. *Consider the semantic abstracted graph in Figure 5(b), and the SPARQL query graph in Figure 2. Based on STP, we can calculate the superclasses of ?p, ?m and ?n are Award, Person and Film, respectively. In G_{c_1}, we initialize cand$(?p) = \{\mathcal{U}^3_{c_1}\}$, cand$(?m) = \{\mathcal{U}^1_{c_1}\}$, cand$(?n) = \{\mathcal{U}^2_{c_1}\}$. After checking, we find all the candidate nodes satisfy the edge constraint and will not be pruned. Then, in G_{c_2}, we refine the candidates set of each variable vertex based on the child-class of each superclass used in G_{c_1}. After the refinement, cand$(?p) = \{\mathcal{U}^7_{c_2}\}$, cand$(?m) = \{\mathcal{U}^2_{c_2}\}$, cand$(?n) = \{\mathcal{U}^6_{c_2}\}$.*

Correctness and Complexity. The candidates of each variable vertex in Q is initialized using the lazy strategy contains all the possible matches. For each query edge $\langle v_1, v_2\rangle$, we use $\mathsf{cand}(v_2)$ to refine $\mathsf{cand}(v_1)$ in each grade concept graph, and only remove those nodes that are not matches (non-matches) for v_1. Since if there indeed exists a data node $\mathcal{U}_{c_i}$ that can match v, then for every query edge $\langle()v, v'\rangle$, there must exist an edge $\langle \mathcal{U}_{c_i}, \mathcal{U}'_{c_i}\rangle$ in the i-th concept graph. Thus, we only remove invalid candidates from the initial candidates of each variable vertex. The correctness thus follows.

Suppose that the semantic abstracted graph $SAG = \{G_{c_1}, G_{c_2}, \ldots, G_{c_n}\}$. The filtering process can be implemented in time $\mathcal{O}(|E_Q||SAG|)$. The construction of candidates of Q is in time $\mathcal{O}(|SAG|)$. Thus, the total time is in $\mathcal{O}(|E_Q||SAG|)$. In practice E_Q is typically small, and the TOTAL complexity can be considered as near-linear $w.r.t.|SAG|$.

5.2. On-demand Saturation

To obtain complete answers of the query, in this section, we present an on-demand saturation strategy, which consists of two stages: edge property saturation and entity type saturation.

Edge property saturation. Edge property saturation is used to check whether a data edge can match a query edge with respect to property, either directly or via entailment. That is, if a data edge has a different property from a query edge, we examine the subproperties entailed by the data edge, to see if any of them matches the query edge.

To this end, let $\langle v, v'\rangle$ be an outing going edge labeled p_v adjacent to v. For each candidate entity vertex u in $\mathsf{cand}(v)$, we check whether there exists an outgoing edge $\langle u, u'\rangle$ labeled p_u adjacent to u such that $p_u = p_v$ or $p_u \in \mathsf{SubPro}(p_v)$. If not, u will be pruned from $\mathsf{cand}(v)$. Otherwise, if $p_u \in \mathsf{SubPro}(p_v)$ and there is no other outing edge adjacent to u with the property p_v, we add the outgoing edge $\langle u, u'\rangle$ labeled p_v into u.

Entity type saturation. Entity type saturation is used to check if a entity vertex matches a query vertex with respect to type in the query graph.

Given a variable vertex v of type $\mathbb{T}_v$ s.t. $flag[v] = \textbf{false}$, for each entity vertex u in $\mathsf{cand}(v)$, we check if one of the following three conditions hold: (1) $\mathbb{T}_u \in \mathsf{SubClass}(\mathbb{T}_v)$ where $\mathbb{T}_u$ is the type of u; (2) there exists an outgoing edge $\langle u, u'\rangle$ labeled p_u adjacent to u such that $p_u \in \mathsf{Domain}(\mathbb{T}_v)$; (3) there exists an incoming edge $\langle u', u\rangle$ labeled p_u adjacent to u such that $p_u \in \mathsf{Range}(\mathbb{T}_v)$. If not, u will be pruned from $\mathsf{cand}(v)$.

5.3. RDF Query Answering

Note that, in the filtering process, we ignore the edge property information for each query edge. In this section, we use the *neighborhood encoding* technology to further prune invalid candidates.

Neighborhood encoding. Neighborhood encoding has been widely adopted to assist various operations in managing RDF data [17], which describes each vertex as a bit string, namely, *vertex signature*. In a similar flavor, we choose to encode, for each vertex in RDF graph, its adjacent edge properties and the corresponding neighbor vertex properties into bit strings via *Bloom filter* [18].

Let $\langle u, u'\rangle$ labeled p_u be an adjacent edge of an entity vertex u in G, m the length of p_u's bit string, n the length of u''s bit string. Bloom filter uses a set of hash functions H to set $\overline{m}$ out of m bits to be "1", and set $\overline{n}$ out of n bits to be "1", where $\overline{m}$ and $\overline{n}$ represent the number of independent hash functions, respectively. The bit string of u, denoted by $Bit(u)$, is formed by performing *bitwise OR* operations over all it's adjacent edge bit strings. Note that given a variable vertex v, if the adjacent neighbor of v is also a variable vertex, we set the bit string of the vertex with all "0" (same as variable edge). u is a candidate of v only if $Bit(v)$ & $Bit(u) = Bit(v)$, where '&' is the *bitwise AND* operator.

Out-edge	Out-vertex	Out-edge+vertex	In-edge	In-vertex	In-edge+vertex

Fig. 6. Bit string of a vertex

The encoding method in [10] divides the bit string of the vertex into two parts: the first part represents the outgoing edge properties information, while the second represents the properties information of linked neighbors. Such method can be insufficient in fully harness the neighborhood information for candidate pruning. In this connection, we propose to encode the neighborhood of a vertex using six parts, as depicted in Figure 6. The first two parts describe the information of outgoing edges information and linked vertices. In the third part, we bind each edge with the neighbor corresponding to it. The last three parts are the information about incoming edges, which are processed in a similar manner as for outgoing edges. In order to avoid the “false drop" problem that may exist in the encoding method, we follow the method in [10] to set the length of each part as 100 and use 3 different hash functions.

Then, upon receiving the final concise candidates set of each query vertex, we conduct subgraph homomorphism calculations to obtain the answers of the query. Since the matching order selection is one of the most important parts of the subgraph matching, we propose an effective cost model for estimating the stepwise cost of query pattern matching, which helps minimize the total number of intermediate results, and hence, generates a judicious matching order for driving query evaluation. More details see Section 6.

6. Cost-driven Pattern Matching

In this section, we intend to harvest BGP matches (or embedding) to the SPARQL query Q by leveraging the candidates set of each vertex in Q. We are in quest of boosting performance by conducting exploration on the corresponding candidates sets.

Standard backtracking is viable but inefficient, which neglects the matching order that may greatly affect the performance. The matching order has been a focused theme in general graph pattern matching [19], which also applies to RDF graphs [9]. The key problem to be addressed is how to reduce the total number of comparisons performed in the candidate verification. Analogous to existing work, we present a novel prediction-based approach to decide an appropriate order for matching vertices.

6.1. Cost model

We use a cost model to guide the choice of next vertex in matching order. Akin to existing work [20], our model follows to be established on the basis of a spanning tree Q_T of the query pattern Q. First and foremost, the first decision is to choose a root vertex, and based on the root vertex spanning trees can be derived. Edges in Q_T are called *tree* edges, and those not in Q_T are *non-tree* edges.

We develop the cost model as follows. If we have a matching order $(v_1, v_2, \ldots, v_n)$ for all vertices of Q, the total number of comparisons (mapping cost) performed in a backtracking algorithm for matching Q is

$$T_{hom} \triangleq T_{|V_Q|} = |M_1| + \sum_{i=2}^{|V_Q|} \sum_{j=1}^{|M_{i-1}|} d_i^j \cdot (r_i + 1), \tag{7}$$

where M_i represents the set of matching results for the subgraph of Q that is induced by $(v_1, v_2, \ldots, v_i)$, d_i^j is the number of vertices in $\mathsf{cand}(v_i)$ joinable with a (partial) match in M_{i-1}, and r_i is the number of non-tree edges between v_i and vertices before v_i in the matching order. In addition, $u \in \mathsf{cand}(v_i)$ is a successful mapping of v_i, if it satisfies all connection constraints specified by the non-tree edges of v_i (r_i in total).

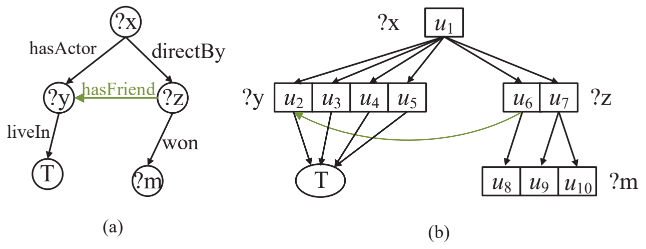


Fig. 7. Sample query pattern and candidates

Example 3. *Consider the sample query pattern Q in Figure 7(a), where edges in black are tree edges, and the one in green is a non-tree edge. Figure 7(b) depicts the candidates of every vertex. Assume we are looking at a matching* *order by $(?x, ?y, T, ?z, ?m)$ for Q, the total number of comparisons, according to Equation (7), can be derived by* $1 + 4 + 4 + 8 \times 2 + 1 = 26$. *In comparison, if we use another order by $(?x, ?z, ?y, T, ?m)$, the total number of comparisons is* $1 + 2 + 8 \times 2 + 1 + 1 = 21$.

The above example unveils that matching order is vital to a performant RDF query evaluator. In the sequel, we present an efficient algorithm to obtain a good and practical order.

6.2. Heuristic approximation

It is noted from Equation (7) that the number of non-tree edges, i.e., r_i between v_i and vertices before v_i in the matching order, largely depends on the actual matching order. The total number of configurations of r_i is exponential in $O(|V_Q|!)$. It is prohibitively expensive to optimize T_{iso} on the fly while involving r_i. As a consequence, we choose to minimize T_{iso} in a greedy manner. That is, every time when matching a vertex of Q, we expect a vertex producing the minimum number of comparisons on the basis of current intermediate results.

Intuitively, for the root vertex, we favor vertex that is of a relatively small number of candidates and a large degree. Thus, the vertex with the minimum value of $\frac{|\mathsf{cand}(v)|}{deg(v)}$ is chosen as the root v_r, where $deg(v)$ is the total degree of v. Subsequently, given a set of vertices adjacent to v_r, the next vertex is the one that together with v_r requests the minimum number of comparisons. Nevertheless, it is still fairly difficult to precisely calculate the actual results of every vertex together with the intermediate matching results. To further simplify the calculation, when we match a vertex v_i, the number of comparisons together with v_i may be estimated by

$$T'(v_i) = |M_{i-1}| \cdot |\mathsf{cand}(v_i)| \cdot \mathcal{P}(p_i) \cdot (r_i + 1). \tag{8}$$

Here, p_i is the edge connecting v_i and v_i's parent vertex in the spanning tree, $\mathcal{P}(p_i)$ denotes the probability that a triple satisfies the constraint of the edge, i.e., the ratio of number of candidate edges satisfying the connection constraint of p_i over total number of edges in G [2]. We exemplify the calculation using Example 4.

Example 4. *Consider the query pattern in Figure 7, and suppose that the vertices $?x$ and $?z$ have been matched. At this time, the number of intermediate results is 2, and we are going to choose the next vertex. If we choose $?m$, the number of comparisons is* $1 + 2 = 3$*; if we choose $?y$, the number of comparisons is* $8 \times 2 = 16$. *According to the greedy selection, we choose $?m$ as the next vertex, and the current total number of comparisons is* $1 + 2 + 3 + 12 \times 2 + 1 = 31$.

The cost estimation in Equation (8) is greedy in the sense that it only considers the cost thus far, but ignores the cost from remaining unmatched vertices. If we choose to match $?y$ (rather than $?m$), we could have a smaller number of 21 comparisons. This means the approximation needs to be amended, and this situation recalls a heuristic strategy used in the classic A* algorithm. Analogously, we divide the cost of matching a vertex into two components: 1. the cost incurred by the current vertex (Equation (8)); and 2. the estimated future cost from the current vertex.

In order to estimate the future cost, we need to anticipate the number of intermediate matching results afterwards and the number of candidates of unvisited vertices. Note that, it is very difficult to precisely calculate the actual intermediate results after mapping v_i. Thus, we use an approximate method that estimates the number of results as

$$|M_i| = |M_{i-1}| \cdot |\mathsf{cand}(v_i)| \cdot \prod_{j=0}^{n-1} \mathcal{P}(p^i_j), \tag{9}$$

where p^i_j represents the j^{th} edge adjacent to v_i that links to the visited vertex. Then, we estimate the number of candidates for the unvisited vertices. Let v_k be an unvisited vertex, the number of candidates for v_k may be represented as $|\mathsf{cand}(v_k)| \times \prod \mathcal{P}(p_k)$, where p_k represents an edge adjacent to v_k. The total number of candidates for the unvisited vertices is the sum of every candidate results of unvisited vertex. Based on intermediate results and candidates for unvisited vertices, the future cost of mapping a vertex v_i is estimated as

$$T''(v_i) = |M_i| \cdot \sum \left[|\mathsf{cand}(v_k)| \times \prod \mathcal{P}(p_k) \right]. \tag{10}$$

[2] If p_i is a variable, $\mathcal{P}(p_i)$ equals number of candidate edges adjacent to M_{i-1} divided by total number of edges.

As a result, the cost of mapping v_i, i.e., $T_{iso++}(v_i)$, is estimated as $T_{hom++}(v_i) = T'(v_i) + T''(v_i)$, which is close to the real cost (minimum cost) in pattern matching.

Correctness and complexity. Based on the discussion, we can implement a procedure for choosing the next vertex for matching, namely, minCostVertex. Note that, we will choose ?y first if we use the new cost model in Example 4 which may bring fewer cost. While the details of the procedure is omitted in the interest of space, it can be seen that the procedure runs in $O\left(|E_G| \times |V_Q| \times |E_Q|\right)$. In the matching process, we ignore the type vertices in Q since these vertices have been used to prune candidates in entity type saturation step (see Section 5.2).

Example 5. *Continuing Figure 7, suppose ?x and ?z have been visited and ?y is the vertex to be mapped at this step. The intermediate matches* $M = \{(u_1, u_6), (u_1, u_7)\}$ *and* $\mathsf{cand}(?y) = \{u_2, u_3, u_4, u_5\}$. *There are two visited vertices* $?x$ *and* $?z$ *that are adjacent to ?y. Here,* (u_1, u_7) *will be removed from M since there are no edges linking the vertices in* $\mathsf{cand}(?y)$ *and* u_7. *As a result, we can get* $M' = \{(u_1, u_6, u_2)\}$.

Remark. In comparison with existing cost models for pattern matching and order selection, the proposed model and algorithm are advantageous in the sense that

- As identified by existing work [20], TurboHom++ [9] fails to be applicable to large and complex query patterns; in contrast, $\mathsf{GQA}_{\mathsf{RDF}}$ lends itself to large and complex BGP queries against the more difficult matching criteria of subgraph homomorphism;
- Compared with QuickSI [21], which merely concentrate on a local cost with a greedy strategy, our proposed cost model generates a more effective matching order, which takes both existing and future costs into account, and hence, reduces a large number of unpromising intermediate results;
- In comparison with CFL [20], which implements a path-based cost model, our model chooses an edge-based cost most, and thus, is more flexible and less computationally expensive, while retaining the quality of order selection.

It can be seen that the cost-driven matching algorithm heavily relies on a good estimation of $\mathsf{cand}(\cdot)$, and the more accurate estimation, the better guidance for matching ordering. In this paper, we strive to offer a good estimation of candidates by levering an online saturation strategy with index support.

7. Rationale of Maintenance

In practice the data graphs are changing frequently over time. In this section, we investigate the incremental maintenance of the semantic abstracted graph index and the graph saturation, which further enables the RDF query answering to cope with dynamic data graphs.

7.1. Index Maintenance upon updates

Instead of recomputing the semantic abstracted graph and the saturation from the scratch each time the RDF graph is updated, we relay on an incremental maintain strategy.

Handling Edge Insertions. Consider an edge $\langle u, u' \rangle$ inserted into G, we take a *split-merge-propagation* strategy for each grade in the abstracted graph as follows. In the 1-*st* grade, we first identify $\mathcal{U}_{c_1}$ and $\mathcal{U}'_{c_1}$ in G_{c_1} that contains u and u', respectively. We then separate u' from $\mathcal{U}'_{c_1}$, and *split* $\mathcal{U}_{c_1}$ similarly if $\mathcal{U}_{c_1}$ and $\mathcal{U}'_{c_1}$ violate the structural constraints of a concept graph due to the edge insertion. Next, we check whether the separated data vertices can be *merged* into other nodes in G_{c_1}, due to satisfying the edge constraints. Since the updates of nodes $\mathcal{U}_{c_1}$ (resp. $\mathcal{U}'_{c_1}$) may *propagate* to its adjacent nodes, we should further check the neighbor nodes of $\mathcal{U}_{c_1}$ (resp. $\mathcal{U}'_{c_1}$) in the same way until there is no update in G_{c_1}. Similarly, after updating G_{c_1}, we update G_{c_i} ($i \geqslant 2$) following the same *split-merge-propagation* strategy.

Handling Edge Deletions. Consider an edge $\langle u, u' \rangle$ deleted from G, we take a similar operations as the updating procedure of edge insertions. After processing the changes directly caused by the edge deletion, it propagates the changes, following the same *split-merge-propagation* strategy. In the 1-*st* grade, we first identify $\mathcal{U}_{c_1}$ and $\mathcal{U}'_{c_1}$ in G_{c_1} that contains u and u', respectively.And then identifying whether $\mathcal{U}'_{c_1}$ still has the child vertex of u (resp. whether $\mathcal{U}_{c_1}$ still has the parent vertex of u'). If not,we separate u form $\mathcal{U}_{c_1}$ (reap. u' form $\mathcal{U}'_{c_1}$), and then check whether the separated data vertices can be merged into other nodes in G_{c_1}

7.2. Saturation Maintenance upon updates

To maintain the saturation efficiently, an important issue is to keep track of the multiple ways in which an edge was entailed. This is significant when considering both implicit data and updates: for a given update, we must decide whether this adds/removes one reason why a triple belongs to the saturation. A naïve implementation would record the inference paths of each implied triple, that is, all sequences of reasoning rules that have lead to that triple being present in the saturation. However, the volume of such justification grows very fast and thus the approach does not scale. Instead, we chose to keep track of the *number of reasons* why an edge has been inferred. In subproperty saturation, the number of reason is 1 since an implied edge only entailed by one explicit edge. In entity type saturation, for each data vertex u in $\mathsf{cand}(v)$, we use the notation $Type(u)$ to record the number of reasons that can entail u has the same type as v. Then, for a given edge insertion (resp. edge deletion), we will decide whether this adds (resp. deletes) one reason why an type edge belongs to the saturation. When this count reaches 0, the implied type edge should be deleted.

8. Experiments

In this section, we evaluate our method over both real and synthetic datasets and compare with te state-of-the-art algorithms. Both real and synthetic data are used to evaluate $\mathsf{GQA_{RDF}}$'s performance. The synthetic data is used to study its scalability.

8.1. Experiment setup

The proposed algorithms were implemented using C++, running on a Linux machine with two Core Intel Xeon CPU 2.2Ghz and 32GB main memory. Particularly, three algorithms were implemented: (1) $\mathsf{GQA_{RDF}}$, our algorithm; (2) TurboHom++, which extends existing subgraph homomorphism method to handle SPARQL queries [9]; (3) gStore, which tags each vertex with a signature and match signatures of data vertices and pattern vertices one by one [10]; (4) CFL, the state-of-the-art subgraph matching algorithm that is used to evaluate the efficiency of our proposed cost model.

Experiments were carried out on real-life RDF and synthetic datasets (as shown in Table 3). For query evaluation,

Table 3
Graph datasets

Dataset	Edge	Predicate	Entity
Yago	20,263,756	21,843	2,218,624
LUBM10M	12,237,135	18	1,684,231
LUBM20M	25,124,227	18	3,243,658
LUBM30M	32,457,671	18	4,752,538

we choose to use the SPARQL BGP queries in [22] over Yago and use the SPARQL BGP queries in [23] over LUBM, each of which has six queries ($Q_1 \sim Q_6$).

We measure and evaluate (1) the effectiveness of the on-demand saturation strategy; (2) the efficiency and scalability of the query answering framework; (3) the effectiveness of our proposed encoding method and the efficiency of our cost model; and (4) the performance and cost of the semantic abstracted graph index.

8.2. Evaluating the effectiveness of on-demand saturation

In this subsection, we evaluate the effectiveness of our on-demand saturation technology, which is scaled by the number of match results. For the sake of simplicity, we use Q_i^y to represent the query Q_i in Yago, and use Q_i^l to represent the query Q_i in LUBM. We ran experiments with both datasets and report the results obtained for all queries. The conclusions are reported below.

Table 4 shows the total number of match results. It is not surprising to notice that $\mathsf{GQA_{RDF}}$ can get more complete match results for almost all the queries than gStore. Especially, in Q_2^y, Q_4^y and Q_3^l, the number of match results

Table 4

Match results

Queries	Yago		LUBM 10M	
	gStore	GQA$_{RDF}$	gStore	GQA$_{RDF}$
Q_1	1,638	3,271	211	495
Q_2	0	1,063	2,201	6,731
Q_3	397	1,817	0	4,062
Q_4	0	18,849	1,336	1,849
Q_5	125	428	29	231
Q_6	863	1,093	784	784

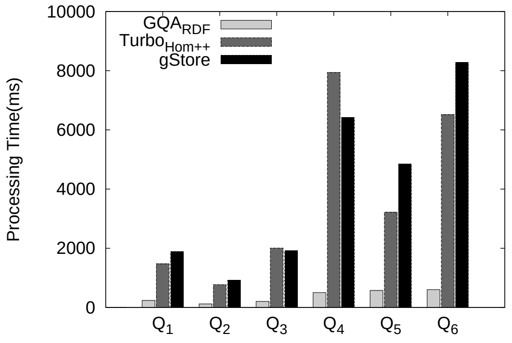
is 0 if we use gStore. This is because in Q_2^y, the edge label "*placedIn*" does not exist in original RDF graph, however, GQA$_{RDF}$ can use the constraint $isLocatedIn \prec_{sp} placedIn$ to get the entailed triples which satisfy the query. Similarly, in Q_4^y and Q_3^l, some edges in pattern graph but not in RDF graph will be entailed, and added to the RDF graph to get more match results. In general, the comparisons verify the effectiveness of our proposed on-demand saturation strategy.

8.3. *Evaluating the efficiency and scalability of GQA$_{RDF}$*

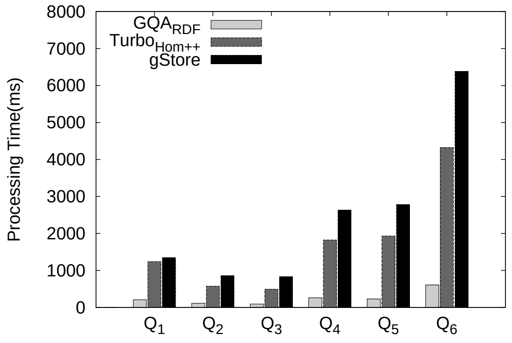
We evaluated the performance of GQA$_{RDF}$, gStore and TurboHom++ using both Yago and LUBM, and their scalability using LUBM. In these experiments, the indexes were precomputed, and thus their construction time were not counted. Note that, gStore and TurboHom++ cannot handle SPARQL query answering since they ignore the essential RDF feature called entailment. As a result, we adopt the *reformulation* reasoning strategy, and **rewrite** the queries that are used in gStore and TurboHom++ to directly compute all the answers.

Query answering time. Figure 8(1) and Figure 8(2) show the query answering time for each RDF query graph over Yago and LUBM, respectively. Since TurboHom++ needs offline process for transforming the RDF graph into labeled graph and gStore needs offline process for building the VS*-tree index, we only consider the online performance for each competitors. GQA$_{RDF}$ consistently outperforms its competitors. This is due to our on-demand saturation strategy that can avoid large amounts of subgraph matching calculations for rewritten queries. Specially, in Yago, GQA$_{RDF}$ outperforms TurboHom++ by up to 14.89 times (see query Q_4^y), gStore by up to 13.75 times (see query Q_6^y); in LUBM, GQA$_{RDF}$ outperforms TurboHom++ by up to 8.32 times (see query Q_5^l), gStore by up to 12.16 times (see query Q_5^l). Note that, in most cases, gStore has the worst performance, since it traverses the RDF graph in a BFS order, which will produce redundant Cartesian products.

Evaluating the scalability. Figure 9 shows the performance results of GQA$_{RDF}$ against existing algorithms regarding the scalability by using LUBM for varying the dataset size. Here, we vary the size of the RDF graph from



(1) Yago



(2) LUBM10M

Fig. 8. Comparison of overall efficiency

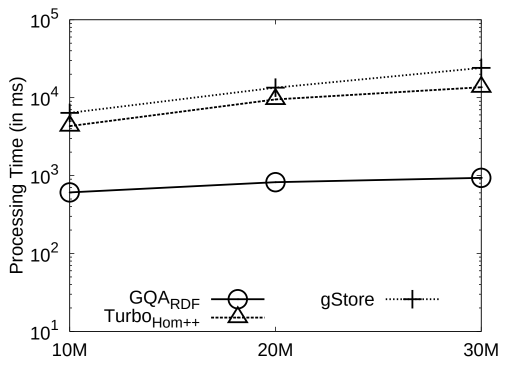


Fig. 9. Comparison of scalability

12,237,135 (LUBM10M) to 32,457,671 (LUBM30M). We use Q_{6b} since the performance gap is largest at this case. It reveals that $\mathsf{GQA_{RDF}}$ consistently outperforms its competitors regardless of the dataset size. In generally, the scalability suggests that $\mathsf{GQA_{RDF}}$ can handle reasonably large real-life graphs as those existing algorithms for deterministic graphs. Specially, $\mathsf{GQA_{RDF}}$ outperforms TurboHom++ by up to 14.73 times and gStore by up to 26.02 times.

8.4. *Comparison of partial performance*

In this subsection, we compare the partial performance with baselines on Yago dataset, including: 1. the pruning power of the new encoding method; and 2. the efficiency of the new cost model. Note that, when comparing the encoding method, there may be many variable vertices that need to retrieve candidates, and we select the vertex with the maximum neighbors to be displayed.

Evaluating the encoding method. The result is shown in Figure 10. As expected, the encoding method in this paper is no worse than gStore system. Note that in Q_6, using the new encoding method can reduce the number of candidates by a factor of 100. However, for the vertex with a large number of candidates, i.e., Q_5, there is no significant improvement in pruning power. This is because the minimum number of candidates is already very large, and only the outgoing edges of the chosen vertex are enough to obtain the minimum number of candidates in these cases, even if we consider the *local features* of the chosen vertex precisely rather than using Bloom Filter. In such cases, the unpromise candidates can only be moved in pattern matching process.

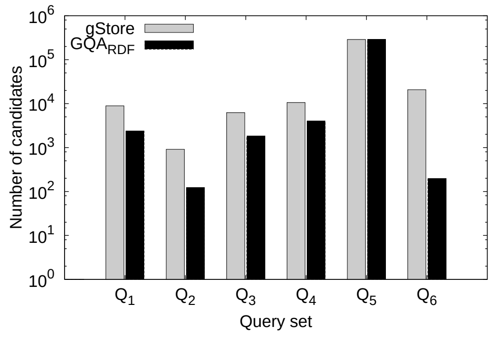


Fig. 10. Number of candidates

Evaluating the cost model. Since the size of candidates is also a key factor affecting the running time despite matching order, for a fair comparison, we choose to use the same candidate set for every variable vertex in both solutions. Here, we use the candidates produced by $\mathsf{GQA_{RDF}}$ and plot the running time in Figure 11. It is revealed that the time cost in $\mathsf{GQA_{RDF}}$ is never higher than that in CFL. We observe that in Q_5, if we use the matching order that our cost model produced, it can help lower the time cost by a factor of 10. This is because CFL does not consider

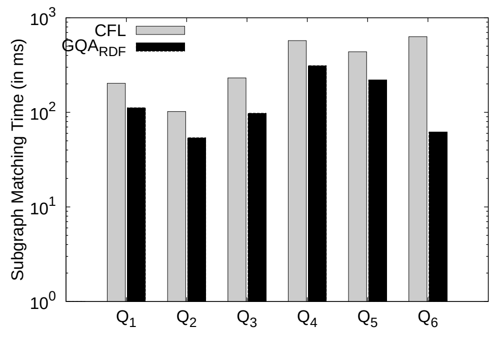


Fig. 11. Join cost

the future costs, which will produce unpromising intermediate match results; and $\mathsf{GQA_{RDF}}$ selects the vertex with minimum immediate results with the aim to postpone cartesian products. In a word, our $\mathsf{GQA_{RDF}}$ demonstrates advantage over CFL as the proposed cost model can work well for normal queries.

8.5. *Evaluating the effectiveness of semantic abstracted graph*

Using synthetic and real-life datasets, we next investigate (1) the index building cost of $\mathsf{GQA_{RDF}}$ and its competitors, including time cost and physical memory; (2) the memory reduction $\mathsf{mr} = \frac{|M_I|}{|M|}$, where $|M_I|$ and $|M|$ are the physical memory cost of the index and the data graph, respectively; (3) the filtering rate $\mathsf{fr} = \frac{|G_{sub}|}{|G|}$, where $|G_{sub}|$ is the average size of the induced subgraphs in the filtering phase, and G is the size of G. The result is shown below.

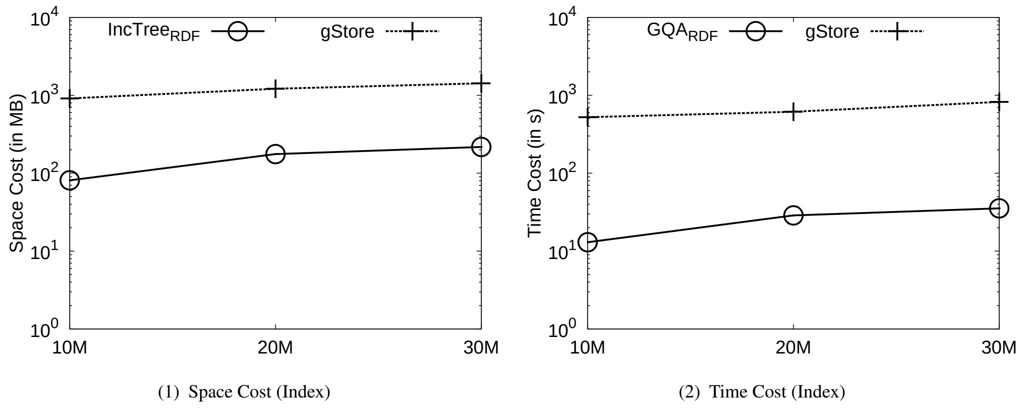


Fig. 12. Evaluation of the offline performance

Figure 12(1) and Figure 12(2) show the space cost and time cost of index construction using LUBM, respectively. Since TurboHom++ does not construct any index, we only compare $\mathsf{GQA_{RDF}}$ with gStore. We see that $\mathsf{GQA_{RDF}}$ has consistently better performance than its competitors regardless of memory and time. What's more, the figure reads a non-exponential increase as the data size grows. In specific, $\mathsf{GQA_{RDF}}$ outperforms gStore by up to 11.24 times and 40.31 times, in terms of the memory cost and time cost, respectively.

Table 5

Effectiveness of index

Dataset	$\mathsf{GQA_{RDF}}$		gStore	
	mr	fr	mr	fr
Yago	0.43	0.13	0.64	0.27

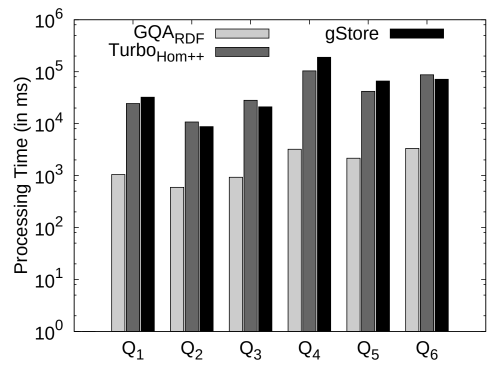


Fig. 13. Comparison of dynamic RDF graph

Table 5 gives the effectiveness of the index using Yago. It reveals: (1) GQA_{RDF} beats gStore regardless of mr and fr; (2) the semantic abstracted graph contains much less nodes and edges over the RDF graph, and takes less than half of its physical memory cost; (3) using semantic abstracted graph can effectively filter the search space, that is, the size of G_{sub} for verification is only 17% over Yago.

We finally compare the performance of GQA_{RDF} and its competitors upon RDF graph changes. We use Yago dataset and fix edge insertions $|E_I| = 1,376,286$. Since updating one edge at a time is too slow for TurboHom++ and will reach the timeout (1-hour) for all queries. As a result, we insert edges in batches of 100K ($= 100 \times 10^3$) for it. Figure 13 tells us that GQA_{RDF} greatly outperforms its competitors. Specially, GQA_{RDF} performs TurboHom++ by up to 32.19 even the edge updates are inserted in batches for TurboHom++.

9. Related Work

Due to the increasing use of RDF data, many well-known RDF stores based on relation come to the fore, including relational-based stores and graph-based stores.

Relational-based RDF query evaluation. Relational-based RDF stores use relational models to store RDF data and translate SPARQL queries into relational algebraic expressions. It, however, need too many join operations. SW-Store [24] uses a column-oriented store as its underlying store, triples are stored as sorted by the subject column. RDF-3X [6, 25] and Hexastore [26] model RDF triples as big three-attribute tabular structures and build six clustered clustered B+-trees as indexes for each permutation of subject, predicate and object. H-RDF-3X [27] is a distributed RDF processing engine where RDF-3X is installed in each cluster node.

Graph-based query evaluation. Graph-based stores use graph traversal approaches, i.e., subgraph homomorphism, and graph indexing. TurboHom++ [9] eliminates corresponding query vertices/edges from a query graph by embedding the types of an entity into a vertex label set to boost query performance. GRIN [28] uses graph partitioning and distance information to construct the index for graph queries. Its index is a balanced binary tree with each of its nodes containing a set of triples. gStore [10] tags each vertex with a signature and matches signatures of data vertices and query vertices by using the VS*-tree index. Grass [29] performs the graph pattern matching by the concept of fingerprint for star subgraph, which can describe a subgraph and be used as a filter to help to prune search space.

However, above methods ignore the essential RDF feature called *entailment*, which allows modeling *implicit information* through some inference schemes. As a result, leading to incomplete answers. In general, inference schemes are inherently divided into two main approaches, forward-chaining and backward-chaining. Forward-chaining generates all the implicit triples based on *graph saturation* (*closure*) and add them to the database. For example, 3store [13], Jena [30], OWLIM [14], Sesame [31] support saturation-based query answering, based on (a subset of) RDF entailment rules. The work by Goasdou*é* et al. [12] extends above studied by the support of blank nodes. While saturation leads to efficient query process, it needs time to be computed iteratively until a stopping criterion is matched and space to be stored. The Backward-chaining approach performs inference at query time, rewriting queries and posing them against the original data. Algorithms in [32] consider some novel rules to reformulate relational conjunctive queries. However, the query-specific backward-chaining techniques adversely affect query response times due to live inference. As for graph-based SPARQL query processing, the system could return more answers that match the query based on the semantic similarity. The work by Zheng et al. [33] proposes an instance-driven approach to automatically discover the diverse structure patterns conveying equivalent semantic meanings from a large RDF graph. However, the answers at this condition are inexactly.

To our best knowledge, there is no research literature directly targeting *entailment* on graph-based RDF query answering. In this paper, we striking the trade-off between forward- and backward-chaining and propose to deal with entailment via on-demand saturation. That is, we carefully choose only the RDF fragment that is revelent to the query to get exact and complete answers.

10. Conclusion

In this paper, we have studied *graph-based approach for efficient query answering*. We devise $\mathsf{GQA}_{\mathsf{RDF}}$ to provide effective support. On top of it, we propose an on-demand saturation strategy, which only selects an RDF fragment that may be potentially affected by the query. In addition, we devise a semantic abstracted graph index for discovering candidate vertices, which brings a constant-time reduction of candidate search space. The semantic abstracted graph and the graph saturation can be efficiently updated upon the changes to the data graph. Most importantly, a cost model is proposed, which takes both existing and future costs into account, and hence, reduces a large number of unpromising intermediate results. The proposed cost model Finally, comprehensive experiments performed on real and benchmark datasets demonstrate that $\mathsf{GQA}_{\mathsf{RDF}}$ outperforms its alternatives.

There are many directions that we intend to follow. In this work, we do not consider the blank nodes in the RDF graph. In the future, we will take the blank nodes into account. Others include support for partitioned RDF repositories, parallel execution of SPARQL queries, and further query optimization techniques.

References

[1] X. Wang, Q. Zhang, D. Guo, X. Zhao and J. Yang, sf $\mathsf{GQA}_{\mathsf{sf\ RDF}}$: A Graph-Based Approach Towards Efficient SPARQL Query Answering, in: *Database Systems for Advanced Applications - 25th International Conference, DASFAA 2020, Jeju, South Korea, September 24-27, 2020, Proceedings, Part II*, 2020, pp. 551–568.

[2] F. Belleau, M. Nolin, N. Tourigny, P. Rigault and J. Morissette, Bio2RDF: Towards a mashup to build bioinformatics knowledge systems, *Journal of Biomedical Informatics* **41**(5) (2008), 706–716.

[3] G. Kobilarov, T. Scott, Y. Raimond, S. Oliver, C. Sizemore, M. Smethurst, C. Bizer and R. Lee, Media Meets Semantic Web - How the BBC Uses DBpedia and Linked Data to Make Connections, in: *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings*, 2009, pp. 723–737.

[4] F.M. Suchanek, G. Kasneci and G. Weikum, Yago: a core of semantic knowledge, in: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, 2007, pp. 697–706.

[5] P. Mika, Social Networks and the Semantic Web, in: *2004 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2004), 20-24 September 2004, Beijing, China*, 2004, pp. 285–291.

[6] T. Neumann and G. Weikum, The RDF-3X engine for scalable management of RDF data, *VLDB J.* **19**(1) (2010), 91–113.

[7] S. Sakr, M. Wylot, R. Mutharaju, D.L. Phuoc and I. Fundulaki, *Linked Data:Storing, Querying, and Reasoning*, Springer, Cham, 2018. ISBN 978-3-319-73514-6.

[8] M. Wylot, M. Hauswirth, P. Cudré-Mauroux and S. Sakr, RDF Data Storage and Query Processing Schemes: A Survey, *ACM Comput. Surv.* (2018), 84:1–84:36. doi:10.1145/3177850.

[9] J. Kim, H. Shin, W. Han, S. Hong and H. Chafi, Taming Subgraph Isomorphism for RDF Query Processing, *PVLDB* **8**(11) (2015), 1238–1249.

[10] L. Zou, J. Mo, L. Chen, M.T. Özsu and D. Zhao, gStore: Answering SPARQL Queries via Subgraph Matching, *PVLDB* **4**(8) (2011), 482–493.

[11] V. Ingalalli, D. Ienco, P. Poncelet and S. Villata, Querying RDF Data Using A Multigraph-based Approach, in: *EDBT 2016, Bordeaux, France, March 15-16, 2016.*, 2016, pp. 245–256. doi:10.5441/002/edbt.2016.24.

[12] F. Goasdoué, I. Manolescu and A. Roatis, Efficient query answering against dynamic RDF databases, in: *EDBT Genoa, Italy, March 18-22*, 2013, pp. 299–310.

[13] S. Harris and N. Gibbins, 3store: Efficient Bulk RDF Storage, in: *PSSS1 , Sanibel Island, Florida, USA, October 20*, 2003.

[14] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev and R. Velkov, OWLIM: A family of scalable semantic repositories, *Semantic Web* **2**(1) (2011), 33–42.

[15] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini and R. Rosati, Tractable Reasoning and Efficient Query Answering in Description Logics: The *DL-Lite* Family, *J. Autom. Reasoning* **39**(3) (2007), 385–429.

[16] G. Gottlob, G. Orsi and A. Pieris, Ontological queries: Rewriting and optimization, in: *ICDE 2011, April 11-16, 2011, Hannover, Germany*, 2011, pp. 2–13.

[17] L. Zou and M.T. Özsu, Graph-Based RDF Data Management, *Data Science and Engineering* **2**(1) (2017), 56–70.

[18] B.H. Bloom, Space/Time Trade-offs in Hash Coding with Allowable Errors, *Commun. ACM* **13**(7) (1970), 422–426.

[19] W. Han, J. Lee and J. Lee, Turbo$_{\mathrm{iso}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases, in: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, 2013, pp. 337–348.

[20] F. Bi, L. Chang, X. Lin, L. Qin and W. Zhang, Efficient Subgraph Matching by Postponing Cartesian Products, in: *SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 2016, pp. 1199–1214.

[21] H. Shang, Y. Zhang, X. Lin and J.X. Yu, Taming verification hardness: an efficient algorithm for testing subgraph isomorphism, *PVLDB* **1**(1) (2008), 364–375.

[22] L. Zou, M.T. Özsu, L. Chen, X. Shen, R. Huang and D. Zhao, gStore: a graph-based SPARQL query engine, *VLDB J.* **23**(4) (2014), 565–590. doi:10.1007/s00778-013-0337-7.

[23] L. Zeng and L. Zou, Redesign of the gStore system, *Frontiers Comput. Sci.* **12**(4) (2018), 623–641. doi:10.1007/s11704-018-7212-z.

[24] D.J. Abadi, A. Marcus, S. Madden and K. Hollenbach, SW-Store: a vertically partitioned DBMS for Semantic Web data management, *VLDB J.* **18**(2) (2009), 385–406.

[25] T. Neumann and G. Weikum, x-RDF-3X: Fast Querying, High Update Rates, and Consistency for RDF Databases, *PVLDB* **3**(1) (2010), 256–263.

[26] C. Weiss, P. Karras and A. Bernstein, Hexastore: sextuple indexing for semantic web data management, *PVLDB* **1**(1) (2008), 1008–1019.

[27] J. Huang, D.J. Abadi and K. Ren, Scalable SPARQL Querying of Large RDF Graphs, *PVLDB* **4**(11) (2011), 1123–1134.

[28] O. Udrea, A. Pugliese and V.S. Subrahmanian, GRIN: A Graph Based RDF Index, in: *AAAI , July 22-26, 2007, Vancouver, British Columbia, Canada*, 2007, pp. 1465–1470.

[29] X. Lyu, X. Wang, Y. Li, Z. Feng and J. Wang, GraSS: An Efficient Method for RDF Subgraph Matching, in: *WISE 2015, Miami, USA, November 1-3*, 2015, pp. 108–122.

[30] J.J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne and K. Wilkinson, Jena: implementing the semantic web recommendations, in: *WWW 2004, New York, USA, May 17-20*, 2004, pp. 74–83.

[31] J. Broekstra, A. Kampman and F. van Harmelen, Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema, in: *ISWC 2002, Sardinia, Italy, June 9-12*, 2002, pp. 54–68.

[32] D. Bursztyn, F. Goasdoué and I. Manolescu, Optimizing Reformulation-based Query Answering in RDF, in: *EDBT 2015, Brussels, Belgium, March 23-27*, 2015, pp. 265–276.

[33] W. Zheng, L. Zou, W. Peng, X. Yan, S. Song and D. Zhao, Semantic SPARQL Similarity Search Over RDF Knowledge Graphs, *PVLDB* **9**(11) (2016), 840–851.