Ontology-Mediated Query Answering Using Graph Patterns with Conditions

Ping Lu, Ting Deng, Haoyuan Zhang, Yufeng Jin, Feiyi Liu, Tiancheng Mao, Lexiao Liu
School of Computer Science and Engineering, Beihang University
{luping@, dengting@act.}buaa.edu.cn, zhyuan97@163.com, {jinyf@act., liufeiyi@act., maotc@act., liulex@}buaa.edu.cn

***Abstract*—This paper proposes an extension of graph patterns, referred to as ontological graph patterns (OGPs), to facilitate ontology-mediated query answering. OGPs employ graph patterns to support topological feature queries, and attach conditions to vertices and edges, to specify ontological constraints. As a result, OGPs can not only express conjunctive queries under ontological constraints, but also provide a succinct graph representation for multiple queries. We develop a PTIME algorithm to generate an equivalent OGP from a conjunctive query under the ontology specified by description logic $DL\text{-}Lite_{\mathcal{R}}$, and provide a matching algorithm to match OGPs in graphs. Using real-life and synthetic data, we experimentally verify that the proposed approach is faster than the state-of-the-art ontology-mediated query answering algorithms by 2–3 orders of magnitude.**

I. Introduction

Ontology-mediated query answering has been extensively studied in the last decade [1], [2], [3], [4], [5], [6] and used in various domains, *e.g.,* chemistry [7], bioinformatics [8], energy sources [9] and e-learning recommendation systems [10], [11]. Given a query Q, a dataset D and an ontology O, ontology-mediated query answering identifies not only the answers of the query Q in D but also the answers *logically implied* by constraints in O. For instance, if a constraint states that every professor is a faculty, then all professors are also returned when Q searches all faculties in D, even if such information is not stored in D. Ontology-mediated query answering provides more complete query results over incomplete data [1], [12].

Query rewriting is a common technique to answer ontology-mediated queries (OMQs) [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]: it first rewrites the query into an equivalent conventional query (*e.g.,* a union of conjunctive queries or a Datalog program), and then directly evaluates the equivalent query in the given database. However, the generated queries may be exponentially large [4] and are costly to evaluate [12].

Graph databases [23] have been developed, and are often more efficient than relational databases [24]. It is desirable to exploit graph techniques to answer ontology-mediated queries. However, existing graph query languages, *e.g.,* SPARQL [25] and conditional graph patterns (CGPs) [26], cannot represent an ontology-mediated query in a single query, since they do not support the alternative selection of constants. Instead, they must use multiple queries, which may contain redundancy.

Example 1: Below are two OMQs adapted from an e-learning recommendation system [10], [11] and a university domain [27]. They are represented by graph patterns in Figure 1.

(1) Query Q_1 of Figure 1 identifies learning resources x that are categorized as hardware and are uploaded in 2023. And an ontology on computer science states that Processor, Memory and I/O Devices are hardware [11]. Then the resources uploaded in 2023 and categorized as Processor, Memory or I/O Devices can be returned. Hence, Q_1 can be rewritten into four queries. However, (a) neither SPARQL nor CGPs can represent these queries as a single graph pattern, since SPARQL and CGPs do not support the alternative semantics of x (*e.g.,* x carries different labels in Q_1 and Q_2). (b) Moreover, when we do not know the ontology in advance, we cannot manually write the four SPARQL queries or CGPs to represent Q_1.

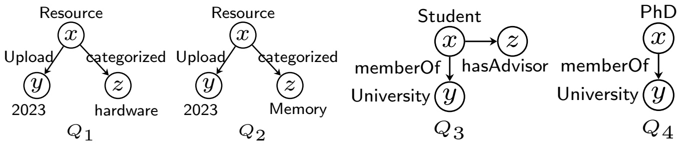


Fig. 1. Graph patterns

(2) Query Q_3 of Figure 1 finds student x who is in university y and has advisor z. The ontological constraints state that every PhD is a student and has an advisor. Thus, if x is a PhD, there is no need to check whether x has an advisor, *i.e.,* the results of query Q_4 of Figure 1 are also results of Q_3. However, queries Q_3 and Q_4 have different topological structures, *e.g.,* z exists in Q_3 but not in Q_4, and such two queries cannot be expressed by a single SPARQL query or CGP. □

The patterns for OMQs overlap and share computation, as shown in Example 1. Can we extend graph patterns to express OMQs and reduce the redundancy? Can we exploit SOTA pattern matching algorithms to accelerate the query answering?

Contributions. To answer these questions, we propose a new class of graph patterns, namely ontological graph patterns, to improve the expressive power of graph patterns.

(1) Ontological graph patterns (Section III). We propose ontological graph patterns (OGPs). In contrast to conventional patterns, SPARQL and CGPs, OGPs support disjunctions of conditions on vertices and edges. Moreover, it also supports the partial matching semantics (*i.e.,* some pattern vertices can have no matches), which is similar to the OPTIONAL feature of SPARQL. Due to these features, we can use OGPs to facilitate ontology-mediate query answering (see below).

(2) A rewriting algorithm (Section IV). We show that a conjunctive query (CQ) under ontological constraints specified by description logic $DL\text{-}Lite_{\mathcal{R}}$ [13] can be expressed as an OGP. Rewriting such queries into a union of CQs (UCQ) may lead to

exponentially large queries [13], [17], [14], [18], [4]. To avoid this, we exploit both the disjunctions of conditions and the partial matching semantics to encode the UCQ. More specifically, we design a PTIME algorithm GenOGP to generate an OGP of polynomial size, which is equivalent to the exponentially large UCQ generated in [13], [17], [14], [18], [4].

(3) A matching algorithm (Section V). Although OGPs are more expressive and complex than conventional patterns, SPARQL and CGPs, we can extend existing matching algorithms without substantial modification to match OGPs. As a case study, we extend the matching algorithm DAF [28] due to its effective pruning techniques [29], and design an algorithm OMatch to compute the matches $Q(G)$ from given OGP Q and graph G, We show that OMatch retains the same complexity and properties (*i.e.,* soundness and equivalence) of DAF.

(4) Experimental study (Section VI). Using real-life and synthetic data, we empirically verify the efficiency, effectiveness and scalability of our algorithms. On average, (a) GenOGP takes 0.1ms to generate an equivalent OGP from a CQ of size 16 and a $DL\text{-}Lite_{\mathcal{R}}$ ontology with 1.7K axioms; and it is 29.9 times faster than the best baseline. (b) Conditions in OGPs do not slow down the matching process, but often accelerate the computation by reducing the search space. Given an OGP Q generated by GenOGP, algorithm OMatch takes 1.3s to compute $Q(G)$ on a real-life graph G with 17.5M vertices and edges, and beats the state-of-the-art algorithms for ontology-mediated query answering by 2–3 orders of magnitude. (c) OMatch scales well with large graphs. On a synthetic graph with 57.3M vertices and edges, it takes 34.6s to compute the query results, while the fastest baseline takes 830s.

Related work. We categorize the related work as follows.

Ontology languages. They are designed to represent and reason about knowledge. (1) Description logics [30] are decidable fragments of first-order logic [4], and are the most studied ontology languages [1]. Different description logics vary in their expressivity and complexity. $DL\text{-}Lite_{\mathcal{R}}$ [13] is used to query large datasets and forms the basis of OWL 2 QL [31]; other description logics, *e.g.,* $\mathcal{EL}$ [32], Horn-$\mathcal{SHIQ}$ [33] and $\mathcal{SROIQ}$ [34], allow more constraints, but result in higher complexity [5]. (2) Rule-based ontology languages are also defined, *e.g.,* existential rules [35] and Datalog$^{\pm}$ [36].

This work focuses on query answering over $DL\text{-}Lite_{\mathcal{R}}$.

Ontology-mediated query answering. There are a host of work on ontology-mediated query answering, including (1) theoretical studies [37], [38], [39], [5], [40] that establish the complexity of query answering; and (2) designing algorithms to answer queries. For example, (a) [41], [42], [43], [44] first adopt a saturation technique to complete the datasets by deducing all implicit information encoded in the ontologies, and then evaluate the query on the completed datasets. (b) [13], [14], [15], [16], [17], [18], [19], [20], [21], [22] rewrite the given queries into equivalent queries that can be directly evaluated on datasets.

This work aims to answer CQs under the ontological constraints expressed by $DL\text{-}Lite_{\mathcal{R}}$ using query rewriting. The existing work is summarized as follows. (1) PerfectRef [13] rewrites a CQ into a union of CQs (UCQ), and various optimizations [17], [14], [18] have been developed to reduce the sizes of generated UCQs; but such UCQs are inevitable to be exponentially large [4]; (2) some algorithms rewrite a CQ into a Datalog program [20], [15], [19], [16]; and (3) CQs are also rewritten into more succinct representations, *e.g.,* unions of semi-CQs [21] and joins of UCQs [22].

This work differs from the prior work as follows. (1) Instead of rewriting a CQ into an exponentially large UCQ or Datalog program, we rewrite a CQ into an OGP, which has polynomial size. (2) We design an efficient ontology-mediated query answering algorithm by extending SOTA matching algorithms.

Matching algorithms. Pattern matching has been extensively studied. Most algorithms [45], [46], [47], [48], [49], [29], [28] adopt a *preprocessing-enumeration* framework [29]. For example, CFL-Match [46] and CECI [48] first build auxiliary structures based on a tree representation of patterns, then enumerate matches following a matching order of pattern vertices. As opposed to tree representations, DAF [28] and VEQ$_{\mathsf{M}}$ [49] adopt directed acyclic graph (DAG) representations, which have better pruning effect [29]. [50] exploits the multiple query optimization strategy to accelerate subgraph isomorphism search.

Pattern matching has also been studied for SPARQL queries [51], [52], [53]. Specifically, gStore [51] designs indices to accelerate the computation of SPARQL, while [52] and [53] extend existing subgraph isomorphism algorithms [47] and [46] to handle SPARQL, respectively. [54] proposes heuristic techniques to exploit the multi-query optimization for SPARQL.

Closer to our work are CGPs [26], which extend patterns by attaching conditions to edges indicating when the edges are matched, and use annotations to encode multiple patterns.

This work differs from the prior work in the following way. (1) We propose OGPs, which support more complex conditions than CGPs, *e.g.,* disjunction of conditions defined on both vertices and edges. (2) We extend the pattern matching algorithm DAF to match OGPs, without increasing its complexity or losing its properties, although OGPs are more complex than conventional patterns, SPARQL and CGPs. (3) As an application of OGPs, we rewrite a CQ under description logic $DL\text{-}Lite_{\mathcal{R}}$ [13] into an OGP, which is not studied in [26].

II. Query answering over description logic

We first introduce the description logic $DL\text{-}Lite_{\mathcal{R}}$, and then define the notation for query answering over $DL\text{-}Lite_{\mathcal{R}}$.

Description logic $DL\text{-}Lite_{\mathcal{R}}$. It is a lightweight description logic targets at efficient query answering, and underpins the W3C OWL 2 QL [31]. The basis of $DL\text{-}Lite_{\mathcal{R}}$ includes concepts, roles, and inclusion and membership assertions.

Concepts and roles. Concepts C denote sets of constants, and roles R denote binary relations between concepts; *i.e.,*

$$C ::= A \mid \exists R, \qquad\qquad R ::= P \mid P^{-},$$

where (1) A is an *atomic concept*; (2) P is an *atomic role*; (3) $\exists R$ is an unqualified existential restriction [30] on role R; and (4) P^{-} is an inverse of role P.

Inclusion assertions. Inclusion assertions, also known as inclusions or axioms, consist of: (1) *concept inclusions* of the form $C_1 \sqsubseteq C_2$, where C_1 and C_2 are two concepts; and (2) *role inclusions* of the form $R_1 \sqsubseteq R_2$, where R_1 and R_2 are two roles. Intuitively, $C_1 \sqsubseteq C_2$ (resp. $R_1 \sqsubseteq R_2$) states that all instances of concept C_1 (resp. role R_1) are also instances of concept C_2 (resp. role R_2). A set of inclusion assertions is called TBox (*i.e.,* ontology), denoted by $\mathcal{T}$.

Remark. We do not consider the negations of concepts and roles, as in [55], since they cannot provide more query results and are not involved in the query answering. Indeed, the negations of concepts and roles can only appear in the negative inclusions of the form $C_1 \sqsubseteq \neg C_2$ (resp. $R_1 \sqsubseteq \neg R_2$) [56], which only restrict the existence of concept C_1 (resp. role R_1) when concept C_2 (resp. role R_2) exists. Moreover, such negative inclusions are rare in real-life ontologies; *e.g.,* there exist only 10 negative inclusions among 1703 inclusions in DBpedia [41].

Membership assertions. Membership assertions on atomic concepts and atomic roles are (1) *concept assertions* of the form $A(c)$ and (2) *role assertions* of the form $P(c_1, c_2)$, respectively. Intuitively, $A(c)$ states that constant c is an instance of concept A, and $P(c_1, c_2)$ states that the pair (c_1, c_2) forms an instance of role P. A set of membership assertions is called ABox (*i.e.,* dataset), denoted by $\mathcal{A}$.

Knowledge bases. A knowledge base (KB) $\mathcal{K}{=}\langle\mathcal{T}, \mathcal{A}\rangle$ consists of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$. A *model* of $\mathcal{K}$ is a set of constants I such that $I \models \mathcal{K}$, *i.e., I* satisfies all constraints specified by ABox $\mathcal{A}$ and TBox $\mathcal{T}$.

Here, we consider TBox $\mathcal{T}$ expressed by $DL\text{-}Lite_{\mathcal{R}}$.

Example 2: Consider a KB $\mathcal{K} = \langle\mathcal{T}, \mathcal{A}\rangle$, where $\mathcal{T} = \{T_1$: Student $\sqsubseteq \exists$takesCourse, T_2: PhD $\sqsubseteq$ Student, T_3: PhD $\sqsubseteq \exists$advisorOf$^-\}$, and $\mathcal{A} = \{$PhD(Ann)$\}$. TBox $\mathcal{T}$ states that each student must take some course (T_1), each PhD is a student (T_2) and has an advisor (T_3). ABox $\mathcal{A}$ says that Ann is a PhD. □

Query answering. A conjunctive query (CQ) q over a KB $\mathcal{K}$ is a first-order logic query of the form $q(\bar{x}) = \exists\bar{y}.\varphi(\bar{x}, \bar{y})$, where $\bar{x}$ and $\bar{y}$ are tuples of *distinguished* variables and *existential* variables, respectively; and $\varphi(\bar{x}, \bar{y})$ is a conjunction of *atoms* of the form $A(x)$ or $P(x, y)$, where A and P are an atomic concept and an atomic role in $\mathcal{K}$, respectively. The existential variables that occur in q only once are called *unbound* and can be represented by the symbol '_' [13]. Otherwise, an existential variable is *bound* if it appears more than once in q.

The result $q(\mathcal{K})$ of q over $\mathcal{K}$ is a set of tuples of constants $\bar{c}$, such that $q(\bar{c})$ is satisfied in *every* model of $\mathcal{K}$ [13].

Example 3: Consider the following CQ q:

$q(x) = \exists y_1, y_2, y_3, z.\mathsf{advisorOf}(y_1, x) \wedge \mathsf{advisorOf}(y_1, y_2) \wedge \mathsf{advisorOf}(y_1, y_3) \wedge \mathsf{takesCourse}(x, z)$.

Given the KB $\mathcal{K}$ in Example 2, Ann is an answer to q, although $\mathcal{A}$ contains only one assertion PhD(Ann). This is because (1) inclusion T_3 in $\mathcal{T}$ states that Ann must have some advisor y, and (2) T_1 and T_2 in $\mathcal{T}$ ensure that Ann takes some course z. If $\mathcal{T}$ is absent, then q has no answer in $\mathcal{A}$. □

TABLE I
NOTATIONS

Notations	Definitions
$G = (V, E, L, F_A)$	a directed labeled graph
A/P	an atomic concept/atomic role
TBox/ABox	a set of inclusion/member assertions
$\mathcal{K} = \langle\mathcal{T}, \mathcal{A}\rangle$	a KB with a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$
$Q[\bar{x}]{=}(V_Q, E_Q, L_Q, \mathcal{C}^l, \mathcal{C}^o)$	an ontological graph pattern
$DL\text{-}Lite_{\mathcal{R}}$	a description logic
CS/OMCS	index structures to compute matches

Query rewriting. For any CQ q and $DL\text{-}Lite_{\mathcal{R}}$ TBox $\mathcal{T}$, there exists an *equivalent* query q_o, denoted by $q_o \equiv_{\mathcal{T}} q$, such that the answers to q over a KB $\mathcal{K}{=}\langle\mathcal{T}, \mathcal{A}\rangle$ are precisely the answers to q_o over $\mathcal{A}$ [13]. Therefore, query answering over $DL\text{-}Lite_{\mathcal{R}}$ can be conducted in two steps: (1) generate an equivalent query q_o from q over $\mathcal{T}$, and (2) evaluate q_o in $\mathcal{A}$.

Then we answer a CQ q over a KB $\mathcal{K}{=}\langle\mathcal{T}, \mathcal{A}\rangle$ by (1) first generating an OGP Q such that $Q \equiv_{\mathcal{T}} q$ (Section IV-B), and (2) then computing the matches of Q in $\mathcal{A}$ (Section V-B).

The notations used in the paper are listed in Table I.

III. Ontological Graph Patterns

In this section we formally define ontological graph patterns.

Definitions. Assume three infinite sets Θ, Υ and U of symbols for labels, attributes and constants, respectively.

Graphs. A directed labeled graph is specified as $G = (V, E, L, F_A)$, where (1) V is a finite set of vertices, where each vertex v carries a label l from Θ, denoted by $L(v) = l$; (2) $E \subseteq V \times \Theta \times V$ is a finite set of edges, where $e{=}(v, l, v')$ is an edge from v to v' labeled l, denoted by $L(e){=}l$; and (3) each vertex $v{\in}V$ carries a tuple $F_A(v){=}(A_1 = a_1, \ldots, A_n = a_n)$ of attributes of a finite arity, written as $v.A_i = a_i$, where $A_i \in \Upsilon$, $a_i \in U$, and $A_i{\neq}A_j$ if $i{\neq}j$, representing properties.

To simplify the presentation, assume that each vertex carries at most one label. The proposed algorithms can be readily extended to handle graphs where vertices carry multiple labels.

Patterns. An *ontological graph pattern*, denoted by OGP, is defined as $Q[\bar{x}]{=}(V_Q, E_Q, L_Q, \mathcal{C}^l, \mathcal{C}^o)$, where (1) V_Q (resp. E_Q) is a finite set of pattern vertices (resp. edges); (2) L_Q assigns a label in Θ to each vertex $u{\in}V_Q$, denoted by $L_Q(u)$; we allow wildcard '$*$' as a special label in Θ, *i.e.,* the wildcard '$*$' can match any symbol in Θ; and (3) $\mathcal{C}^l(u)$, $\mathcal{C}^l(e)$ and $\mathcal{C}^o(u)$ are *conditions* τ that are recursively defined over vertices u and edges e as follows:

$$\tau ::= x.A \oplus c \mid x.A \oplus y.B \mid l(x) \mid l(x, y) \mid \tau \wedge \tau \mid \tau \vee \tau,$$

where (a) x and y are vertices in V_Q (*i.e.,* variables in $\bar{x}$); (b) $x.A$ and $y.B$ denote attributes A and B of vertices x and y, respectively; (c) c is a constant in U; (d) $\oplus$ is one of the following comparison operators: $=, \neq, <, \leq, >, \geq$; (e) $l(x)$ denotes that vertex x carries label l; (f) $l(x, y)$ denotes an edge labeled l from vertex x to vertex y; and (g) the conditions can be combined with conjunction $\wedge$ and disjunction $\vee$.

Intuitively, an OGP Q extends a conventional pattern by attaching (1) *matching* condition $\mathcal{C}^l(x)$ (resp. $\mathcal{C}^l(e)$) to vertex x (resp. edge e), specifying when the vertex x (resp. edge e) can be matched; and (2) *omission* condition $\mathcal{C}^o(x)$ to vertex x, stating when x can be omitted from the query results (see below).

We consider *w.l.o.g.* only connected OGPs in the sequel, as in [28], [26]. This said, all the techniques proposed here can be readily extended to disconnected patterns.

Queries to graphs. CQs over KBs can be expressed by graph patterns [5], [57]. For any CQ $q(\bar{x}) = \exists \bar{y}.\varphi(\bar{x}, \bar{y})$, we define an OGP $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mathcal{C}^l, \mathcal{C}^o)$ from q as follows. (1) V_Q contains vertices that represent variables in $\bar{x} \cup \bar{y}$. (2) For each atom g in q, if g is $A(x)$, then L_Q assigns a label A to x, and if g is $P(x, y)$, then E_Q contains an edge e from x to y with label P. To simplify the presentation, we assume *w.l.o.g.* that for each $x \in \bar{x}$ (resp. pair $x, y \in \bar{x}$) there exists *at most one* atom $A(x)$ (resp. $P(x, y)$) in q. Accordingly, (3) we define the matching conditions $C^l(x) = A(x)$ and $C^l(e) = P(x, y)$, and for each $u \in V_Q$, the omission condition is $C^o(u) = \emptyset$.

Intuitively, *atom* $A(x)$ (resp. $P(x, y)$) in CQs is represented by a *condition* $A(x)$ (resp. $P(x, y)$) in OGPs. Note that the proposed algorithms can be readily extended to handle OGP encoding CQs with multiple atoms on the same variables

Example 4: We next show that OGPs can (a) express CQs under ontological constraints and (b) encode multiple patterns.

(1) OGP $Q'_1[x, y, z] = (V_{Q'_1}, E_{Q'_1}, L_{Q'_1}, \mathcal{C}^l_1, \mathcal{C}^o_1)$ extends pattern Q_1 of Figure 1 as follows: (a) assign label wildcard '$*$' to all vertices x, y and z; and (b) attach matching condition $\mathcal{C}^l_1(z) = \mathsf{hardward}(z) \vee \mathsf{Processor}(z) \vee \mathsf{Memory}(z) \vee$ I/O Devices(z) to z, *i.e.,* z carries label hardward, Processor, Memory or I/O Devices. Thus OGP Q'_1 contains the two patterns Q_1 and Q_2, and enforces the ontological constraint, *i.e.,*Processor, Memory and I/O Devices are all hardware [11].

(2) OGP $Q'_3[x, y, z] {=} (V_{Q'_3}, E_{Q'_3}, L_{Q'_3}, \mathcal{C}^l_3, \mathcal{C}^o_3)$ extends pattern Q_3 of Fig. 1 with (a) labeling wildcard '$*$' on x; (b) matching condition $\mathcal{C}^l_3(x){=}\mathsf{Student}(x) \vee \mathsf{PhD}(x)$, *i.e.,* the label of x is Student or PhD; and (b) omission condition $\mathcal{C}^o_3(z){=}\mathsf{PhD}(x)$, *i.e.,* z can be omitted in Q'_3 when x is labeled PhD. Then OGP Q'_3 expresses both Q_3 and Q_4, and enforces the ontological constraints, *i.e.,* every PhD is a student and has an advisor.

(3) OGPs can also encode multiple patterns. Consider patterns Q_5 and Q_6 in Fig. 2. (a) Q_5 finds all professors x_1 who work for university x_4, and teach student x_2 who publishes article x_3; and (b) Q_6 finds all teachers x_1 who teach a student x_2 taking course x_3. Due to their similar topological structures, Q_5 and Q_6 can be encoded as an OGP $Q'_5[x_1, x_2, x_3, x_4] = (V_{Q'_5}, E_{Q'_5}, L_{Q'_5}, \mathcal{C}^l_5, \mathcal{C}^o_5)$, where (a) $V_{Q'_5}$, $E_{Q'_5}$ are the same as Q_5 in Fig. 2; (b) vertices x_1, x_3 and the edge from x_2 to x_3 are labeled wildcard '$*$', and the other labels are the same as in Q_5; and (c) conditions $\mathcal{C}^l_5$ and $\mathcal{C}^o_5$ are defined as follows:

○ $\mathcal{C}^l_5(x_1){=}\mathsf{Professor}(x_1) \vee \mathsf{Teacher}(x_1)$, *i.e.,* the label of vertex x_1 is either Professor as in Q_5, or Teacher as in Q_6;

○ $\mathcal{C}^l_5(x_2, *, x_3){=}(\mathsf{publishes}(x_2, x_3) \wedge \mathsf{Professor}(x_1)) \vee (\mathsf{takes}(x_2, x_3) \wedge \mathsf{Teacher}(x_1))$, *i.e.,* when the edge $e = (x_2, *, x_3)$ carries label publishes, x_1 carries label Professor, as in Q_5; and when edge e is labeled takes, x_1 is labeled Teacher, as in Q_6.

○ $\mathcal{C}^l_5(x_3) = (\mathsf{Article}(x_3) \wedge \mathsf{Professor}(x_1)) \vee (\mathsf{Course}(x_3) \wedge \mathsf{Teacher}(x_1))$, *i.e.,* when x_3 carries label Article, x_1 carries label Professor, as in Q_5; and when x_3 is labeled Course, x_1 is labeled Teacher, as in Q_6.

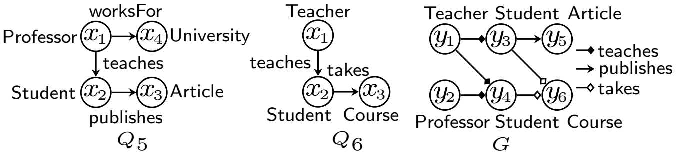


Fig. 2. Graph patterns and graphs

○ $\mathcal{C}^o_5(x_4) = \mathsf{Teacher}(x_1)$, *i.e.,* vertex x_4 can be omitted in Q'_5 when vertex x_1 carries label Teacher, as in Q_6. □

Semantics. To interpret OGPs, we define *ontological pattern matching*. Assume a graph $G = (V, E, L, F_A)$, an OGP $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mathcal{C}^l, \mathcal{C}^o)$ and a subset V_h of V_Q.

A *partial mapping* h from $\bar{x}$ (*i.e.,* V_Q) to V *w.r.t.* V_h is a *homomorphism* from $\bar{x}$ to V such that (a) $h(x) \in V$ when $x \in V_h$, and (b) $h(x) = \perp$ when $x \notin V_h$ (*i.e.,* $x \in V_Q \setminus V_h$), where $\perp$ is a *dummy* vertex that is not in V; that is, vertices in $V_Q \setminus V_h$ do not have matches in the graph G.

We say that a partial mapping h *satisfies* a condition τ denoted by $h \models \tau$, if the following holds: (a) if τ is $x.A \oplus c$, then $h(x) \neq \perp$, $h(x)$ carries attribute A, and $h(x).A \oplus c$; (b) if τ is $x.A \oplus y.B$, then $h(x) \neq \perp$, $h(y) \neq \perp$, $h(x)$ (resp. $h(y)$) carries attribute A (resp. B), and $h(x).A \oplus h(y).B$; (c) if τ is $l(x)$, then $h(x) \neq \perp$, and $h(x)$ carries label l; (d) if τ is $l(x, y)$, then $h(x) \neq \perp$, $h(y) \neq \perp$, and there exists an edge in G from $h(x)$ to $h(y)$ labeled l; (e) if τ is $\tau_1 \wedge \tau_2$, then both $h \models \tau_1$ and $h \models \tau_2$; and (f) if τ is $\tau_1 \vee \tau_2$, then either $h \models \tau_1$ or $h \models \tau_2$.

A *match* of an OGP $Q[\bar{x}]$ in a graph $G{=}(V, E, L, F_A)$ is a *partial mapping* h from $\bar{x}$ to V that satisfies the following:

○ for each vertex $x \in V_Q$, if $h(x) {\neq} \perp$, *i.e.,* x has a match, then $L_Q(x) \asymp L(h(x))$ and $h \models \mathcal{C}^l(x)$; otherwise, $\mathcal{C}^o(x) \neq \emptyset$ and $h \models \mathcal{C}^o(x)$; and

○ for each edge $e{=}(x, l, y) {\in} E_Q$, there is an edge $e' = (h(x), l', h(y))$ in G such that $l \asymp l'$ and $h \models \mathcal{C}^l(e)$; otherwise, (a) $\mathcal{C}^o(x) \neq \emptyset$ and $h \models \mathcal{C}^o(x)$, or (b) $\mathcal{C}^o(y) \neq \emptyset$ and $h {\models} \mathcal{C}^o(y)$; *i.e.,* x or y is omitted in h.

Here $l {\asymp} l'$ if $l{=}l'$ or l is '$*$', *i.e.,* wildcard matches any label.

Denote by $h(\bar{x})$ the tuple of $h(x)$ for all $x {\in} \bar{x}$ with $h(x) {\neq} \perp$, and $Q(G)$ the set of tuples $h(\bar{x})$ for all matches h of Q in G.

Example 5: For the OGP Q'_5 described in Example 4 and the graph G illustrated in Figure 2, a match of Q'_5 in G is h_1: $x_1 {\mapsto} y_1$, $x_2 {\mapsto} y_3$, $x_3 {\mapsto} y_6$, $x_4 {\mapsto} \perp$. Since vertex x_1 in Q'_5 is mapped to vertex y_1 labeled Teacher in G, $h_1 \models \mathcal{C}^o(x_4)$ and x_4 can be omitted in h_1, *i.e.,* $h_1(x_4) = \perp$. Similarly, another match of Q'_5 in G is h_2: $x_1 {\mapsto} y_1$, $x_2 {\mapsto} y_4$, $x_3 {\mapsto} y_6$, $x_4 {\mapsto} \perp$, and the set of matches $Q'_5(G) = \{h_1(\bar{x}), h_2(\bar{x})\}$. □

IV. Generating Ontological Patterns

We propose a PTIME algorithm GenOGP to generate an OGP Q that is equivalent to a given CQ q under a set $\mathcal{T}$ of ontological constraints specified by description logic $DL\text{-}Lite_{\mathcal{R}}$. We design GenOGP based on the classic rewriting algorithm PerfectRef [13]. In the following, we first review PerfectRef in Section IV-A, and then present GenOGP in Section IV-B.

TABLE II
DEDUCTIONS OVER $DL\text{-}Lite_{\mathcal{R}}$

Inclusions I	**Atoms g**	$\boldsymbol{gr(g,I)}$	**Condition deductions**
I_1: $A_2 \sqsubseteq A_1$	$A_1(x)$	$A_2(x)$	r_1: $g \in \mathcal{C}^l(x) \rightarrow \mathcal{C}^l(x) \cup \{gr(g,I)\}$ r_2: $\forall z(g{\in}\mathcal{X}(z){\rightarrow}\mathcal{X}(z){\cup}\{gr(g,I)\})$
I_2: $P_2 \sqsubseteq P_1$ I_3: $P_2^- \sqsubseteq P_1$	$P_1(x,y)$ $P_1(x,y)$	$P_2(x,y)$ $P_2(y,x)$	r_3: $g{\in}\mathcal{C}^l(x,y){\rightarrow}\mathcal{C}^l(x,y){\cup}\{gr(g,I)\}$ r_4: $\forall z(g{\in}\mathcal{X}(z){\rightarrow}\mathcal{X}(z){\cup}\{gr(g,I)\})$
I_4: $\exists P_2 \sqsubseteq \exists P_1$	$P_1(x,\_)$	$P_2(x,\_)$	r_5: $g{\in}\mathcal{C}^l(x){\rightarrow}\mathcal{C}^l(x){\cup}\{gr(g,I)\}$ r_6: $\forall z(g{\in}\mathcal{X}(z){\rightarrow}\mathcal{X}(z){\cup}\{gr(g,I)\})$
I_5: $\exists P_2^- \sqsubseteq \exists P_1$	$P_1(x,\_)$	$P_2(\_,x)$	
I_6: $\exists P_2 \sqsubseteq \exists P_1^-$	$P_1(\_,x)$	$P_2(x,\_)$	
I_7: $\exists P_2^- \sqsubseteq \exists P_1^-$	$P_1(\_,x)$	$P_2(\_,x)$	
I_8: $\exists P \sqsubseteq A$ I_9: $\exists P^- \sqsubseteq A$	$A(x)$ $A(x)$	$P(x,\_)$ $P(\_,x)$	r_7: $g \in \mathcal{C}^l(x) \rightarrow \mathcal{C}^l(x) \cup \{gr(g,I)\}$ r_8: $\forall z(g{\in}\mathcal{X}(z){\rightarrow}\mathcal{X}(z){\cup}\{gr(g,I)\})$ r_9: $P(x,z)$ is unique$\rightarrow\mathcal{U}(x){\cup}\{P(x,z)\}$ (for I_8) r_{10}: $P(z,x)$ is unique$\rightarrow\mathcal{U}(x){\cup}\{P(z,x)\}$ (for I_9)
I_{10}: $A \sqsubseteq \exists P$ I_{11}: $A \sqsubseteq \exists P^-$	$P(x,\_)$ $P(\_,x)$	$A(x)$ $A(x)$	r_{11}: $g \in \mathcal{C}^l(x) \rightarrow \mathcal{C}^l(x) \cup \{gr(g,I)\})$ r_{12}: $\forall y(g{\in}(\mathcal{C}^l(x,y) \cap \mathcal{U}(y))$ $\rightarrow(\mathcal{C}^l(x,y){\cup}\{gr(g,I)\}) \parallel (\mathcal{C}^o(y){\cup}\{gr(g,I)\})$

A. *Overview of the rewriting algorithm* PerfectRef

Given a CQ q and a set $\mathcal{T}$ of ontological constraints in $DL\text{-}Lite_{\mathcal{R}}$, algorithm PerfectRef generates an equivalent UCQ q_o such that $q_o \equiv_{\mathcal{T}} q$. More specifically, it works in the following three steps: (1) it first initializes a set $\mathcal{S}(q,\mathcal{T}) = \{q\}$; (2) then it iteratively extends $\mathcal{S}(q,\mathcal{T})$ with the newly generated CQs by *interleaving* two key procedures (*i.e.,* Deduction and Reduction; see below), until no more CQs can be generated; (3) finally it returns the union of all CQs in $\mathcal{S}(q,\mathcal{T})$.

Deduction. It generates new CQs from each CQ q_i in $\mathcal{S}(q,\mathcal{T})$ by applying inclusions in $\mathcal{T}$ on atoms in q_i. More specifically, for each atom g in q_i it generates a new CQ q_i' by replacing g with an atom $gr(g,I)$ that is generated by applying an inclusion I in $\mathcal{T}$ on g (see below for the definition of $gr(g,I)$).

Recall that there exist 11 types of inclusions in $DL\text{-}Lite_{\mathcal{R}}$ [13], denoted by I_1–I_{11} (see Table II). Based on these inclusions, the new atom $gr(g,I)$ is defined as follows: (1) if the inclusion I is $A_2 \sqsubseteq A_1$ (*i.e.,* I_1) and the atom g is $A_1(x)$, then $gr(g,I_1)$ is $A_2(x)$; and (2) if I is $A \sqsubseteq \exists P$ (*i.e.,* I_{10}) and g is $P(x,\_)$, then $gr(g,I_{10})$ is $A(x)$; recall that '_' represents an unbound variable that appears only once in q_i (see Section II). Other types of inclusions in $\mathcal{T}$ can be handled similarly.

Reduction. It removes redundant atoms from the generated CQs q_i in $\mathcal{S}(q,\mathcal{T})$. More specifically, if q_i contains two atoms g_1 and g_2 that can be unified by their *most general unifier* $\mathcal{G}(g_1,g_2)$, then it generates a new CQ q_i' by replacing g_1 and g_2 in q_i with $\mathcal{G}(g_1,g_2)$. Here, (1) the *most general unifier* $\mathcal{G}(g_1,g_2)$ is generated from g_1 and g_2 by replacing the unbound variable "_" in one atom with the variable in the other atom that appears in the same position as "_"; *e.g.,* if g_1 and g_2 are $P(x,\_)$ and $P(x,y)$, respectively, then their most general unifier is $P(x,y)$, *i.e.,* the symbol '_' is replaced by variable y that appears in the same position. (2) Atoms g_1 and g_2 can be unified by $\mathcal{G}(g_1,g_2)$, if g_1 and g_2 are the same role, and g_1 and g_2 have the same variable in the same position.

After the reduction, a bound variable in q_i may become unbound in q_i' (*i.e.,* appearing only once in q_i'), and some inclusions in $\mathcal{T}$ may now be further applied to $\mathcal{G}(g_1,g_2)$ in q_i'.

Example 6: Given the CQ q in Example 3 and the TBox $\mathcal{T}$ in Example 2, PerfectRef works as follows.

(1) At first, $\mathcal{S}(q,\mathcal{T})$ contains the CQ q, *i.e.,* $\mathcal{S}(q,\mathcal{T}){=}\{q\}$;

(2) It then applies T_1 to $\mathsf{takesCourse}(x,z)$ of q, and expands $\mathcal{S}(q,\mathcal{T})$ with the CQ $q_1(x) = \exists y_1,y_2,y_3.\mathsf{advisorOf}(y_1,x) \wedge \mathsf{advisorOf}(y_1,y_2) \wedge \mathsf{advisorOf}(y_1,y_3) \wedge \mathsf{Student}(x)$.

(3) It next reduces atoms $\mathsf{advisorOf}(y_1,x)$, $\mathsf{advisorOf}(y_1,y_2)$ and $\mathsf{advisorOf}(y_1,y_3)$ in q_1, and adds two new queries $q_2(x) = \exists y_1,y_2.\mathsf{advisorOf}(y_1,x) \wedge \mathsf{advisorOf}(y_1,y_2){\wedge}\mathsf{Student}(x)$ and $q_3(x){=}\exists y_1,y_3.\mathsf{advisorOf}(y_1,x){\wedge}\mathsf{advisorOf}(y_1,y_3){\wedge}\mathsf{Student}(x)$. This is because (a) y_2 and y_3 appear only once in q_1, and hence are unbound, (b) the most general unifier of $\mathsf{advisorOf}(y_1,x)$ and $\mathsf{advisorOf}(y_1,y_2)$ is $\mathsf{advisorOf}(y_1,x)$, and (c) the most general unifier of $\mathsf{advisorOf}(y_1,y_2)$ and $\mathsf{advisorOf}(y_1,y_3)$ is $\mathsf{advisorOf}(y_1,y_2)$. Similarly, it also reduces these atoms in q, and adds two more CQs q_4 and q_5 to $\mathcal{S}(q,\mathcal{T})$.

(4) Then it applies the inclusion T_2 to the atom $\mathsf{Student}(y)$ in q_1, q_2 and q_3 (see I_1 in Table II), and extends $\mathcal{S}(q,\mathcal{T})$ with three CQs q_6, q_7 and q_8, which is obtained from q_1, q_2 and q_3 by replacing $\mathsf{Student}(x)$ with $\mathsf{PhD}(x)$, respectively.

(5) It reduces atoms $\mathsf{advisorOf}(y_1,x)$ and $\mathsf{advisorOf}(y_1,y_2)$, $\mathsf{advisorOf}(y_1,y_3)$ in q_2–q_8, and adds the following three queries $q_9(x) = \exists y_1.\mathsf{advisorOf}(y_1,x) \wedge \mathsf{takesCourse}(x,z)$, $q_{10}(x) = \exists y_1,z.\mathsf{advisorOf}(y_1,x) \wedge \mathsf{Student}(x)$ and $q_{11}(x) = \exists y_1.\mathsf{advisorOf}(y_1,x) \wedge \mathsf{PhD}(x)$.

(6) After that, it applies T_3 to atom $\mathsf{advisorOf}(y_1,x)$ of q_{10} and q_{11} (see I_{10} in Table II), and adds two CQs $q_{12}(x) = \mathsf{Student}(x)$ and $q_{13}(x) = \mathsf{PhD}(x)$ to $\mathcal{S}(q,\mathcal{T})$.

(7) Since no more CQs can be generated, it returns the union of all CQs in $\mathcal{S}(q,\mathcal{T})$ (*i.e.,* q and q_1–q_{13}) as the rewriting q_o.

Then Ann is an answer to q_o in $\mathcal{A}{=}\{\mathsf{PhD(Ann)}\}$. □

PerfectRef may return a UCQ of exponential size, as deduction and reduction may be invoked exponentially many times.

Example 7: Consider $q_e(y_1){=}\exists x,y_2,\ldots,y_n.\bigwedge_{i\in[1,n]} P_i(x,y_i)$ and a TBox $\mathcal{T}$ with inclusions $\exists P_1{\sqsubseteq}\exists P_i$ ($i{\in}[2,n]$). Intuitively, q_e forms a star rooted at vertex x. According to the inclusions $\exists P_1 \sqsubseteq \exists P_i$ ($i \in [2,n]$), each atom $P_i(x,y_i)$ in $q_e(y_1)$ can be replaced by $P_1(x,y_i)$ (see I_4 in Table II). Then, during the deduction step, PerfectRef generates exponentially many CQs, one for each subset of $P_2(x,y_2), P_3(x,y_3),\ldots,P_n(x,y_n)$. Indeed, each subset $P_{i_1}(x,y_{i_1}), P_{i_2}(x,y_{i_2}),\ldots,P_{i_k}(x,y_{i_k})$ can be replaced by $P_1(x,y_{i_1}), P_1(x,y_{i_2}),\ldots,P_1(x,y_{i_k})$ due to the inclusions $\exists P_1{\sqsubseteq}\exists P_i$ ($i \in [2,n]$) in $\mathcal{T}$, and results in a new CQ.

Moreover, the reduction step can also result in exponentially many CQs, since (1) variables $y_1,\ldots,y_n$ appear only once in q_e and are unbound, and (2) any subset of atoms $P_1(x,y_2)$, $\ldots, P_1(x,y_n)$ in each CQ can be removed by reduction. □

Challenges. To avoid the exponentially large UCQ, we represent all CQs in $\mathcal{S}(q,\mathcal{T})$ by a *single* OGP, and show that such OGP can be constructed in PTIME. Similar to PerfectRef, the OGP can be constructed by iteratively applying inclusions in $\mathcal{T}$ to atoms in q, and deducing conditions in the sets $\mathcal{C}^l(\cdot)$ and $\mathcal{C}^o(\cdot)$ in OGPs. However, we need to address the following.

(1) Atoms (*i.e.,* vertices and edges) may be added to (*i.e.,* when applying rules I_8–I_9) or removed from (*i.e.,* when applying rules I_{10}–I_{11} or the reduction step) a CQ q during Deduction and Reduction. How can we encode these in a single OGP?

(2) If the updated vertices and edges are represented by conditions, how can we conduct Deduction and Reduction?

(3) Deduction and Reduction may run exponentially many times. Can we generate an equivalent OGP Q in PTIME? And whether the generated OGP Q has polynomial size?

Strategies. We tackle these challenges as follows.

(1) An auxiliary structure. We encode the added and removed atoms by matching and omission conditions in OGPs.

(a) For added atoms via inclusions I_8-I_9, the inclusions ensure the existence of vertex labels, and we can expand the matching conditions $\mathcal{C}^l(x)$ and $\mathcal{C}^l(e)$ to record such information.

(b) For removed atoms via inclusions I_{10}-I_{11}, such rules ensure the existences of some edges, and we can exploit the *omission* conditions to record such information. For examples, if rule $A \sqsubseteq \exists P$ (*i.e.,* I_{11}) is applied to edge $e(x,y)$, then we can add $A(x)$ to the omission condition $\mathcal{C}^o(y)$ of vertex y, indicating that if x carries label A, then vertex y can be omitted.

(c) The reduction step in PerfectRef may result in exponentially many CQs (see Example 7). To avoid this, we adopt a *lazy reduction* strategy (see below). To facilitate the lazy reduction we define a set $\mathcal{U}(x)$ of conditions for each edge (x,y) in Q, which record when x is unbound. We initialize the sets $\mathcal{U}(x)$ based on the input CQ q. More specifically, $\mathcal{U}(x)$ contains items likes $l(x,\_)$ or $l(\_,x)$ in q.

(2) Condition deduction. Using condition sets, we can conduct deduction as PerfectRef, via deducing new conditions. We define rules r *w.r.t.* inclusions in $DL\text{-}Lite_{\mathcal{R}}$ (Table II) as follows:

$$r\text{: } g \in \mathcal{C} \to \mathcal{C}' \cup \{gr(g,I)\},$$

where (a) g and $gr(g,I)$ are conditions (*i.e.,* atoms) as shown in Table II, and (b) $\mathcal{C}$ and $\mathcal{C}'$ are condition sets in OGPs or $\mathcal{U}$. Here, we use a set to denote the disjunction of conditions.

The rule r states that if g is in a condition set $\mathcal{C}$, then $gr(g,I)$ is deduced and added to the condition set $\mathcal{C}'$.

Based on the left-hand side (LHS) and right-hand side (RHS) of inclusions, rules are classified as follows (Table II):

(a) Rules r_1–r_6 handle inclusions I_1–I_7 whose LHS and RHS are in the same form. Consider inclusion I_1 (*i.e.,* $A_2 \sqsubseteq A_1$), which enforces that if x is an instance of concept A_2, then x is also an instance of concept A_1. So, A_2 is a candidate label for x, *i.e.,* rule r_1. Moreover, if $A_1(x)$ exists in other condition sets, then $A_2(x)$ is also added to the sets, *i.e.,* rule r_2. Here, $\mathcal{X}(\cdot)$ (see Table II) is one of the following sets: $\mathcal{C}^l(x,y), \mathcal{C}^o(y)$ and $\mathcal{U}(y)$. Other inclusions are handled similarly.

(b) Rules r_7–r_{12} handle inclusions I_8–I_{11} whose LHS and RHS are in different forms. For example, inclusion I_8 (*i.e.,* $\exists P \sqsubseteq A$) enforces that each vertex with an outgoing edge $P(x,\_)$ is an instance of concept A. Then when applying I_8 we (i) first add $P(x,\_)$ to the condition sets containing $A(x)$ (*i.e.,* rules r_7 and r_8); and (ii) if $P(x,z)$ is the unique edge of x (*i.e.,* r_9), then x becomes unbound, *i.e.,* $\mathcal{U}(x)$ is extended with $P(x,z)$. Rule r_{12} removes an atom from queries. More specifically, if y is unbound (*i.e.,* $g{\in}(\mathcal{C}^l(x,y) \cap \mathcal{U}(y))$, then we can remove the edge $P(x,y)$ (*i.e.,* $\mathcal{C}^l(x,y){\cup}\{gr(g,I)\}$) and record that y can be omitted (*i.e.,* $C^o(y){\cup}\{gr(g,I)\}$). Note that the operator $\|$ means that both set $\mathcal{C}^l(x,y)$ and $\mathcal{C}^o(y)$ are updated.

Algorithm 1: GenOGP

```
Input: A CQ $q$ and a TBox $\mathcal{T}$.
Output: An OGP $Q$ such that $Q \equiv_{\mathcal{T}} q$.
1 initialize an initial OGP $Q$ and the sets $\mathcal{U}(\cdot)$ based on $q$;
2 repeat  /* Iteratively extend condition sets */
3     $\langle Q, \mathcal{U} \rangle \leftarrow$ CondDeduction($\mathcal{T}, Q, \mathcal{U}$);
4     $\langle Q, \mathcal{U} \rangle \leftarrow$ LazyReduction($\mathcal{T}, Q, \mathcal{U}$);
5 until no more updates to all condition sets;
6 return $Q$;
```

Example 8: Consider again Example 7. Given CQ q_e, we initialize the set $\mathcal{U}(y_i)$ ($i \in [2,n]$) with condition $P_i(x,\_)$ for each variable y_i, since such variables appear only once in q_e and do not carry any label in q_e. We then apply rule r_3 to add condition $P_1(x,\_)$ to every set $\mathcal{U}(y_i)$. Then the edge (x,y_i) of the generated OGP can carry label P_1 or P_i. Therefore, such OGP is equivalent to the exponentially large UCQ produced by unfolding $\bigwedge_{i\in[2,n]}(P_1(x,y_i) \vee P_i(x,y_i))$. □

(3) Lazy reduction. To avoid performing Reduction exponentially many times, we adopt a lazy strategy. Recalling Example 7, although exponentially many CQs are generated during the reduction, they are equivalent to the CQ $P_1(x,y_1)$. This is because (a) variables $y_2,\ldots,y_n$ appear only once in q_e, and are unbound; and (b) atoms $P_2(x,y_2),\ldots,P_n(x,y_n)$ can be reduced and merged into $P_1(x,y_1)$. Note that after the reduction, new inclusions may be applied, since the variable x becomes unbound in the generated CQs. Therefore, we adopt the following lazy strategy: conduct the reduction only when such variable x can become unbound. We verify the conditions for Reduction by inspecting condition sets in OGPs and $\mathcal{U}(\cdot,\cdot)$. Observe that if x cannot become unbound after reduction, then we do not invoke Reduction, since the generated queries after the reduction are equivalent to the original query, and we would not lose any new query.

B. The Rewriting Algorithm

Algorithm. Putting these together, we present the rewriting algorithm GenOGP in Algorithm 1. Given a CQ q and a TBox $\mathcal{T}$, GenOGP generates an OGP Q such that $Q \equiv_{\mathcal{T}} q$. It first constructs an initial OGP Q from q, and then constructs condition sets based on Q (line 1). After these, it iteratively extends these sets by interleaving two procedures: CondDeduction and LazyReduction (lines 3-5). When no more condition is generated, it returns the constructed OGP Q (line 6).

Procedure CondDeduction. It deduces new conditions to extend the condition sets, by applying rules in Table II. In contrast to PerfectRef, which applies inclusions to atoms of CQs, CondDeduction (1) applies rules in Table II to the OGP Q, to deduce new conditions, and (2) repeatedly applies rules to add new conditions until no more condition can be deduced. This is to facilitate the lazy reduction strategy.

Example 9: Consider the CQ q in Example 3, and the TBox $\mathcal{T}$ in Example 2. GenOGP first constructs an initial OGP Q.

It then initializes the condition sets based on Q (step (1) in Table III), *e.g.,* since z in Q is an unbound vertex with edge $(x, \mathsf{takesCourse}, z)$, condition $\mathsf{takesCourse}(x,z)$ is added to the set $\mathcal{U}(z)$. Then CondDeduction extends these sets by applying rules in Table II (step (2) in Table III), *e.g.,* since $\mathcal{T}$ has an inclusion T_1: Student$\sqsubseteq\exists$takesCourse and condition $\mathsf{takesCourse}(x,z)$ is in $\mathcal{U}(z)$, rule r_{12} is applied to add atom $\mathsf{Student}(x)$ to $\mathcal{C}^o(z)$. □

Procedure LazyReduction. It is to reduce redundant edges in an OGP Q. More specifically, to conduct a reduction on a vertex x in Q, it checks the following.

(1) All edges adjacent to x can be reduced, *i.e.,* (a) all these edges have the same direction and label; (b) at most one neighbor y of x is not unbound. For such case, (i) it first identifies the direction and label of common edges of x by computing the intersection of matching conditions of its adjacent edges; and (ii) it next checks whether at most one neighbor is unbound, *i.e.,* checks whether the omission conditions of its neighbors are not empty.

When *not all* adjacent edges of x have the same direction and label, we can also conduct reduction if these edges can be divided into two categories such that (a) the first category C_1 satisfies the reduction condition (*i.e.,* the same direction and label), and (b) the other category C_2 can be omitted due to x and edges in C_1. We can first reduce the edges in C_2 based on the omitted conditions, and then reduce edges in C_1 as above.

(2) Vertex x can be unbound after reduction, *i.e.,* x appears only once. Let $P(x,y)$ be the unique edge of x. Then it adds $P(x,y)$ to the set $\mathcal{U}(x)$, and new rules can be applied.

Example 10: After the condition sets cannot be extended by CondDeduction in Example 9, LazyReduction is called, since (1) all edges incident to y_1 in Q carry label advisorOf and are linked to vertices x, y_2, y_3, and (2) both y_2 and y_3 carry the label wildcard '$*$'. Since x is distinguished (see Section II), vertices y_2 and y_3 are removed. Moreover, since y_1 become unbounded after the reduction, we further construct the set $\mathcal{U}(y_1)$. Note that if we do not adopt the lazy reduction strategy, and first reduce atoms $\mathsf{advisorOf}(y_1,x)$ and $\mathsf{advisorOf}(y_1,y_3)$, then the generated OGP may contain redundant conditions.

After the reduction, CondDeduction applies the rule r_{12} to remove vertex y_1, *i.e.,* add $\mathsf{Student}(x)$ and $\mathsf{PhD}(x)$ to $\mathcal{C}^o(y_1)$. Now no condition is generated, and the algorithm terminates.

The matching conditions and the omission conditions in the generated OGP is given in step (4) in Table III. When such OGP is evaluated on ABox $\mathcal{A}$={PhD(Ann)}, since $\mathsf{PhD}(x)$ exists in the omission condition of y_1 and z, and the existences of y_2 and y_3 are also depended on the matches of y_1, we only need to evaluate $\mathsf{PhD}(x)$ on ABox $\mathcal{A} = \{\mathsf{PhD(Ann)}\}$, and deduce the results Ann, as expected. □

Theorem 1: *Given a conjunctive query q and* TBox $\mathcal{T}$, *it is in* PTIME *to generate an* OGP Q_o *such that* $Q_o \equiv_{\mathcal{T}} q$. □

Proof sketch: We start with the correctness of GenOGP (*i.e.,* $Q_o \equiv_{\mathcal{T}} q$) and then show GenOGP is in PTIME.

TABLE III
THE PROCESS OF EXTENDING CONDITION SETS

Step (1): Initialization	**Step (2): CondDeduction**
$\mathcal{C}^l(y_1,x) = \{\boldsymbol{\mathsf{advisorOf}(y_1,x)}\}$ $\mathcal{C}^l(y_1,y_i) = \{\boldsymbol{\mathsf{advisorOf}(y_1,y_i)}\}$, $i = 2,3.$ $\mathcal{C}^l(x,z) = \mathcal{U}(z) = \{\boldsymbol{\mathsf{takesCourse}(x,z)}\}$	$\mathcal{C}^l(y_1,x) = \{\mathsf{advisorOf}(y_1,x)\}$ $\mathcal{C}^l(y_1,y_i) = \{\mathsf{advisorOf}(y_1,y_i)\}$, $i = 2,3.$ $\mathcal{C}^l(x,z) = \mathcal{U}(z) = \{\mathsf{takesCourse}(x,z)\}$ $\mathcal{C}^o(z) = C^l(x) = \{\boldsymbol{\mathsf{Student}(x)}, \boldsymbol{\mathsf{PhD}(x)}\}$
Step (3): LazyReduction	**Step (4): CondDeduction**
$\mathcal{C}^l(y_1,x) = \{\mathsf{advisorOf}(y_1,x)\}$ $\mathcal{C}^l(y_1,y_i) = \{\mathsf{advisorOf}(y_1,y_i)\}$, $i = 2,3.$ $\mathcal{C}^l(x,z) = \mathcal{U}(x) = \{\mathsf{takesCourse}(x,z)\}$ $\mathcal{C}^o(z) = \mathcal{C}^l(x) = \{\mathsf{Student}(x), \mathsf{PhD}(x)\}$ $\mathcal{C}^o(y_2) = \mathcal{C}^o(y_3) = \{\boldsymbol{\mathsf{advisorOf}(y_1,x)}\}$ $\mathcal{U}(y_1) = \{\boldsymbol{\mathsf{advisorOf}(y_1,x)}\}$	$\mathcal{C}^l(y_1,x) = \{\mathsf{advisorOf}(y_1,x)\}$ $\mathcal{C}^l(y_1,y_i) = \{\mathsf{advisorOf}(y_1,y_i)\}$, $i = 2,3.$ $\mathcal{C}^l(x,z) = \mathcal{U}(x) = \{\mathsf{takesCourse}(x,z)\}$ $\mathcal{C}^o(z) = \mathcal{C}^l(x) = \{\mathsf{Student}(x), \mathsf{PhD}(x)\}$ $\mathcal{C}^o(y_2) = \mathcal{C}^o(y_3) = \{\mathsf{advisorOf}(y_1,x)\}$ $\mathcal{U}(y_1) = \{\mathsf{advisorOf}(y_1,x)\}$ $\mathcal{C}^o(y_1) = \{\mathcal{C}^l(\boldsymbol{x})\}$

Note: Newly added conditions are in **bold** type.

Correctness. We show that $Q_o \equiv_{\mathcal{T}} q$. We prove this by induction on the steps that GenOGP generates Q_o. Let $Q_o^1, \ldots, Q_o^M$ be the sequence of OGPs generated by GenOGP, and $Q_o^1 = q$. We construct two sequences $Q_L^1, \ldots, Q_L^S$ and $Q_U^1, \ldots, Q_U^K$ of CQs generated by PerfectRef such that each OGP Q_o^i is bounded by Q_L^i and Q_U^i, *i.e.,* $Q_L^i \sqsubseteq_{\mathcal{T}} Q_o^i \sqsubseteq_{\mathcal{T}} Q_U^i$. If these hold, since PerfectRef returns the same query regardless of in what order the rules are applied [13], we have that $Q_L^S \equiv_{\mathcal{T}} Q_U^K \equiv_{\mathcal{T}} Q_o^M$. Therefore, $Q \equiv_{\mathcal{T}} Q_o$ follows. Although these sequences may have different lengths, since both PerfectRef and GenOGP terminate when no more updates can be made, we assume that all sequences terminate in N steps, where N is the maximum number among S, M and K.

Complexity. GenOGP runs in $O(|q|^2|\mathcal{T}|^2)$ time, where $|\mathcal{T}|$ is the number of inclusions in $\mathcal{T}$. Observe that (1) constructing an initial OGP Q is in $O(|q|)$ time (line 1); (2) initializing condition sets $\mathcal{U}(\cdot)$ is in $O(|q|)$ time, because these sets consist of only labels in Q (line 2); (3) the iteration takes $O(|q|^2|\mathcal{T}|^2)$ time (lines 3–5), since for each vertex u (resp. edge e), it has at most $|\mathcal{T}|$ conditions in $\mathcal{C}^l(u)$ and $\mathcal{C}^o(u)$ (resp. $\mathcal{C}^l(e)$), and it generates a new condition by deduction and lazy reduction in $O(|\mathcal{T}|)$ and $O(|q|)$ time, respectively. □

Remark. We can show that generating minimal OGP is NP-hard by a reduction from the conjunctive query minimization problem [58]. Intuitively, we first define the ontology $\mathcal{T} = \emptyset$, and then show that generating minimum OGP is the same as to find the minimum CQ query, which is NP-hard [58].

V. Matching Ontological Patterns

In this section we develop an algorithm OMatch to compute the matches $Q(G)$ of a given OGP Q in a graph G. We design OMatch by extending an SOTA algorithm DAF for subgraph isomorphism [28]. Note that although OGP adopts homomorphic semantics, rather than isomorphism semantics as in [28], we can still exploit the developed techniques in [28] to accelerate the computation. Actually, we can extend any SOTA matching algorithms to match OGPs, *e.g.,* [52], [26]. As a case study, we extend DAF [28] due to its effective pruning strategy.

A. Overview of Algorithm OMatch

We first give a review of DAF and an overview of our extensions (Section V-A), and then present OMatch (Section V-B).

Algorithm DAF. It computes the matches of conventional patterns, *i.e.,* an OGP $Q = (V_Q, E_Q, L_Q, \mathcal{C}^l, \mathcal{C}^o)$, in which neither $\mathcal{C}^l$ nor $\mathcal{C}^o$ specifies any condition. Given such a pattern Q and a graph G, it computes the matches $Q(G)$ with three procedures: (1) BuildDAG, to build a rooted directed acyclic graph (DAG) Q_D from Q by conducting a BFS on Q from a root vertex u_r that has a large degree and a small candidate set $C(u_r)$; here the set $C(u)$ maintains all candidates of vertex u, and initially contains all vertices in G that have the same label as u; (2) BuildCS, to build an index structure CS of polynomial-size and record both candidates of each vertex in V_Q and edges between these candidates; and (3) Backtrack, to enumerate all the matches of Q in G by accessing CS only.

BuildCS. Given the DAG Q_D and a graph G, (1) it first uses dynamic programming to refine the candidate set $C(u)$ for each vertex u in V_Q; more specifically, a candidate v exists in $C(u)$ if and only if for each vertex u' adjacent to u in Q_D, there exists a candidate $v' \in C(u')$ such that v' is also adjacent to v in G; and (2) then it extends the structure CS by creating edges between these candidates, *i.e.,* for each edge (u, l, u') in Q_D, each vertex $v \in C(u)$ and each vertex $v' \in C(u')$, if there exists an edge (v, l, v') in G, then add an edge (v, l, v') to CS. To simplify the description, for each edge (u, l, u') in Q_D and each vertex $v \in C(u)$, denote by $N^u_{u'}(v)$ the set of all vertices v' that are in $C(u')$ and adjacent to v in G.

Backtrack. Given the DAG Q_D and the structure CS, it computes all matches h of Q in G by recursively mapping each vertex u in V_Q to a vertex v in the candidate set $C(u)$. More specifically, (1) it first maps the root u_r to a candidate v in $C(u_r)$; (2) then it updates the candidate set $C(u')$ for each child u' of u_r based on v, *i.e.,* $C(u') = C(u') \cap N^u_{u'}(v)$; (3) after that it maps a child u' of u_r to its candidates if all of its parent vertices in Q_D have been mapped. When multiple children are available, it prefers the child having the minimum candidate sets (*i.e.,* the candidate-size matching order [28]). But if all children u' of u_r have empty candidate sets, it backtracks and maps u_r to another candidate v' in $C(u_r)$. It repeats steps (1)-(3) until all vertices in V_Q have been mapped.

Challenges. Since both vertices and edges in OGPs can carry conditions, these give rise to new challenges.

(1) A vertex u may be omitted in matches of OGPs (*i.e.,* $\mathcal{C}^o(u) \neq \emptyset$), but cannot be omitted in matches of conventional patterns. Hence, the structure CS may be unsound for OGPs, *i.e.,* it misses valid matches when u is omitted.

(2) Conditions $\mathcal{C}^l(u)$ and $\mathcal{C}^o(u)$ may involve other vertices, *i.e.,* matching u depends on the properties of other vertices. How can we capture such dependencies to prune candidates of u?

(3) Condition evaluation can be costly. Consider global conditions that inspect non-local information (see below for the definition). Since BuildCS of DAF inspects only properties of the two vertices on an edge, the values of global conditions can be determined only in Backtrack, and will be verified for all matches of an OGP in G, the number of which is exponentially large. Here *global conditions* are (a) conditions on edges (u, l, v) that involve vertices other than u and v, or (b) conditions on vertices u that involve another vertices u' with $u \neq u'$.

Extensions. To address these, we extend DAF as follows.

Algorithm 2: OMatch

```
Input: An OGP $Q$ and a graph $G$.
Output: Matches $Q(G)$ of $Q$ in $G$.
1 $Q_D \leftarrow$ BuildOMDAG($Q, G$);
2 OMCS $\leftarrow$ BuildOMCS($Q, Q_D, G$);
3 $Q(G) \leftarrow$ OMBacktrack($Q, Q_D$, OMCS);
4 return $Q(G)$;
```

(1) For each vertex u with $\mathcal{C}^o(u) \neq \emptyset$ (*i.e.,* can be omitted), we add a *dummy* vertex $\perp$ to $C(u)$. Then u is mapped to $\perp$ if and only if $\mathcal{C}^o(u)$ is evaluated to be true. This helps prune matches in BuildCS and enumerate matches in Backtrack.

(2) To capture dependencies between conditions, we extend the DAG Q_D and the structure CS as follows: if a condition on vertex u involves vertex u', we add an edge (u', u) to Q_D. Then we use $C(u')$ to refine the candidate set $C(u)$, and store edges between $C(u')$ and $C(u)$ in CS. The edge (u', u) ensures that u' is mapped before u in Backtrack. Denote by OMDAG and OMCS the extended DAG and CS, respectively.

(3) To accelerate the verification of global conditions, (a) we add additional entries to OMCS to cache computed global conditions [59]; when verifying the global conditions, we can first check whether its value exists in the caches; and (b) we also use a shared binary decision diagram (SBDD) [60] to simplify and share the computation of multiple conditions.

B. The Matching Algorithm

We now present the matching algorithm OMatch.

Algorithm. The outline of OMatch is in Algorithm 2. Given an OGP Q and a graph G, it computes the matches $Q(G)$ as follows. It first builds OMDAG Q_D from Q that captures the dependencies between conditions in Q (line 1). Then it creates OMCS from Q_D (line 2), and recursively enumerates the matches $Q(G)$ (line 3). Finally, it returns $Q(G)$ (line 4).

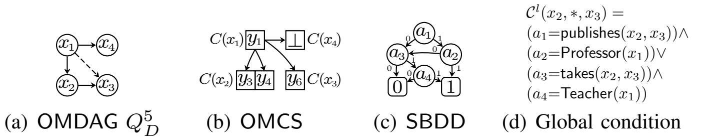
Procedure BuildOMDAG. It builds OMDAG Q_D from an OGP Q. Recall that OMDAG differs from the DAG of DAF in that it has edges to represent dependencies between conditions.

(1) It first initializes OMDAG by performing the following steps on each vertex u in Q: (a) initialize the candidate set $C(u)$ based on the label and non-global conditions of u; (b) add a dummy candidate $\perp$ to $C(u)$ if $\mathcal{C}^o(u) \neq \emptyset$; and (c) add an edge (u', u) to Q, if either $\mathcal{C}^l(u)$ or $\mathcal{C}^o(u)$ involves vertex u'.

(2) It conducts a BFS from root u_r to build Q_D as BuildDAG, except that it prefers root u_r that does not depend on other vertices, *i.e.,* $\mathcal{C}^l(u_r)$ and $\mathcal{C}^o(u_r)$ do not involve other vertices.

Procedure BuildOMCS. It builds an auxiliary structure OMCS from OMDAG Q_D. To retain the soundness and equivalence, it (1) revises the pruning strategy in BuildCS of DAF, and (2) stores additional entries for global conditions in Q.

(1) It first refines the candidate sets using non-global conditions, and then initializes OMCS by creating edges between the candidates as in DAF. Note that (a) it does not process vertices and edges with global conditions, since these conditions cannot be determined now; and (b) it does not prune candidates for vertex u when $\mathcal{C}^o(u) =$ true, since u can be omitted.



$\mathcal{C}^l(x_2,*,x_3) =$
$(a_1{=}\text{publishes}(x_2,x_3))\wedge$
$(a_2{=}\text{Professor}(x_1))\vee$
$(a_3{=}\text{takes}(x_2,x_3))\wedge$
$(a_4{=}\text{Teacher}(x_1))$

(a) OMDAG Q_D^5 (b) OMCS (c) SBDD (d) Global condition

Fig. 3. Demonstration of OMatch

(2) It then stores entries for global conditions in Q to avoid probing G during enumerating the matches. To conduct this, (1) if a global condition involves $l(u)$ (resp. $u.A$), then for each candidate $v \in C(u)$ with label l (resp. attribute $v.A = a$), OMCS stores an entry $l(v)$=true (resp. $v.A$=a); and (2) if a global condition carries an edge condition $l(u, u')$, then for each $v{\in}C(u)$ and $v'{\in}C(u')$ such that there exists an edge from v to v' labeled l in G, OMCS contains an entry $l(v, v') =$ true.

Example 11: Recall OGP Q'_5 from Example 4 and graph G from Example 5. BuildOMDAG (1) first initializes $C(x_1) = \{y_1, y_2\}$, $C(x_2){=}\{y_3, y_4\}$, $C(x_3){=}\{y_5, y_6\}$ and $C(x_4){=}\{\perp\}$ based on the labels and non-global conditions of vertices; then (2) it builds an OMDAG Q_D^5 rooted at x_1 with an additional edge (x_1, x_3) (Fig. 3(a)), since the condition $\mathcal{C}_5^l(x_3)$ on vertex x_3 depends on x_1. After that, (3) BuildOMCS refines the candidate sets, *i.e.,* y_2 is pruned from $C(x_1)$ and y_5 is pruned from $C(x_3)$. Finally, (4) it creates the structure OMCS (Fig. 3(b)).

In contrast to DAF, (1) OMCS prunes y_5 from $C(x_3)$ due to the edge (x_1, x_3) in Q_D^5; and (2) it stores Teacher(y_1) = true, takes(y_3, y_6) = true and takes(y_4, y_6) = true, to accelerate the computation of $\mathcal{C}_5^l(x_2, *, x_3)$. □

Procedure OMBacktrack. It computes the matches $Q(G)$. Apart from Backtrack in DAF, (1) it constructs a SBDD to simplify and represent all global conditions in Q before starting the enumeration; (2) it updates the values of the conditions involving vertex u, once u is mapped to a new vertex $v \in C(u)$ using the stored entries in OMCS; and (3) it updates the value of a condition through the SBDD if all vertices in the condition have been mapped.

Example 12: Continuing with Example 11, OMBacktrack computes the matches as follows: (1) it first constructs a SBDD for global condition $\mathcal{C}_5^l(x_2, *, x_3)$ (Figures 3(c)–3(d)); (2) it then maps the root x_1 of Q_D^5 to its only candidate y_1, marks vertices x_2, x_3 and x_4 as extendable; and uses OMCS to update their candidates; (3) since both x_3 and x_4 have a unique candidate, it maps x_3 to y_6 and x_4 to $\perp$; (4) then it maps x_2 to y_3, verifies that $\mathcal{C}_5^l(x_2, *, x_3)$ is true, and finds a match; (5) it finally maps x_2 to y_4 and finds another match. □

Analysis. We next show that OMatch retains the same complexity as DAF, and OMCS is sound and equivalent [28].

Time complexity. OMatch runs in $O(|G|^{|Q|})$, the same as DAF, since (1) BuildOMDAG is in $O(|Q|)$ time to build OMDAG Q_D; although it adds additional edges to Q_D, it guarantees that Q_D has no duplicate edges; (2) BuildOMCS takes (a) $O(|Q|{\cdot}|G|)$ time to construct the structure OMCS, and (b) $O(|G|)$ time to store the additional entries; and (3) OMBacktrack is in $O(|G|^{|Q|})$ time, because the number of all possible matches of Q in G is bounded by $\prod_{u \in V_Q} |C(u)|$.

TABLE IV
STATISTICS OF DATASETS AND ONTOLOGIES

Name	$\|D\|$	$\|V\|$	$\|E\|$	$\|O\|$	$\|\Sigma_V\|$	$\|\Sigma_E\|$	Domain
DBpedia	29.7M	4.1M	13.4M	1.7K	512	833	Wikipedia
NPD	3.8M	1.6M	2.3M	566	354	173	Petroleum
LUBM$_{100}$	13.9M	3.3M	11.1M	86	43	32	University
LUBM$_{200}$	27.6M	6.6M	22.1M				
LUBM$_{300}$	41.3M	9.8M	33M				
LUBM$_{400}$	55.3M	13.1M	44.2M				
OWL2Bench$_{100}$	7.3M	1.8M	6.7M	375	136	121	
OWL2Bench$_{200}$	14.6M	3.4M	13.5M				
OWL2Bench$_{300}$	22M	5M	20.4M				
OWL2Bench$_{400}$	29.4M	6.6M	27.2M				

Space complexity. The space required by OMatch is dominated by the structure OMCS, which is in $O(|Q|{\cdot}|G|)$ and is the same as the structure CS of DAF, since OMCS differs from CS only in its additional entries, which record attributes, labels and edges in G at most once, and are bounded by $O(|G|)$.

Properties of OMCS. We show that OMCS is sound, and equivalent to G; *i.e.,* (1) for each match h of Q and each vertex u in Q, if $h(u) = v$, then $v \in C(u)$ (*i.e.,* soundness); and (2) graph G is unnecessary after building the OMCS (*i.e.,* equivalence). One can verify that (1) OMCS is sound with the dummy candidate $\perp$ and revised pruning strategy, and (2) OMCS is equivalent to G, due to its additional entries.

VI. Experimental Study

Using real-life and synthetic data, we conducted four sets of experiments to evaluate the (1) efficiency, (2) effectiveness, (3) scalability and (4) end-to-end performance of our approach.

Experimental setting. We start with the setting.

Datasets and ontologies. We used two real-life datasets and two synthetic datasets, all with ontologies: (1) DBpedia [61], a large knowledge base with dataset and ontology provided by [41], which enriched the original DBpedia ontology with a tourism ontology; (2) NPD [9], a realistic dataset with an ontology about petroleum activities on the Norwegian continental shelf; (3) LUBM [27] and OWL2Bench [62], two synthetic university domain benchmarks with data generators; for each of the two, we varied the number of universities from 100 to 400 to generate four versions of datasets with distinct scaling factors. *e.g.,* both LUBM$_{100}$ and OWL2Bench$_{100}$ contain 100 different universities. The detailed statistics of these datasets and ontologies are summarized in Table IV, where (1) $|D|$ is the number of triples (*i.e.,* membership assertions) in the dataset; (2) $|V|$ (resp. $|E|$) is the number of vertices (resp. edges) in the transformed graph (see below); and (3) $|O|$, $|\Sigma_V|$ and $|\Sigma_E|$ are the numbers of axioms (*i.e.,* the size of TBox), distinct concepts and roles in the ontology, respectively.

Note that (1) since these datasets are all based on RDF data model, we transformed them into graphs using the type-aware transformation [52]; and (2) we chose the OWL 2 QL [31] version of LUBM and OWL2Bench, and used OWL API [63] to remove axioms in the ontologies of DBpedia and NPD that fall outside the OWL 2 QL profile.

Queries. We generated conjunctive queries (CQs) using a random walk strategy, as in many pattern matching studies [46],

[29], [28], [49], [48]. For each of DBpedia, NPD, LUBM_{100} and $\mathsf{OWL2Bench}_{100}$, we generated four query sets $\mathcal{Q}_i$ ($i \in \{4, 8, 12, 16\}$). Each $\mathcal{Q}_i$ contains 100 queries, where each query Q in $\mathcal{Q}_i$ has (1) i atoms of the form $A(x)$ or $P(x, y)$ (*i.e.,* the size $|Q| = i$), and (2) at least 1 and up to 10^8 answers to avoid excessive evaluation time. For each query, we randomly marked some variables as distinguished (see Section II).

We took care to ensure that the associated ontology can constrain each query Q, *i.e.,* there exist some rules in Table II that can be applied to atoms of Q. To achieve this, we randomly picked some atoms of Q, and replaced them with more generalised ones according to the axioms in the ontology. For instance, if there exists an axiom $A_1 \sqsubseteq A_2$ (resp. $P_1 \sqsubseteq P_2$) in the ontology, then we replaced the atom $A_1(x)$ (resp. $P_1(x, y)$) in Q with more generalised concept $A_2(x)$ (resp. role $P_2(x, y)$).

To evaluate the performance of GenOGP and OMatch in practical situations, we adopted the OWL 2 QL version of benchmark queries provided by LUBM and OWL2Bench, which contain 14 and 10 queries, respectively. We also randomly selected 10 queries from the LSQ dataset [64], [65], which includes SPARQL queries issued to DBpedia by users.

Algorithms. We implemented our (a) rewriting algorithm GenOGP (Section IV) and (b) matching algorithm OMatch (Section V) in C++. We also considered (c) $\mathsf{OMatch}_{\mathsf{BFS}}$, a variant of OMatch that uses a static BFS matching order [48], to demonstrate the effectiveness of OMatch. Note that we stored the data into memory, *i.e.,* our algorithms directly access data, without using any database engine for optimization.

We compared with 8 baselines: (1) Iqaros [17], a UCQ-rewriting algorithm; (2) Rapid+gStore, where we first used Rapid [14] to generate a UCQ-rewriting, then transformed the rewriting into an equivalent SPARQL query with UNION operators [57], and finally exploited the RDF graph database gStore [51] for evaluation; (3) Graal [18], where we took their compilation-based rewriting algorithm $\mathsf{Pure}_{\mathsf{C}}$ to generate UCQ-rewritings, and chose their memory-based graph store for evaluation; (4) CLIPPER [15], which rewrites a CQ into a datalog program and evaluates it using the DLV system [66]; (5) Ontop [67], where we adopted their datalog-rewriting algorithm tree-witness [19], and deployed the relational database PostgreSQL for evaluation; (6) Drewer [16], a datalog-rewriting system that uses VLog [42] as the evaluation reasoner; (7) PAGOdA [41], a query answering system that combines the datalog reasoner RDFox [43] and the OWL 2 reasoner HermiT [68]; (8) Stardog [69], an enterprise knowledge graph with Pellet [70] as the reasoner. All these baselines are implemented in Java, except that some query evaluation parts are written in C/C++ (*e.g.,* gStore, DLV and VLog).

Note that (1) Iqaros has no evaluation stage; (2) PAGOdA has no rewriting phase; and (3) the rewriting part of Stardog is in its evaluation part, and the two cannot be separated [62]. We also made attempts to use the state-of-the-art reasoners VLog and RDFox as saturation-based systems [55], but they failed to run on any dataset and cannot return any result, since the saturation leads to excessive memory usage.

We set a time limit of 10 and 30 minutes for query rewriting and evaluation, respectively, so that our experiments can terminate within reasonable time. For any query, if the rewriting or evaluation algorithms exceeded the time limit or stopped with an error, then we marked the query as *unsolved* and regarded the time limit as its running time, as in [49], [29].

Environment. We ran experiments on a server with 2.20GHz Intel Xeon Silver CPU, 1.8TB SSD and 256GB RAM. Each experiment was run 5 times, and the average is reported here. Due to the space limits, we only show results on some datasets, and the results on the other datasets are consistent.

Experimental results. We now report our findings.

Exp-1: Efficiency. We tested the impact of the query size and the ontology size on the efficiency of GenOGP and OMatch.

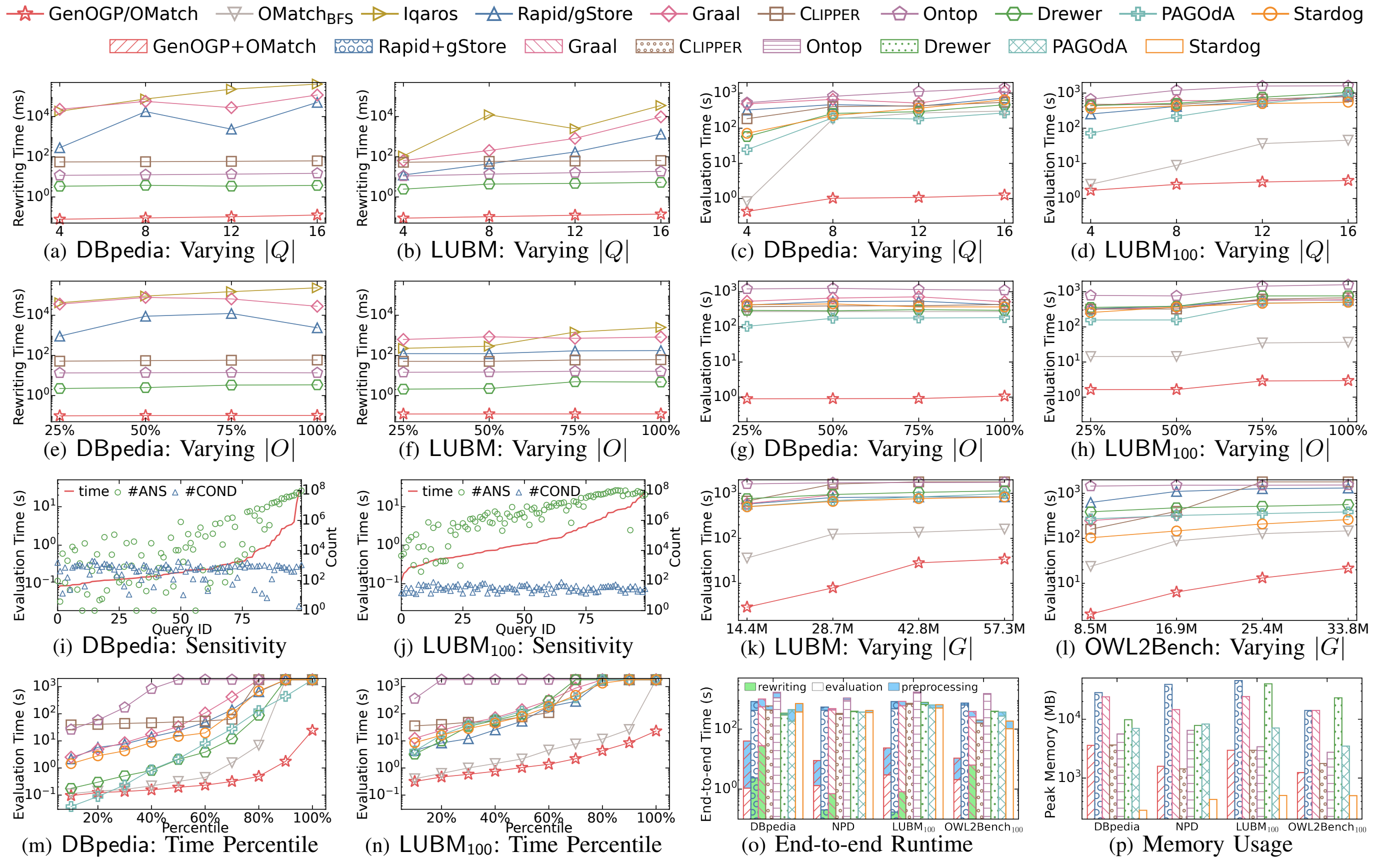
Varying $|Q|$. We varied the size $|Q|$ of the CQ query from 4 to 16 to evaluate the efficiency of GenOGP and OMatch.

(1) Query rewriting. Figures 4(a)–4(b) report the performance of GenOGP. We find that (a) GenOGP and the datalog-rewriting algorithms are insensitive to $|Q|$; they take moderately longer when $|Q|$ gets larger. In contrast, the runtime of all UCQ-rewriting algorithms fluctuate with $|Q|$, since they fail to rewrite some queries due to the exponentially large rewriting, *e.g.,* on LUBM, Iqaros cannot terminate within the time limit on 2 queries when $|Q| = 8$, but solves all queries when $|Q| = 12$. (b) GenOGP consistently outperforms all the baselines, since it conducts the rewriting on a single graph pattern, rather than on exponentially many CQs or rules. On average, it takes 0.1ms. When $|Q| = 16$, GenOGP outperforms the fastest baseline Drewer by 29.9× and 38.2× on DBpedia and LUBM, respectively. And GenOGP beats the other baselines by 2–6 orders of magnitude.

(2) Query evaluation. As shown in Figures 4(c)–4(d), (a) OMatch takes longer when $|Q|$ gets larger, as expected. (b) OMatch is able to match reasonably large OGPs. When $|Q| = 16$, OMatch on average takes 1.3s and 3.2s on DBpedia and LUBM_{100}, respectively, while all baselines take hundreds of seconds and have dozens of unsolved queries. For example, the fastest baseline PAGOdA on average takes 272s (resp. 921s) on DBpedia (resp. LUBM_{100}) and fails to solve 10 (resp. 40) queries. (c) OMatch on average beats all baselines by 2–3 orders of magnitude, since handling conditions in OGPs is more lightweight than handling exponentially many CQs or rules. (d) OMatch also beats the variant $\mathsf{OMatch}_{\mathsf{BFS}}$ by an average of 2 orders of magnitude and 9× on DBpedia and LUBM_{100}, respectively. This shows the effectiveness of the matching order on the performance of OMatch.

Varying $|O|$. Fixing $|Q| = 12$, we varied the scaling factor of the ontology from 25% to 100% to evaluate the impact of the ontology size on the efficiency of GenOGP and OMatch.

(1) Query rewriting. As shown in Figures 4(e)–4(f), (a) on average, GenOGP is 28.5× faster than the fastest baseline Drewer, and beats the other datalog-rewriting algorithms CLIPPER and Ontop by 2 orders of magnitude; and GenOGP outperforms the UCQ-rewriting algorithms by 3–6



(a) DBpedia: Varying $|Q|$ (b) LUBM: Varying $|Q|$ (c) DBpedia: Varying $|Q|$ (d) LUBM$_{100}$: Varying $|Q|$

(e) DBpedia: Varying $|O|$ (f) LUBM: Varying $|O|$ (g) DBpedia: Varying $|O|$ (h) LUBM$_{100}$: Varying $|O|$

(i) DBpedia: Sensitivity (j) LUBM$_{100}$: Sensitivity (k) LUBM: Varying $|G|$ (l) OWL2Bench: Varying $|G|$

(m) DBpedia: Time Percentile (n) LUBM$_{100}$: Time Percentile (o) End-to-end Runtime (p) Memory Usage

Fig. 4. Performance evaluation

orders of magnitude. Even when the scaling factor is 25%, GenOGP is on average 20× and 3 orders of magnitude faster than the fastest datalog-rewriting algorithm Drewer and the fastest UCQ-rewriting algorithm Rapid, respectively. (b) The runtime of GenOGP increases when $|O|$ gets larger, as expected, while the runtime of some baselines may decrease when $|O|$ increases (*e.g.,* Rapid and Graal). This is because (i) they perform several optimizations to reduce the rewriting size and avoid redundant rewritings, and (ii) when more axioms are introduced, more rewritings may become redundant.

(2) Query evaluation. As shown in Figures 4(g)–4(h), (a) the runtime of OMatch increases when $|O|$ increases, as expected. Since some baselines generate rewritings whose sizes fluctuate with $|O|$ as mentioned above, their evaluation times also fluctuate with $|O|$. (b) OMatch and all baselines are more insensitive to $|O|$ on DBpedia than LUBM$_{100}$, since fewer added axioms in DBpedia can be applied to the queries. (c) OMatch beats all baselines by 2–3 orders of magnitude. Even when $|O|$=25%, the fastest baseline PAGOdA on average takes 104s and 157s to process a query on DBpedia and LUBM$_{100}$, respectively, while OMatch takes only 0.9s and 1.6s, respectively. (d) OMatch is faster than its variant OMatch$_{BFS}$, as expected.

Exp-2: Effectiveness. We tested the effectiveness of GenOGP and OMatch, by analyzing the (1) rewriting size, (2) sensitive factors, (3) cumulative distribution of query evaluation time, and (4) performance on real-life queries.

Rewriting size. We compared the rewriting sizes of all rewriting algorithms by measuring the number of atoms [16] in the generated UCQ queries (Iqaros, Rapid, Graal) and datalog queries (CLIPPER, Ontop, Drewer). For GenOGP, we counted the condition sizes of generated OGPs. We find that on DBpedia (resp. LUBM), the rewriting of GenOGP is 2.82× (resp. 16.2×) smaller than the best UCQ-rewriting algorithm Graal, but 4.49× (resp. 5.16×) larger than the best datalog-rewriting algorithm Drewer. This is because (1) GenOGP avoids the exponentially many queries of UCQ-rewriting, and thus has smaller rewriting sizes; but (2) unlike datalog-rewriting, GenOGP is not aimed at minimizing the rewriting size. Actually, the rewriting size has a small impact on the subsequent query evaluation algorithm OMatch (see below).

Sensitivity analysis. We relabeled the query IDs in a query set in the ascending order of the query evaluation time, as in [71], [72]. For each query, we recorded the number of answers (denoted by #ANS) and the condition size of the OGP generated by GenOGP (denoted by #COND).

Figures 4(i)–4(j) depict the results on the query set $\mathcal{Q}_{12}$ of DBpedia and LUBM$_{100}$, respectively. We find that (1) #ANS has a direct impact on the efficiency of OMatch, since more enumeration time are needed to return more answers, and storing these answers also takes time. (2) #COND has small impact on the runtime of OMatch. Although conditions, especially omission conditions, often help prune unnecessary search space, the enumerating time dominates the computation cost of OMatch, and verifying conditions in OGPs only has slight overhead. These findings are similar to CGPs [26].

The worst-case performance. To show the performance of OMatch on hard queires (*i.e.,* queries with the large runtime overhead), we computed the cumulative distribution of the query evaluation time [73]. Figures 4(m)–4(n) report the results on the query set $\mathcal{Q}_{12}$ of DBpedia and LUBM$_{100}$, respectively. The gap between the runtime of OMatch and baselines grows when the percentile increases, which shows the efficiency of OMatch in handling hard queries.

Number of unsolved queries. Since we set a time limit of 30 minutes for query evaluation, Figures 4(m)–4(n) also show the number of unsolved queries of all methods. Only GenOGP+OMatch answers all queries without any failure.

Real-life queries. We further compared the performance of all methods on real-life queries by using 14 (resp. 10 and 10) queries from LUBM (resp. OWL2Bench and DBpedia). The results are detailed in [74]. Different from the queries we randomly generated, these queries are simple, with over 70% of them having fewer than 4 atoms. But the results are similar. GenOGP is consistently the fastest, and OMatch is the fastest in most cases, especially when the query is complex.

Exp-3: Scalability. Fixing $|Q|$=12, we varied $|G| = |V|+|E|$ of LUBM and OWL2Bench from 14.4M to 57.3M and 8.5M to 33.8M, respectively, to evaluate the scalability of OMatch.

As shown in Figures 4(k)–4(l), (a) when $|G|$ gets larger, OMatch and all baselines take longer, as expected. But the runtime of all baselines grows more slow than OMatch, since they all have dozens of unsolved queries (*i.e.,* the queries that cannot finish within 30 minutes); *e.g.,* on LUBM, as $|G|$ increases from 14.4M to 28.7M, the running time (resp. the number of unsolved queries) of Stardog increases from 502s to 661s (resp. 16 to 27), while the runtime of OMatch increases from 3.0s to 7.9s without any unsolved query. (b) OMatch is 37.2× and 16.6× faster than the fastest baseline Stardog on LUBM and OWL2Bench, up to 170× and 49.8×, respectively. (c) OMatch scales well with $|G|$. On average, It takes 34.6s and 21.7s on LUBM$_{400}$ with $|G|$ = 57.3M and OWL2Bench$_{400}$ with $|G|$ = 33.8M, respectively, on which Stardog takes 830s and 261s, respectively. (d) OMatch only fails in one query on LUBM when $|G| \geq$ 42.8M due to out of memory error; in contrast, Stardog, PAGOdA and gStore have 33, 35 and 41 unsolved queries even on LUBM$_{300}$ with $|G|$ = 42.8M, respectively, and the remaining systems have more than 50 unsolved queries. In particular, CLIPPER suffers from memory issues and cannot process any queries when $|G| \geq$ 28.7M.

Exp-4: End-to-end Performance. We finally compared the end-to-end performance of GenOGP+OMatch with all baselines, in terms of end-to-end time and peak memory usage.

End-to-end time. We measured the completion time to answer queries over the $DL\text{-}Lite_{\mathcal{R}}$ ontology. We broken down the time into (a) *rewriting* time, (b) *evaluation* time and (c) *preprocessing* time (including data loading time and potentially indexing time). For disk-based baselines that store data on SSD (*i.e.,* gStore, Ontop and Stardog), the offline time to preload data onto the SSD is counted in preprocessing time, as in [55].

As shown in Fig. 4(o), (a) GenOGP+OMatch outperforms all baselines in all cases *w.r.t.* the end-to-end time. On average, it is 32.5×, 22.2×, 22.9× and 21.9× faster than the memory-based baselines Graal, CLIPPER, Drewer and PAGOdA, respectively. It beats the disk-based baselines Rapid+gStore, Ontop and Stardog by 35.1×, 71.8× and 23.5×, respectively. (b) GenOGP+OMatch spends on average 87.7% of its time to read and store the dataset as a graph (*i.e.,* preprocessing time), while the evaluation time dominates the end-to-end time for all baselines. (c) The preprocessing of GenOGP+OMatch is the fastest, *i.e.,* it takes 39s, 7.6s, 21s and 8.8s on DBpedia, NPD, LUBM$_{100}$ and OWL2Bench$_{100}$, respectively; on average it is 7% faster than the best baseline Drewer.

Memory usage. We recorded the peak memory usage to evaluate the space cost of GenOGP+OMatch. As shown in Figure 4(p), (a) memory-based GenOGP+OMatch outperforms all baselines except Stardog, since Stardog stores data on disk and also adopts query rewriting techniques. (b) on average GenOGP+OMatch spends 5% less memory than the best memory-based baseline CLIPPER. (c) Although gStore and Ontop are disk-based, they construct indices to accelerate the query evaluation, which demand lots of memory. (d) Since the reasoners underlying Drewer and PAGOdA are based on saturation techniques, they take more space.

Summary. We find the following. (1) GenOGP is efficient. On average, it takes 0.1ms to generate an equivalent OGP from a CQ of size 16 and a $DL\text{-}Lite_{\mathcal{R}}$ ontology with 1.7K axioms; and it outperforms the fastest baseline Drewer by 29.9×. (2) OMatch is 2–3 orders of magnitude faster than the state-of-the-art algorithms for ontology-mediated query answering; *e.g.,* on a real-life graph with 4.1M vertices and 13.4M edges, on average OMatch takes 1.3s to answer queries with $|Q|$=16, while the fastest baseline PAGOdA takes 272s. (3) GenOGP+OMatch can answer complex queries effectively, and is also the fastest to process most real-life queries (*i.e.,* over 75% of real-life queries). (4) OMatch scales well with large graphs. *e.g.,* on a graph with 13.1M vertices and 41.2M edges, on average it takes 34.6s to answer queries with $|Q|$=12, while the fastest baseline Stardog takes 830s. (5) GenOGP+OMatch handles ontology-mediated queries efficiently. It is at least 21.9× faster than the baselines.

VII. CONCLUSION

We have proposed ontological graph patterns OGPs, to accelerate ontology-mediated query answering. OGPs attach conditions to both vertices and edges in graph patterns, to increase their expressive power. We have designed a rewriting algorithm GenOGP to encode a conjunctive query over $DL\text{-}Lite_{\mathcal{R}}$ into an equivalent OGP, and developed a matching algorithm OMatch for OGPs. We have experimentally verified the efficiency, effectiveness and scalability of our approaches.

One topic for future work is to explore more applications for OGPs, *e.g.,* multi-query optimization and event prediction.

References

[1] M. Bienvenu, “Ontology-mediated query answering: Harnessing knowledge to get more from data,” in *IJCAI*, 2016, p. 4058–4061.

[2] G. Xiao, D. Calvanese, R. Kontchakov, D. Lembo, A. Poggi, R. Rosati, and M. Zakharyaschev, “Ontology-based data access: A survey,” in *IJCAI*. International Joint Conferences on Artificial Intelligence, 2018, pp. 5511–5519.

[3] T. Schneider and M. Šimkus, “Ontologies and data management: a brief survey,” *KI-Künstliche Intelligenz*, vol. 34, no. 3, pp. 329–353, 2020.

[4] M. Bienvenu and M. Ortiz, “Ontology-mediated query answering with data-tractable description logics,” *Reasoning Web. Web Logic Rules: 11th International Summer School 2015, Berlin, Germany, July 31-August 4, 2015, Tutorial Lectures. 11*, pp. 218–307, 2015.

[5] M. Ortiz and M. Šimkus, *Reasoning and query answering in description logics*. Springer, 2012.

[6] R. Kontchakov, M. Rodriguez-Muro, and M. Zakharyaschev, “Ontology-based data access with databases: A short course,” *Reasoning Web. Semantic Technologies for Intelligent Data Access: 9th International Summer School 2013, Mannheim, Germany, July 30–August 2, 2013. Proceedings*, pp. 194–229, 2013.

[7] A. Gaulton, L. J. Bellis, A. P. Bento, J. Chambers, M. Davies, A. Hersey, Y. Light, S. McGlinchey, D. Michalovich, B. Al-Lazikani *et al.*, “ChEMBL: a large-scale bioactivity database for drug discovery,” *Nucleic acids research*, vol. 40, no. D1, pp. D1100–D1107, 2012.

[8] B. M. Good, K. Van Auken, D. P. Hill, H. Mi, S. Carbon, J. P. Balhoff, L.-P. Albou, P. D. Thomas, C. J. Mungall, J. A. Blake *et al.*, “Reactome and the gene ontology: Digital convergence of data resources,” *Bioinformatics*, vol. 37, no. 19, pp. 3343–3348, 2021.

[9] M. G. Skjæveland, E. H. Lian, and I. Horrocks, “Publishing the norwegian petroleum directorate’s factpages as semantic web data,” in *ISWC*, vol. 8219, 2013, pp. 162–177.

[10] J. Joy and V. G. Renumol, “An ontology-based hybrid e-learning content recommender system for alleviating the cold-start problem,” *Educ. Inf. Technol.*, vol. 26, no. 4, pp. 4993–5022, 2021.

[11] F. Colace, M. D. Santo, and M. Gaeta, “Ontology for e-learning: a case study,” *Interact. Technol. Smart Educ.*, vol. 6, no. 1, pp. 6–22, 2009.

[12] D. Bursztyn, F. Goasdoué, and I. Manolescu, “Teaching an rdbms about ontological constraints,” in *Very Large Data Bases*, 2016.

[13] 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. Reason.*, vol. 39, no. 3, pp. 385–429, Oct. 2007.

[14] A. Chortaras, D. Trivela, and G. Stamou, “Optimized query rewriting for owl 2 ql,” in *CADE*, 2011, pp. 192–206.

[15] T. Eiter, M. Ortiz, M. Simkus, T. Tran, and G. Xiao, “Query rewriting for horn-shiq plus rules,” in *AAAI*, 2012.

[16] Z. Wang, P. Xiao, K. Wang, Z. Zhuang, and H. Wan, “Query answering for existential rules via efficient datalog rewriting,” in *IJCAI*. ijcai.org, 2020, pp. 1933–1939.

[17] T. Venetis, G. Stoilos, and G. B. Stamou, “Incremental query rewriting for OWL 2 QL,” in *Proceedings of the 2012 International Workshop on Description Logics*. Citeseer, 2012.

[18] J.-F. Baget, M. Leclère, M.-L. Mugnier, S. Rocher, and C. Sipieter, “Graal: A toolkit for query answering with existential rules,” in *RuleML*, 2015, pp. 328–344.

[19] M. Rodrıguez-Muro, R. Kontchakov, and M. Zakharyaschev, “Query rewriting and optimisation with database dependencies in ontop,” *Proc. of DL*, vol. 2013, 2013.

[20] R. Rosati and A. Almatelli, “Improving query answering over dl-lite ontologies,” in *KR*. AAAI Press, 2010.

[21] M. Thomazo, “Compact rewriting for existential rules,” in *IJCAI: International Joint Conference on Artificial Intelligence*, 2013.

[22] D. Bursztyn, F. Goasdoué, and I. Manolescu, “Optimizing reformulation-based query answering in rdf,” in *EDBT: 18th International Conference on Extending Database Technology*, 2015.

[23] Y. Tian, “The world of graph databases from an industry perspective,” *ACM SIGMOD Record*, vol. 51, no. 4, pp. 60–67, 2023.

[24] B. M. Sasaki, J. Chao, and R. Howard, “Graph databases for beginners,” *Neo4j*, 2018.

[25] S. Harris, A. Seaborne, and E. Prud’hommeaux, “Sparql 1.1 query language,” *W3C recommendation*, vol. 21, no. 10, p. 778, 2013.

[26] G. Fan, W. Fan, Y. Li, P. Lu, C. Tian, and J. Zhou, “Extending graph patterns with conditions,” in *SIGMOD*, pp. 715–729.

[27] Y. Guo, Z. Pan, and J. Heflin, “LUBM: A benchmark for OWL knowledge base systems,” *Journal of Web Semantics*, vol. 3, no. 2-3, pp. 158–182, 2005.

[28] M. Han, H. Kim, G. Gu, K. Park, and W. Han, “Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together,” in *SIGMOD*, 2019, pp. 1429–1446.

[29] S. Sun and Q. Luo, “In-memory subgraph matching: An in-depth study,” in *SIGMOD*, 2020, pp. 1083–1098.

[30] F. Baader, D. Calvanese, D. McGuinness, P. Patel-Schneider, and D. Nardi, *The description logic handbook: Theory, implementation and applications*. Cambridge university press, 2003.

[31] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz *et al.*, “Owl 2 web ontology language profiles,” *W3C recommendation*, vol. 27, no. 61, 2009.

[32] F. Baader, S. Brandt, and C. Lutz, *Pushing the EL envelope*. Technische Universität Dresden, 2005.

[33] T. Eiter, G. Gottlob, M. Ortiz, and M. Šimkus, “Query answering in the description logic horn-shiq,” in *Logics in Artificial Intelligence: 11th European Conference, JELIA 2008, Dresden, Germany, September 28-October 1, 2008. Proceedings 11*. Springer, 2008, pp. 166–179.

[34] I. Horrocks, O. Kutz, and U. Sattler, “The even more irresistible sroiq.” *Kr*, vol. 6, pp. 57–67, 2006.

[35] M.-L. Mugnier and M. Thomazo, “An introduction to ontology-based query answering with existential rules,” *Reasoning Web. Reasoning on the Web in the Big Data Era: 10th International Summer School 2014, Athens, Greece, September 8-13, 2014. Proceedings 10*, pp. 245–278, 2014.

[36] A. Calì, G. Gottlob, and T. Lukasiewicz, “A general datalog-based framework for tractable query answering over ontologies,” in *PODS*, 2009, pp. 77–86.

[37] G. Cima, F. Croce, M. Lenzerini, A. Poggi, and E. Toccacieli, “On queries with inequalities in dl-liter$\neq$,” in *Description Logics*, vol. 2373, 2019.

[38] M. Bienvenu, S. Kikot, R. Kontchakov, V. Ryzhikov, and M. Zakharyaschev, “On the parameterised complexity of tree-shaped ontology-mediated queries in owl 2 ql,” in *DL: Description Logics*, no. 1879, 2017.

[39] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, “Data complexity of query answering in description logics,” *Artificial Intelligence*, vol. 195, pp. 335–360, 2013.

[40] S. Kikot, R. Kontchakov, and M. Zakharyaschev, “On (in) tractability of obda with owl 2 ql.” CEUR Workshop Proceedings, 2011.

[41] Y. Zhou, B. C. Grau, Y. Nenov, M. Kaminski, and I. Horrocks, “Pagoda: Pay-as-you-go ontology query answering using a datalog reasoner,” *J. Artif. Intell. Res.*, vol. 54, pp. 309–367, 2015.

[42] D. Carral, I. Dragoste, L. González, C. Jacobs, M. Krötzsch, and J. Urbani, “Vlog: A rule engine for knowledge graphs,” in *The Semantic Web–ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part II 18*. Springer, 2019, pp. 19–35.

[43] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee, “Rdfox: A highly-scalable rdf store,” in *The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II 14*. Springer, 2015, pp. 3–20.

[44] L. Bellomarini, E. Sallinger, and G. Gottlob, “The vadalog system: Datalog-based reasoning for knowledge graphs,” *Proc. VLDB Endow.*, vol. 11, no. 9, pp. 975–987, 2018.

[45] H. He and A. K. Singh, “Graphs-at-a-time: query language and access methods for graph databases,” in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 405–418.

[46] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, “Efficient subgraph matching by postponing cartesian products,” in *SIGMOD*, 2016, pp. 1199–1214.

[47] W. Han, J. Lee, and J. Lee, “$\text{Turbo}_{\text{ISO}}$: towards ultrafast and robust subgraph isomorphism search in large graph databases,” in *SIGMOD*, 2013, pp. 337–348.

[48] B. Bhattarai, H. Liu, and H. H. Huang, “CECI: compact embedding cluster index for scalable subgraph matching,” in *SIGMOD*, 2019, pp. 1447–1462.

[49] H. Kim, Y. Choi, K. Park, X. Lin, S. Hong, and W. Han, “Versatile equivalences: Speeding up subgraph query processing and subgraph matching,” in *SIGMOD*, 2021, pp. 925–937.

[50] X. Ren and J. Wang, "Multi-query optimization for subgraph isomorphism search," *Proc. VLDB Endow.*, vol. 10, no. 3, pp. 121–132, 2016.

[51] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gStore: Answering SPARQL Queries via Subgraph Matching," *Proc. VLDB Endow.*, vol. 4, no. 8, pp. 482–493, 2011.

[52] J. Kim, H. Shin, W.-S. Han, S. Hong, and H. Chafi, "Taming Subgraph Isomorphism for RDF Query Processing," *Proc. VLDB Endow.*, vol. 8, no. 11, 2015.

[53] X. Wang, Q. Zhang, D. Guo, X. Zhao, and J. Yang, "$\mathsf{GQA}_{\mathsf{RDF}}$: A graph-based approach towards efficient SPARQL query answering," in *DASFAA*, vol. 12113, 2020, pp. 551–568.

[54] W. Le, A. Kementsietsidis, S. Duan, and F. Li, "Scalable multi-query optimization for SPARQL," in *ICDE*, 2012, pp. 666–677.

[55] A. Alhazmi, T. Blount, and G. Konstantinidis, "ForBackBench: a benchmark for chasing vs. query-rewriting," *Proc. VLDB Endow.*, vol. 15, no. 8, pp. 1519–1532, 2022.

[56] D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati, "Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family," *J. Autom. Reason.*, vol. 39, no. 3, pp. 385–429, 2007.

[57] J. Corman and G. Xiao, "Certain answers to a sparql query over a knowledge base," in *Semantic Technology: 9th Joint International Conference, JIST 2019, Hangzhou, China, November 25–27, 2019, Proceedings*. Springer, 2020, pp. 320–335.

[58] A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational data bases," in *STOC*, 1977, pp. 77–90.

[59] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *Acm Sigmod Record*, vol. 22, no. 2, pp. 297–306, 1993.

[60] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient boolean function manipulation," in *DAC*, 1990, pp. 52–57.

[61] "Dbpedia," https://www.dbpedia.org/, 2023.

[62] G. Singh, S. Bhatia, and R. Mutharaju, "Owl2bench: a benchmark for OWL 2 reasoners," in *The Semantic Web–ISWC 2020: 19th International Semantic Web Conference, Athens, Greece, November 2–6, 2020, Proceedings, Part II 19*. Springer, 2020, pp. 81–96.

[63] M. Horridge and S. Bechhofer, "The owl api: A java api for owl ontologies," *Semantic web*, vol. 2, no. 1, pp. 11–21, 2011.

[64] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A. N. Ngomo, "LSQ: the linked SPARQL queries dataset," in *ISWC*, vol. 9367, 2015, pp. 261–269.

[65] X. Jian, Y. Wang, X. Lei, L. Zheng, and L. Chen, "SPARQL rewriting: Towards desired results," in *SIGMOD*, 2020, pp. 1979–1993.

[66] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The dlv system for knowledge representation and reasoning," *ACM Transactions on Computational Logic (TOCL)*, vol. 7, no. 3, pp. 499–562, 2006.

[67] "Ontop v4 beta version," https://github.com/ontop/ontop/releases/tag/ontop-4.0.0-beta-1, 2019.

[68] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, "Hermit: an owl 2 reasoner," *Journal of Automated Reasoning*, vol. 53, pp. 245–269, 2014.

[69] "Stardog-8.2.1," https://downloads.stardog.com/stardog/stardog-latest.zip, 2023.

[70] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical owl-dl reasoner," *Journal of Web Semantics*, vol. 5, no. 2, pp. 51–53, 2007.

[71] S. Sun, X. Sun, B. He, and Q. Luo, "Rapidflow: an efficient approach to continuous subgraph matching," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2415–2427, 2022.

[72] X. Sun, S. Sun, Q. Luo, and B. He, "An in-depth study of continuous subgraph matching," *Proc. VLDB Endow.*, vol. 15, no. 7, pp. 1403–1416, 2022.

[73] H. Wang, Y. Zhang, L. Qin, W. Wang, W. Zhang, and X. Lin, "Reinforcement learning based query vertex ordering model for subgraph matching," in *ICDE*, 2022, pp. 245–258.

[74] "Full version," 2023, https://github.com/dengt2000/OGPs/blob/main/paper.pdf.