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 accelerate ontology-mediated query answering. OGPs employ graph patterns to support topological queries, attach conditions to both vertices and edges to specify additional restrictions, and support conditional partial matching semantics. Hence, OGPs can express conjunctive queries (CQs) under ontological constraints. We develop a PTIME algorithm to generate an equivalent OGP from a CQ over the ontology specified by description logic $DL\text{-}Lite_{\mathcal{R}}$, and design a matching algorithm to match OGPs in graphs. Using real-life and synthetic data, we experimentally verify that the proposed approach outperforms the state-of-the-art algorithms for ontology-mediated query answering by 2–3 orders of magnitude.**

I. Introduction

Ontology-mediated query answering (OMQA) has been extensively studied and used in various domains, *e.g.,* bioinformatics [1], chemistry [2] and e-learning recommendation systems [3]. Given a query Q, a dataset D and an ontology O, OMQA identifies not only the answers of the query Q in D but also the answers *logically implied* by axioms in O. For instance, when Q searches for all faculty members in D, if an axiom in O states that every professor is a faculty member, then all professors are returned, even if such information is not stored in D. The need for OMQA is evident since it can provide more complete query results over incomplete data [4], [5].

Query rewriting is a common technique to answer ontology-mediated queries (OMQs) [6]–[15]: it (1) first rewrites the query into an equivalent query (*e.g.,* a union of conjunctive queries or a Datalog query) that encodes the ontological constraints, and (2) then directly evaluates the equivalent query on a database. However, the generated queries may be exponentially large [16] and are costly to evaluate [5].

Graph databases [17] have been developed, and are often more efficient than relational databases [18]. It is desirable to exploit graph techniques to answer OMQs. However, existing graph query languages, including conventional patterns [17], SPARQL [19], and conditional graph patterns (CGPs) [20], cannot express some OMQ by a single query but a union of multiple similar queries, which results in redundancy.

Example 1: Consider two OMQs adapted from an e-learning recommendation system [3] and a university domain [21], respectively. They are represented by graph patterns in Figure 1.

(1) Query Q_1 of Figure 1 searches for learning resources x, which (i) are categorized as hardware z, (ii) are uploaded in 2023, and (iii) have an abstract w about "GPU study". But if the categorization of x is processor, then the abstract w of x can be anything or even missing (*i.e.,* query Q_2). An ontology on computer science states that processor, memory and I/O devices are all hardware [3]. Then the query returns (i) resources that are uploaded in 2023, are categorized as hardware, memory or I/O devices and have an abstract "GPU study"; and (ii) *all the resources* about processor that are uploaded in 2023. Hence, Q_1 needs to be rewritten into four queries. However, (a) neither CGPs nor SPARQL can encode these four queries as a single query without using union, since CGPs do not support the alternative semantics of z (*e.g.,* z carries different labels in Q_1 and Q_2), and SPARQL cannot express the condition that w can be omitted only when x is processor; and (b) manually writing all SPARQL queries or CGPs to express Q_1 based on the given ontology is error-prone, since the categories may be complex, *e.g.,* memory may contain other sub-categories.

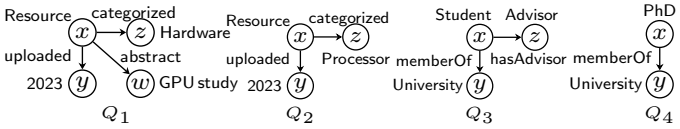


Fig. 1. Graph pattern queries

(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, (a) x carries different labels in Q_3 and Q_4, and (b) z exists in Q_3 but not in Q_4. Both SPARQL queries and CGPs are in the form of a union of two queries to represent Q_3 and Q_4. □

The graph patterns for OMQs overlap and share computation, as shown in Example 1. Can we extend graph patterns to express OMQs? Can we exploit state-of-the-art pattern matching algorithms to accelerate OMQA?

Contributions. To answer these questions, we propose a new class of graph patterns that has a more succinct representation, and design practical rewriting and matching algorithms for it.

(1) Ontological graph patterns (Section III). We propose ontological graph patterns (OGPs). In contrast to conventional patterns, SPARQL and CGPs, OGPs support (a) disjunctions of conditions on both vertices and edges, and (b) conditional partial matching semantics (*i.e.,* vertices can have no match under certain conditions). Due to these two features, we can exploit OGPs to accelerate ontology-mediated query answering.

(2) A rewriting algorithm (Section IV). We show that a conjunctive query (CQ) over ontologies specified by description logic $DL\text{-}Lite_{\mathcal{R}}$ [6] can be expressed as an OGP of polynomial size. However, rewriting such a query into a union of CQs (UCQ) can lead to an exponentially large query [16]. More specifically, we exploit both the disjunctions of conditions and the conditional partial matching semantics of OGPs, and design a PTIME algorithm GenOGP to generate such an OGP, which is equivalent to the exponentially large UCQ [6].

(3) A matching algorithm (Section V). We show that existing matching algorithms for conventional patterns can be adapted with minimum changes to match OGPs. As a case study, we extend a state-of-the-art algorithm DAF [22] due to its effective pruning techniques [23], and design an algorithm OMatch, that given an OGP Q and a graph G, compute the matches $Q(G)$. OMatch also retains the complexity and properties of DAF.

(4) Experimental study (Section VI). Using real-life and synthetic data, we empirically verify 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× faster than the best baseline. (b) Conditions in OGPs do not necessarily slow down the matching. Given an OGP Q generated by GenOGP, 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 OMQA 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 results, while the fastest baseline takes 830s. And OMatch is also feasible on billion-scale graphs.

Related work. We categorize the related work as follows.

Ontology languages. They are designed to represent and reason about knowledge. (1) Description logics [24] are decidable fragments of first-order logic [16], and are the most studied ontology languages [4]. Different description logics vary in their expressivity and complexity. $DL\text{-}Lite_{\mathcal{R}}$ [6] is used to query large datasets and forms the basis of OWL 2 QL [25]; other description logics, *e.g.,* $\mathcal{EL}$ [26], Horn-$\mathcal{SHIQ}$ [27] and $\mathcal{SROIQ}$ [28], allow more constraints, but result in higher complexity [29]. (2) Rule-based ontology languages are also defined, *e.g.,* existential rules [30] and Datalog$^{\pm}$ [31].

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

Ontology-mediated query answering (OMQA). There has been a host of work on OMQA, including (1) theoretical studies [29], [32]–[35] that establish the complexity of OMQA; and (2) designing efficient algorithms to answer queries; *e.g.,* (a) [36]–[39] first adopt a saturation technique to complete the datasets, and then evaluate the given query on the completed datasets; and (b) [6]–[15] rewrite the queries into equivalent queries that can be directly evaluated on datasets.

This work aims to answer CQs over ontologies expressed by $DL\text{-}Lite_{\mathcal{R}}$ via query rewriting. Existing work is summarized as follows. (1) PerfectRef [6] rewrites a CQ into an equivalent UCQ, and there exist various optimizations [7], [10], [11] to reduce the sizes of generated UCQs, but the UCQs are inevitably exponentially large [16]; and (2) more succinct query languages have been explored, including Datalog queries [8], [9], [12], unions of semi-CQs (USCQs) [14] and joins of UCQs (JUCQs) [15]; but they still can be exponentially large [16].

This work differs from the prior work as follows. (1) Instead of rewriting CQs into potentially exponentially large UCQ, Datalog, USCQ or JUCQ, we rewrite a CQ into an OGP of polynomial size. (2) We also design an efficient algorithm for OMQA by extending a state-of-the-art matching algorithm.

Matching algorithms. Graph pattern matching has been extensively studied. Most algorithms [22], [23], [40]–[44] adopt a preprocessing-enumeration framework [23]. For example, CFL-Match [41] and CECI [43] first build auxiliary structures based on a tree representation of graph patterns, then enumerate matches following a matching order of pattern vertices. As opposed to tree representations, DAF [22] and VEQ$_{\mathsf{M}}$ [44] adopt directed acyclic graph (DAG) representations, which have better pruning effect [23]. [45] exploits multi-query optimization strategy to accelerate subgraph isomorphism search.

Pattern matching has also been studied for SPARQL queries [46]–[48]. Specifically, gStore [46] designs indices to accelerate the computation of SPARQL, while [47] and [48] extend existing subgraph isomorphism algorithms [42] and [41] to handle SPARQL, respectively. [49] proposes heuristic techniques to exploit the multi-query optimization for SPARQL.

Closer to our work are CGPs [20], 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 these work as follows. (1) We propose OGPs, which are more complex than CGPs; *e.g.,* support (a) disjunctions of conditions defined on vertices and edges, and (b) conditional partial matching semantics. (2) We extend 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 over description logic $DL\text{-}Lite_{\mathcal{R}}$ [6] into an OGP, which is not studied in [20].

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 targeted at efficient query answering, and underpins the W3C OWL 2 QL ontology [25]. $DL\text{-}Lite_{\mathcal{R}}$ 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.,*

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

where (1) A is an *atomic concept*; (2) P is an *atomic role*; (3) $\exists R$ is an unqualified existential restriction [24] 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 [50], since they cannot provide more query results and are not involved in the query answering. Indeed, they only appear in the negative inclusions of the form $C_1 \sqsubseteq \neg C_2$ (resp. $R_1 \sqsubseteq \neg R_2$) [6], which restrict that the concepts C_1 and C_2 (resp. roles R_1 and R_2) are disjoint. Such negative inclusions are also rare in real-life ontologies; *e.g.,* there exist only 10 negative inclusions among 1703 inclusions in DBpedia [36].

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 $\mathcal{A}$. The semantics of $\mathcal{K}$ is defined in terms of interpretations [6]. Here an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a nonempty domain $\Delta^{\mathcal{I}}$ and an interpretation function $\cdot^{\mathcal{I}}$ that maps (1) each concept C to a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, and (2) each role R to a set of pairs $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. A *model* of $\mathcal{K}$ is an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \mathcal{K}$, *i.e.,* $\mathcal{I}$ satisfies all constraints specified by $\mathcal{T}$ and $\mathcal{A}$ of $\mathcal{K}$.

In this paper, 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 (FOL) 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 '_' [6]. 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 $\mathcal{K}$ [6].

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
$\mathcal{K} = \langle \mathcal{T}, \mathcal{A}\rangle$	a KB with a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$
$G = (V, E, L, F_A)$	a directed labeled graph
$Q[\bar{x}]$=$(V_Q, E_Q, L_Q, \mathcal{C}^l, \mathcal{C}^o)$	an ontological graph pattern
$\mathcal{C}^l/\mathcal{C}^o/\mathcal{U}$	matching/omission/unbound conditions
$gr(g, I)$	an atom generated by applying I on g
$C(u)$	candidate matches of vertex u
$N^u_{u'}(v)$	vertices $v' \in C(u')$ that are adjacent to v
CS/OMCS	index structures to compute matches

Query rewriting. The problem is to check, given a CQ q, a $DL\text{-}Lite_{\mathcal{R}}$ TBox $\mathcal{T}$ and a query language $\mathcal{L}$, whether there exists an *equivalent* query q_o in the query language $\mathcal{L}$ such that the answers of q over any KB $\mathcal{K} = \langle \mathcal{T}, \mathcal{A}\rangle$ are precisely the answers of q_o over $\mathcal{A}$ [6], denoted by $q_o \equiv_{\mathcal{T}} q$. Here, q_o can be a FOL query (*e.g.,* UCQ [6], USCQ [14] or JUCQ [15]) or a Datalog query [8]. Therefore, query answering over $DL\text{-}Lite_{\mathcal{R}}$ can be solved in two steps: (1) generate an equivalent query q_o from q over $\mathcal{T}$, and (2) evaluate q_o in $\mathcal{A}$.

In this paper, we consider rewriting q into an OGP. Then we can 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), (2) next computing the matches of Q in $\mathcal{A}$ (Section V-B), and (3) finally projecting the matches onto distinguished variables of q.

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. Consider *directed labeled graphs* $G = (V, E, L, F_A)$, in which (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. The graph G is a property graph [17] without attributes on edges for clarity.

We assume that each vertex or edge in G carries exactly one label. Our algorithms can be readily extended to handle graphs in which vertices and edges can carry zero or multiple labels.

Patterns. An *ontological graph pattern* (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 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.,* 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).

Here we consider only connected OGPs for simplicity, and our techniques can be readily extended to disconnected OGPs.

CQs *to* OGPs. CQs over KBs can be expressed by OGPs. Given a CQ $q(\bar{x}) = \exists \bar{y}.\varphi(\bar{x}, \bar{y})$, we define an equivalent OGP $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mathcal{C}^l, \mathcal{C}^o)$ as follows: (1) V_Q contains vertices that represent variables in $\bar{x} \cup \bar{y}$ and carry the label wildcard '$*$'; (2) for each atom g in q, if g is $A(x)$, then add $A(x)$ to $C^l(x)$; and if g is $P(x, y)$, then add $P(x, y)$ to $C^l(e)$ for edge $e = (x, y)$; and (3) $C^o(x) = \emptyset$ for each vertex x.

To simplify the presentation, assume that for each $x \in \bar{x} \cup \bar{y}$ there exists *at most one* atom $A(x)$ in q. To encode CQs with multiple atoms, OGPs can use *conjunctions* of conditions.

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

(1) OGP $Q'_1[x, y, z, w]=(V_{Q'_1}, E_{Q'_1}, L_{Q'_1}, \mathcal{C}^l_1, \mathcal{C}^o_1)$ extends pattern Q_1 of Figure 1 as follows: (a) add the label wildcard '$*$' to all vertices x, y, z and w; (b) attach (i) a matching condition $\mathcal{C}^l_1(z) = \mathsf{Hardware}(z) \vee \mathsf{Processor}(z) \vee \mathsf{Memory}(z) \vee \mathsf{I/O\ Devices}(z)$ to z, *i.e.,* z carries label Hardware, Processor, Memory or I/O Devices, and (ii) an omission condition $\mathcal{C}^o_1(w) = \mathsf{Processor}(z)$ to w, since w can have no match when z carries label Processor. Thus, the OGP Q'_1 contains both patterns Q_1 and Q_2, and encodes the ontological constraints, *i.e.,* processor, memory and I/O devices are hardware.

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

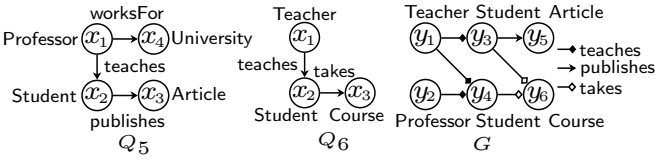


Fig. 2. Graph patterns and graphs

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, if $h(x) = \perp$, then $\mathcal{C}^o(x) \neq \emptyset$ and $h \models \mathcal{C}^o(x)$; here $l \asymp l'$ if $l = l'$ or l is '$*$', *i.e.,* wildcard matches any label; and
- for each edge $e = (x, l, y) \in E_Q$, there exists 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.

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 $Q'_5(G) = \{h_1(\bar{x}), h_2(\bar{x})\}$. □

Remark. OGPs can be expressed in SPARQL, but the equivalent SPARQL query of one OGP can be exponentially large. Indeed, (a) conditions $\mathcal{C}^l(u)$ and $\mathcal{C}^l(e)$ can be encoded using AND, UNION and FILTER operators of SPARQL. But (b)

condition $\mathcal{C}^o(u)$ cannot be directly expressed by OPTIONAL operators of SPARQL, since the vertex u can be omitted only after $\mathcal{C}^o(u)$ is satisfied, while the OPTIONAL operator omits u as long as it does not have a match. To encode $\mathcal{C}^o(u)$, SPARQL needs to take union of all queries representing all possible values of $\mathcal{C}^o(u)$, which can be exponentially large.

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 [6]. In the following, we first review PerfectRef (Section IV-A), and then present GenOGP (Section IV-B).

A. Overview of Algorithm GenOGP

Review. Given a CQ q and a set $\mathcal{T}$ of ontological constraints expressed by $DL\text{-}Lite_{\mathcal{R}}$, 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 new CQs that are generated by *interleaving* two key procedures 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 $DL\text{-}Lite_{\mathcal{R}}$, 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). Atom $gr(g,I)$ can be defined similarly for other inclusions.

Reduction. It removes redundant atoms from the CQs q_i in $\mathcal{S}(q,\mathcal{T})$. More specifically, if q_i contains two atoms g_1 and g_2 that have a *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 by $\mathcal{G}(g_1,g_2)$. Here the *most general unifier* $\mathcal{G}(g_1,g_2)$ is constructed 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 argument 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 the variable y that appears in the same argument. Atoms g_1 and g_2 have a most general unifier, if (a) g_1 and g_2 are instances of the same role, (b) they have a common variable in the same argument, and (c) g_1 or g_2 has an unbound variable in the other argument; *e.g.,* atoms $P(x,\_)$ and $P(x,y)$ above.

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'.

TABLE II
Deductions over $DL\text{-}Lite_{\mathcal{R}}$

Inclusions I	Atoms g	$gr(g,I)$	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: $\exists P_2 \sqsubseteq \exists P_1$ I_5: $\exists P_2^- \sqsubseteq \exists P_1$ I_6: $\exists P_2 \sqsubseteq \exists P_1^-$ I_7: $\exists P_2^- \sqsubseteq \exists P_1^-$	$P_1(x,\_)$ $P_1(x,\_)$ $P_1(\_,x)$ $P_1(\_,x)$	$P_2(x,\_)$ $P_2(\_,x)$ $P_2(x,\_)$ $P_2(\_,x)$	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_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}: $\forall y(g \in (\mathcal{C}^l(x,y) \cap \mathcal{U}(y))$ $\rightarrow (\mathcal{C}^l(x,y) \cup \{gr(g,I)\}) \parallel (\mathcal{C}^o(y) \cup \{gr(g,I)\})$

Note: $\mathcal{X}(\cdot)$ is one of the following sets: $\mathcal{C}^l(\cdot), \mathcal{C}^l(\cdot,\cdot), \mathcal{C}^o(\cdot)$ and $\mathcal{U}(\cdot)$.

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 $\mathcal{A} = \{\mathsf{PhD}(\mathsf{Ann})\}$. □

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 $P_2(x,y_2)$, $P_3(x,y_3)$, …, $P_n(x,y_n)$. In-

deed, 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 first define four sets to record the generated conditions: (a) $\mathcal{C}^l(x)$ and $\mathcal{C}^o(x)$ (resp. $\mathcal{C}^l(e)$) to record matching and omission conditions (resp. matching conditions) for vertex x (resp. edge e), respectively; and (b) $\mathcal{U}(x)$ to store conditions for vertex x to be unbound. Then we can encode the added and removed atoms by matching and omission conditions in OGPs using these sets as follows.

(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 inclusions ensure the existence of some edges, and we can exploit the *omission* conditions to record such information. For example, if inclusion $A \sqsubseteq \exists P$ (*i.e.,* I_{11}) is applied to edge (x, y), then we 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:

$$r\colon\ g \in \mathcal{C} \rightarrow \mathcal{C}' \cup \{gr(g, I)\},$$

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: $\mathcal{C}^l(x, y), \mathcal{C}^o(x)$ and $\mathcal{U}(x)$. Other inclusions can be 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 x 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} can remove an atom from queries based on the inclusions I_{10} and I_{11}. Consider the inclusion I_{10}, if the label of x is A and y is unbound (*i.e.,* $P(x, y) \in (\mathcal{C}^l(x, y) \cap \mathcal{U}(y))$), then we can remove the edge $P(x, y)$ (*i.e.,* $\mathcal{C}^l(x, y) := \mathcal{C}^l(x, y) \cup \{A(x)\}$) from the query, and record that y can be omitted (*i.e.,* $C^o(y) := C^o(y) \cup \{A(x)\}$). Here, the operator $\|$ means that both sets $\mathcal{C}^l(x, y)$ and $\mathcal{C}^o(y)$ are updated.

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 $P_1(x, y_1)$. This is because (a) variables $y_2, \ldots, y_n$ appear only once in q_e, and are unbound; and (b) $P_2(x, y_2), \ldots, P_n(x, y_n)$ can be substituted by $P_1(x, y_1)$. After the reduction, new inclusions may be applied, since variable x becomes unbound in the generated OGPs.

More specifically, we adopt the following strategy: conduct the reduction only when x can become unbound. We verify the conditions for reduction by checking condition sets in OGPs and $\mathcal{U}(\cdot, \cdot)$ (see below). Indeed, when x cannot become unbound after the reduction, if we do not conduct reduction, we can still obtain an equivalent rewriting, since CQs generated via Reduction are equivalent to the CQ before the reduction.

B. The Rewriting Algorithm GenOGP

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

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 set $\mathcal{U}(\cdot)$ based on q;
2 **repeat** */* Iteratively extend condition sets */*
3 $\quad \langle Q, \mathcal{U} \rangle \leftarrow \mathsf{CondDeduction}(\mathcal{T}, Q, \mathcal{U})$;
4 $\quad \langle Q, \mathcal{U} \rangle \leftarrow \mathsf{LazyReduction}(\mathcal{T}, Q, \mathcal{U})$;
5 **until** no more updates to all condition sets;
6 **return** Q;

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 2–5). When no more condition is generated, it returns the constructed OGP Q (line 6).

Procedure CondDeduction. It deduces new conditions to extend 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 OGP Q, and (2) repeatedly applies rules to add new conditions until no more condition can be deduced. This is to facilitate the lazy reduction strategy (see below). Consider rules r_1 and r_2 for inclusion $A_2 \sqsubseteq A_1$. CondDeduction extends the condition sets as follows: (a) for each set $\mathcal{C}^l(x)$, if it contains a condition $A_1(x)$, then CondDeduction extends $\mathcal{C}^l(x)$ with condition $A_2(x)$; and (b) for each set among $\mathcal{C}^l(y,z)$,$\mathcal{C}^o(z)$ and $\mathcal{U}(z)$, if it contains condition $A_1(x)$, then CondDeduction extends it with $A_2(x)$. The vertices y and z are not necessarily the same as x, since the conditions in the generated OGP Q can depend on any vertex in Q (see Section III for the definitions).

Example 9: Consider the CQ q in Example 3, and the TBox $\mathcal{T}$ in Example 2. GenOGP first constructs an initial OGP Q from q, by reproducing the topological structure of q, and then initializes the condition sets in Q based on q (step (1) in Table III; see also Section III). For example, since vertex z in Q is an unbound vertex with an edge $(x, \mathsf{takesCourse}, z)$, the condition $\mathsf{takesCourse}(x,z)$ is added to the set $\mathcal{U}(z)$.

Then CondDeduction extends the condition sets in Q by applying rules in Table II. The results are given in step (2) of Table III. For example, since $\mathcal{T}$ has an inclusion $\mathsf{Student} \sqsubseteq \exists \mathsf{takesCourse}$ and condition $\mathsf{takesCourse}(x,z)$ is in $\mathcal{U}(z)$, rule r_{12} is applied to add $\mathsf{Student}(x)$ to $\mathcal{C}^o(z)$. □

Procedure LazyReduction. It is to reduce redundant edges. To conduct a reduction on 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, and (b) at most one neighbor y of x is not unbound. Then (i) it first identifies the common direction and label of adjacent edges of x by computing the intersection of matching conditions of these adjacent edges. More specifically, assume that $(x, l_1, y_1), \ldots, (x, l_n, y_n)$ are all adjacent edges of x in Q. LazyReduction checks whether there exists a same edge, *i.e.,* same condition of the form $l(x, \_)$ or $l(\_, x)$, in each set $\mathcal{C}^l(x, y_i)$ with $i \in [1, n]$. Note that after CondDeduction, an adjacent edge (x, l, y_i) of x may carry multiple candidate labels or different directions in the matching condition $\mathcal{C}^l(x, y_i)$, so we need to compute the intersection of all these sets $\mathcal{C}^l(x, y_1), \ldots, \mathcal{C}^l(x, y_n)$ to identify all the common direction and label of the adjacent edges of x. And (ii) it next checks whether at most one neighbor y_i is bound, *i.e.,* the computed common condition exists in all sets $\mathcal{U}(y_1), \ldots, \mathcal{U}(y_n)$, except at most one set $\mathcal{U}(y_j)$ $(j \in [1, n])$.

(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.

TABLE III
THE PROCESS OF EXTENDING CONDITION SETS

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

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

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 and y_3, respectively; 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 unbound after the reduction, we further construct the set $\mathcal{U}(y_1)$. The results are given in step (3) of Table III. 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 omission conditions are given in step (4) of Table III. When such OGP is evaluated on ABox $\mathcal{A} = \{\mathsf{PhD(Ann)}\}$, since $\mathsf{PhD}(x)$ exists in the omission condition of y_1 and z, and the existence of y_2 and y_3 are also depended on the matches of y_1, we only evaluate $\mathsf{PhD}(x)$ on the ABox $\mathcal{A}$, and deduce the results Ann. □

Theorem 1: *Given a* CQ q *and a* TBox $\mathcal{T}$*, it is in* $O(|q|^2|\mathcal{T}|^2)$ *time to generate an* OGP Q_o *such that* $Q_o \equiv_{\mathcal{T}} q$. □

Remark. One would expect to find the minimal OGP $Q_{\min}$ such that $Q_{\min} \equiv_{\mathcal{T}} q$. But we can show that finding the minimal OGP is NP-hard by a reduction from the conjunctive query minimization problem [51], which is NP-hard. Intuitively, we can define the ontology $\mathcal{T} = \emptyset$, and show that generating the minimum OGP is the same as finding the minimum CQ.

V. MATCHING ONTOLOGICAL PATTERNS

We next develop an algorithm OMatch to compute the matches $Q(G)$ of a given OGP Q in a graph G. We show that existing matching algorithms for conventional graph patterns

can be extended to match OGPs. As a case study, we design OMatch by extending the state-of-the-art algorithm DAF [22]. Although OGPs adopt homomorphic semantics, rather than isomorphic semantics as in [22], we can still exploit the techniques developed in [22] to accelerate the computation.

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

A. Overview of Algorithm OMatch

Review. DAF 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 conditions. 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 matching 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 matches of Q in G by accessing the structure 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' in $C(u')$ that are adjacent to v in G.

Backtrack. Given the DAG Q_D and the index structure CS, it computes all matches 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 $C(u')$ 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 one of its candidates if all of its parent vertices in Q_D have been mapped. When multiple children are available, it prefers the child u' that has the minimum candidate sets. But if $C(u')$ is empty, it backtracks and maps u_r to another candidate in $C(u_r)$. (4) It treats u' as the new root and 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 the following 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 some 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). 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 need to be verified for all matches of an OGP in G, the number of which can be exponentially large. Here *global conditions* include (a) conditions on edges (u, l, v) that involve vertices other than u and v, and (b) conditions on vertices u that involve another vertex u' with $u \neq u'$.

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** $Q(G)$;

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

(1) For each vertex u with $\mathcal{C}^o(u) \neq \emptyset$ (*i.e.,* u 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 candidates 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 another 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; when verifying a global condition, we check its value using the entries to avoid recomputation; and (b) we also use a shared binary decision diagram (SBDD) [52] to simplify and share the computation of multiple conditions.

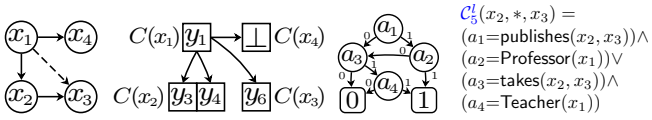
B. The Matching Algorithm OMatch

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,



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

Fig. 3. Demonstration of OMatch

except that it prefers root u_r that does not depend on other vertices, *i.e.,* neither $\mathcal{C}^l(u_r)$ nor $\mathcal{C}^o(u_r)$ involves other vertices.

Procedure BuildOMCS. It builds an auxiliary structure OMCS from OMDAG Q_D. To retain the soundness and equivalence, it (a) revises the pruning strategy in BuildCS of DAF, and (b) 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 $\mathcal{C}^o(u) = \mathsf{true}$, since u can be omitted.

(2) It then stores entries for global conditions in Q to avoid probing G during enumerating the matches. To conduct this, (a) 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 $l(v) = \mathsf{true}$ (resp. $v.A = a$); and (b) 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 $l(v, v') = \mathsf{true}$.

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 $\mathsf{Teacher}(y_1) = \mathsf{true}$, $\mathsf{takes}(y_3, y_6) = \mathsf{true}$ and $\mathsf{takes}(y_4, y_6) = \mathsf{true}$, to accelerate the computation of the global condition $\mathcal{C}_5^l(x_2, *, x_3)$. □

Procedure OMBacktrack. It recursively computes the matches $Q(G)$. Different from procedure Backtrack in DAF, (1) it constructs a SBDD to simplify and represent all global conditions in Q before starting the enumeration; (2) once a vertex u is mapped to a new vertex $v \in C(u)$, it updates the values of the conditions involving vertex u based on the entries in OMCS; and (3) it computes 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. □

Remark. (1) OMatch retains the complexity of DAF [22], and its structure OMCS satisfies the soundness and equivalence (see [53]). (2) For graphs with zero or multi-labeled vertices and edges, we can extend the pruning strategy for candidate sets and OMCS to consider each label of a vertex or edge.

TABLE IV
STATISTICS OF DATASETS AND ONTOLOGIES

Name	$\|D\|$	$\|V\|$	$\|E\|$	$\|O\|$	$\|\Theta_V\|$	$\|\Theta_E\|$	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				
$\mathsf{OWL2Bench}_{100}$	7.3M	1.8M	6.7M	375	136	121	
$\mathsf{OWL2Bench}_{200}$	14.6M	3.4M	13.5M				
$\mathsf{OWL2Bench}_{300}$	22M	5M	20.4M				
$\mathsf{OWL2Bench}_{400}$	29.4M	6.6M	27.2M				

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 [54], a large knowledge base with the dataset and ontology provided by [36], which enriched the original DBpedia ontology with a tourism ontology; (2) NPD [55], a realistic dataset with an ontology about petroleum activities on the Norwegian continental shelf; (3) LUBM [21] and OWL2Bench [56], 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 $\mathsf{OWL2Bench}_{100}$ 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) $|O|$, $|\Theta_V|$ and $|\Theta_E|$ are the numbers of axioms (*i.e.,* the ontology size), distinct concepts and roles in the ontology, respectively.

Note that (1) since these datasets are all based on the RDF data model [57], we transformed them into graphs using the type-aware transformation [47]; and (2) we chose the OWL 2 QL [25] version of LUBM and OWL2Bench, and used the OWL API [58] 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 [22], [23], [41], [43], [44]. 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 generalized 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 may replace the atom $A_1(x)$ (resp. $P_1(x, y)$) in Q with more generalized $A_2(x)$ (resp. $P_2(x, y)$).

Algorithms. We implemented (a) the rewriting algorithm GenOGP (Section IV) and (b) the matching algorithm OMatch (Section V) in C++. We also considered (c) $\text{OMatch}_{\text{BFS}}$, a variant of OMatch that uses a static BFS matching order [43], to demonstrate the effectiveness of OMatch. Note that we stored the data into memory, *i.e.,* our algorithms directly access the data without using any database engine for optimization.

We compared with 8 baselines: (1) Iqaros [10], a UCQ-rewriting algorithm; (2) Rapid+gStore, where we first used Rapid [7] to generate a UCQ-rewriting, then transformed the rewriting into an equivalent SPARQL query with UNION operators [60], and finally exploited the RDF graph database gStore [46] for evaluation; (3) Graal [11], where we took their compilation-based rewriting algorithm Pure_{C} to generate UCQ-rewritings, and chose their memory-based graph store for evaluation; (4) CLIPPER [8], which rewrites a CQ into a Datalog program and evaluates it using the DLV system [61]; (5) Ontop [62], where we adopted their Datalog-rewriting algorithm tree-witness [12], and deployed the relational database PostgreSQL for evaluation; (6) Drewer [9], a Datalog-rewriting system that uses VLog [37] as the evaluation reasoner; (7) PAGOdA [36], a query answering system that combines the Datalog reasoner RDFox [38] and the OWL 2 reasoner HermiT [63]; and (8) Stardog [64], an enterprise knowledge graph with Pellet [65] as the reasoner. All these baselines are implemented in Java, except that some query evaluation parts are written in C/C++ (*i.e.,* 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 integrated with its evaluation part, and cannot be separated [56]. We also made attempts to use the state-of-the-art reasoners VLog and RDFox as saturation-based systems [50], 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 algorithm 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 [23], [44].

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 query size, ontology size and depth on the efficiency of GenOGP and OMatch.

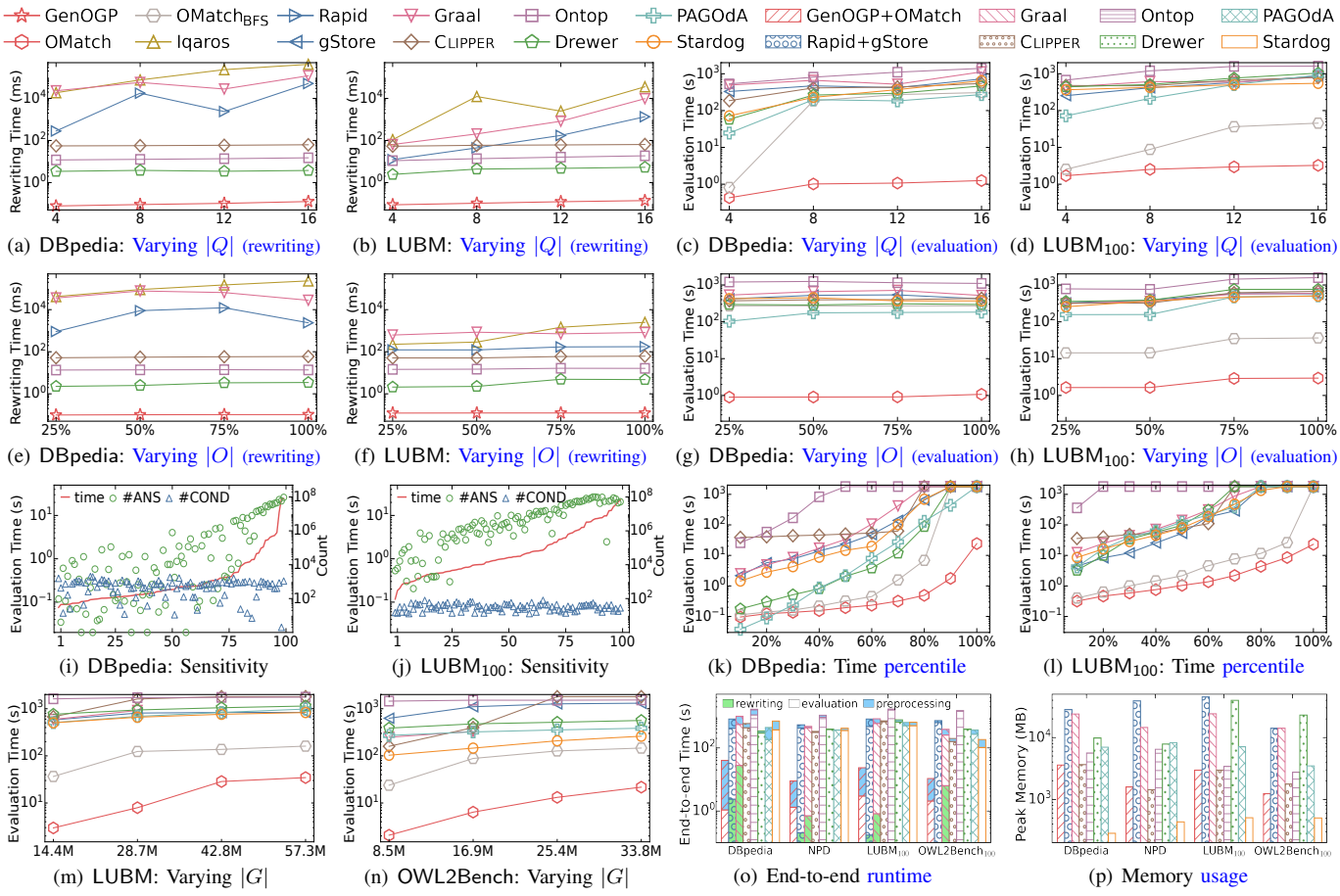
Varying $|Q|$. We varied the size $|Q|$ of the CQ from 4 to 16 to evaluate the efficiency of GenOGP and OMatch.

(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 each UCQ-rewriting algorithm fluctuates 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 of 10 minutes 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 (resp. Stardog) on average takes 272s (resp. 550s) on DBpedia (resp. LUBM_{100}) and fails to solve 10 (resp. 14) 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 $\text{OMatch}_{\text{BFS}}$ by an average of 2 orders of magnitude and 9× on DBpedia and LUBM_{100}, respectively. This shows the impact 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 $|O|$ on the efficiency of GenOGP and OMatch.

(1) Query rewriting. As shown in Figures 4(e)–4(f), (a) on average, GenOGP is 29× 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 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 runtimes 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



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

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

(i) DBpedia: Sensitivity (j) LUBM$_{100}$: Sensitivity (k) DBpedia: Time percentile (l) LUBM$_{100}$: Time percentile

(m) LUBM: Varying $|G|$ (n) OWL2Bench: Varying $|G|$ (o) End-to-end runtime (p) Memory usage

Fig. 4. Performance evaluation

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 shown above, their evaluation times also fluctuate with $|O|$. (b) OMatch and all baselines are more insensitive to $|O|$ on DBpedia than on 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. When the scaling factor is 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$_{BFS}$.

Varying d_O. We also analyzed the impact of the ontology depth d_O on the performance of GenOGP and OMatch. Here the depth d_O of an ontology O is defined as the maximum distance between any two concepts or roles in O [66].

Fixing $|Q| = 12$, we varied the depth of each ontology O from the original depth d_O to d_O+15 by adding more specific concepts and roles to O (see [53]). We find the following. (1) When d_O increases, all algorithms take longer, but GenOGP and OMatch are less sensitive to the increment of d_O. (2) Both GenOGP and OMatch can handle ontologies with large depth; *e.g.*, on DBpedia with $d = 22$, GenOGP (resp. OMatch) on average takes 0.34ms (resp. 1.1s), and beats Drewer (resp. PAGOdA) by 43× (resp. 2 orders of magnitude).

Exp-2: Effectiveness. We tested the effectiveness of GenOGP and OMatch by analyzing (1) rewriting size, (2) sensitivity factors, and (3) cumulative distribution of evaluation time.

Rewriting size. We compared the rewriting sizes of all rewriting algorithms by measuring the number of atoms [9] in the generated UCQ (via Iqaros, Rapid, Graal) and Datalog query (via CLIPPER, Ontop, Drewer). For GenOGP, we counted the size $|Q|$ of the generated OGP Q. 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 size; but (2) Drewer provides a compact Datalog representation to minimize the rewriting size, which is not the target of GenOGP. Actually, the number of conditions in OGPs has a small impact on the query evaluation (see below).

Sensitivity analysis. We relabeled the query IDs in a query set from 1 to 100 in the ascending order of the query evaluation time, as in [67], [68]. For each query, we recorded the number of answers (denoted by #ANS) and the number of conditions in 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. Indeed, conditions, especially omission conditions, often help prune unnecessary search space. Although the enumeration time dominates the computation cost of OMatch, verifying conditions in OGPs only has slight overhead. These findings are similar to CGPs [20].

The worst-case performance. To show the performance of OMatch on hard queires (*i.e.,* queries with large runtime overhead), we computed the cumulative distribution of the query evaluation time [69]. Figures 4(k)–4(l) 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.

Real-life performance. To test GenOGP and OMatch in practical situations, we adopted 14 and 10 benchmark queries provided by LUBM and OWL2Bench, respectively, and randomly selected 10 queries from the LSQ dataset [59], which includes SPARQL queries issued to DBpedia by users. The results are consistent but are not shown due to page limitations (see [53]).

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(m)–4(n), (a) when $|G|$ gets larger, OMatch and all baselines take longer, as expected. But the runtimes of all baselines grow slower than OMatch, since they all have dozens of unsolved queries (*i.e.,* 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 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. (c) OMatch only fails in one query on LUBM when $|G| \geq$ 42.8M due to out of memory error; in contrast, on LUBM_{300} with $|G| = 42.8$M, Stardog, PAGOdA and gStore have 33, 35 and 41 unsolved queries, respectively, and the remaining systems have more than 50 unsolved queries.

We then tested OMatch on a billion-scale graph LUBM_{10000} with 0.33B vertices and 1.1B edges. On the query set $\mathcal{Q}_{12}$, OMatch solves 58 queries, and on average takes 64.4s to answer each query, while Drewer, gStore and Ontop only solve 13, 6 and 2 queries, and on average take 401s, 437s and 836, respectively, to handle each query (see [53] for more results).

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 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 [50].

As shown in Figure 4(o), (a) GenOGP+OMatch beats all baselines in all cases. 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 also 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 the dataset and transform the dataset into 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. It takes 39s, 7.6s, 21s and 8.8s on DBpedia, NPD, LUBM_{100} and OWL2Bench_{100}, respectively; on average it is 1.07× 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 adopts query rewriting techniques to reduce used space. (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 up a lot of 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 beats the fastest baseline Drewer by 29.9×. (2) OMatch is 2–3 orders of magnitude faster than all the baselines; *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) OMatch scales well with large graphs; *e.g.,* on a graph with 13.1M vertices and 44.2M edges, on average it takes 34.6s to answer queries with $|Q| = 12$, while the fastest baseline Stardog takes 830s. (4) The end-to-end runtime of GenOGP+OMatch outperforms the baselines by as least 21.9×.

VII. CONCLUSION

We have proposed ontological graph patterns (OGPs) to accelerate ontology-mediated query answering. OGPs attach conditions to both vertices and edges, and support conditional partial matching semantics. We designed a rewriting algorithm GenOGP to encode a CQ over $DL\text{-}Lite_{\mathcal{R}}$ into an equivalent OGP, and developed a matching algorithm OMatch for OGPs. We also verified that our approach is promising in practice.

One topic for future work is to explore more applications for OGPs, *e.g.,* multi-query optimization. Another topic is to extend OGPs to handle more ontology languages, *e.g.,* $DL\text{-}Lite_{\mathcal{R}}^{\neq}$ [32] or the ontologies supporting property chains [25].

REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

[5] D. Bursztyn, F. Goasdoué, and I. Manolescu, "Teaching an RDBMS about ontological constraints," in *Proc. VLDB Endow*, 2016.

[6] 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, 2007.

[7] A. Chortaras, D. Trivela, and G. Stamou, "Optimized query rewriting for OWL 2 QL," in *CADE*, 2011, pp. 192–206.

[8] T. Eiter, M. Ortiz, M. Simkus, T. Tran, and G. Xiao, "Query rewriting for Horn-SHIQ plus rules," in *AAAI*, 2012.

[9] Z. Wang, P. Xiao, K. Wang, Z. Zhuang, and H. Wan, "Query answering for existential rules via efficient Datalog rewriting," in *IJCAI*, 2020, pp. 1933–1939.

[10] T. Venetis, G. Stoilos, and G. B. Stamou, "Incremental query rewriting for OWL 2 QL," in *Description Logics*, 2012.

[11] 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.

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

[13] R. Rosati and A. Almatelli, "Improving query answering over DL-Lite ontologies," in *KR*, 2010.

[14] M. Thomazo, "Compact rewriting for existential rules," in *IJCAI*, 2013.

[15] D. Bursztyn, F. Goasdoué, and I. Manolescu, "Optimizing reformulation-based query answering in RDF," in *EDBT*, 2015.

[16] M. Bienvenu and M. Ortiz, "Ontology-mediated query answering with data-tractable description logics," *Reasoning Web*, pp. 218–307, 2015.

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

[18] B. M. Sasaki, J. Chao, and R. Howard, "Graph databases for beginners," *Neo4j*, 2018.

[19] S. Harris, A. Seaborne, and E. Prud'hommeaux, "SPARQL 1.1 query language," *W3C recommendation*, vol. 21, no. 10, p. 778, 2013.

[20] G. Fan, W. Fan, Y. Li, P. Lu, C. Tian, and J. Zhou, "Extending graph patterns with conditions," in *SIGMOD*, 2020, pp. 715–729.

[21] 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.

[22] 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.

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

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

[25] 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.

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

[27] T. Eiter, G. Gottlob, M. Ortiz, and M. Šimkus, "Query answering in the description logic Horn-SHIQ," in *JELIA*, 2008, pp. 166–179.

[28] I. Horrocks, O. Kutz, and U. Sattler, "The even more irresistible SROIQ," *KR*, vol. 6, pp. 57–67, 2006.

[29] M. Ortiz and M. Šimkus, "Reasoning and query answering in description logics," *Reasoning Web*, pp. 1–53, 2012.

[30] M.-L. Mugnier and M. Thomazo, "An introduction to ontology-based query answering with existential rules," *Reasoning Web*, pp. 245–278, 2014.

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

[32] 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.

[33] 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.

[34] 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.

[35] S. Kikot, R. Kontchakov, and M. Zakharyaschev, "On (in)tractability of OBDA with OWL 2 QL," in *Description Logics*, 2011.

[36] 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.

[37] D. Carral, I. Dragoste, L. González, C. Jacobs, M. Krötzsch, and J. Urbani, "Vlog: A rule engine for knowledge graphs," in *ISWC*, 2019, pp. 19–35.

[38] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu, and J. Banerjee, "RDFox: A highly-scalable RDF store," in *ISWC*. Springer, 2015, pp. 3–20.

[39] 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.

[40] H. He and A. K. Singh, "Graphs-at-a-time: query language and access methods for graph databases," in *SIGMOD*, 2008, pp. 405–418.

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

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

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

[44] 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.

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

[46] 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.

[47] 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.

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

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

[50] 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.

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

[52] 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.

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

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

[55] 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.

[56] G. Singh, S. Bhatia, and R. Mutharaju, "OWL2Bench: a benchmark for OWL 2 reasoners," in *ISWC*, 2020, pp. 81–96.

[57] F. Manola, E. Miller, B. McBride *et al.*, "RDF primer," *W3C recommendation*, vol. 10, no. 1-107, p. 6, 2004.

[58] M. Horridge and S. Bechhofer, "The OWL API: A Java API for OWL ontologies," *Semantic web*, vol. 2, no. 1, pp. 11–21, 2011.

[59] 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.
[60] J. Corman and G. Xiao, "Certain answers to a SPARQL query over a knowledge base," in *JIST*, 2020, pp. 320–335.
[61] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The DLV system for knowledge representation and reasoning," *TOCL*, vol. 7, no. 3, pp. 499–562, 2006.
[62] "Ontop v4 beta version," https://github.com/ontop/ontop/releases/tag/ontop-4.0.0-beta-1, 2019.
[63] 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.
[64] "Stardog-8.2.1," https://downloads.stardog.com/stardog/stardog-latest.zip, 2023.
[65] 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.
[66] M. Bienvenu, S. Kikot, and V. Podolskii, "Tree-like queries in OWL 2 QL: succinctness and complexity results," in *LICS*, 2015, pp. 317–328.
[67] 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.
[68] 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.
[69] 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.

APPENDIX

A. Proof of Theorem 1

We start with the correctness of GenOGP (*i.e.,* $Q_o \equiv_{\mathcal{T}} q$) and then show GenOGP is in $O(|q|^2|\mathcal{T}|^2)$ time.

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.,* $Q_L^i \sqsubseteq_{\mathcal{T}} Q_o^i \sqsubseteq_{\mathcal{T}} Q_U^i$. Here, $Q_1 \sqsubseteq_{\mathcal{T}} Q_2$ means that given any KB $\mathcal{K}=\langle \mathcal{T}, \mathcal{A} \rangle$, the answers to the query Q_1 over $\mathcal{K}$ is contained within the answers to Q_2 over $\mathcal{K}$. If these hold, since PerfectRef returns the same query regardless of in what order the rules are applied [6], 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 $\mathcal{T}$, and $|q|$ is the size of the CQ q, *i.e.,* the sum of the number of vertices (*i.e.,* atoms like $A(x)$) and edges (*i.e.,* atoms like $P(x, y)$) in q. 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. □

B. Properties of Algorithm OMatch

We show that OMatch retains the same complexity of DAF, and its index OMCS satisfies soundness and equivalence [22].

Time complexity. OMatch runs in $O(|G|^{|Q|})$, the same complexity as DAF, where $|Q|$ is the sum of the number of vertices and edges and the number of conditions like $x.A \oplus c$ in the OGP Q. This is because (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 index OMCS, and (b) $O(|G|)$ time to store the additional entries; and (3) OMBacktrack is in $O(|G|^{|Q|})$ time, since the number of all matches of Q in G is bounded by $\prod_{u \in V_Q} |C(u)| \leq |G|^{|Q|}$.

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 compute $Q(G)$; *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) G is unnecessary after building OMCS (*i.e.,* equivalence). Indeed, (1) OMCS is sound due to the dummy candidate $\perp$ and the revised pruning strategy, and (2) OMCS is equivalent to G, due to the additional entries (see also [22]).

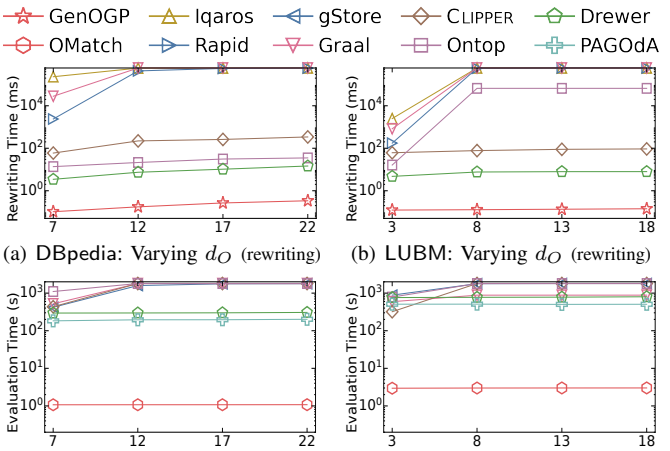
C. Impact of Ontology Depth

Fixing $|Q| = 12$, we analyzed the impact of ontology depth d_O on the performance of GenOGP and OMatch. Here the ontology depth d_O is defined as the maximum distance of two concepts or roles in the ontology [66]; *e.g.,* if there exist k concepts $C_1, \ldots, C_k$ such that $C_1 \sqsubseteq C_2 \sqsubseteq \ldots \sqsubseteq C_k$, then d_O is at least $k-1$. The ontology depths of our used datasets DBpedia, NPD, LUBM and OWL2Bench are 7, 10, 3 and 7, respectively.

To increase d_O, we introduced 5, 10 and 15 more specific concepts and roles for each existing concept and role in the ontology; *e.g.,* for concept C_1, we added 5 axioms (*i.e.,* $C_2 \sqsubseteq C_1, C_3 \sqsubseteq C_2, \ldots, C_6 \sqsubseteq C_5$) to increase d_O by 5.

Figure 5 shows the results on query set $\mathcal{Q}_{12}$ of DBpedia and LUBM_{100}. We find the following. (1) When d_O increases, GenOGP and OMatch take moderately longer, while all the baselines also take longer, they are more sensitive to the increment. (2) Varying d_O from 7 to 22 on DBpedia, the sizes of generated rewriting are increased by 16×, 2468×, 939×, 11×, 16× and 14× for GenOGP, Iqaros, Rapid, CLIPPER, Ontop and Drewer, respectively, while Graal cannot rewrite any query due to the exponential explosion. (3) Since the additional concepts and roles in the ontology are not present in the dataset, although the rewritten queries become much more complex as d_O increases, the runtimes of PAGOdA, Drewer and OMatch increase slowly. (4) Both GenOGP and OMatch can handle ontologies with large depth; *e.g.,* on DBpedia with $d = 22$, GenOGP (resp. OMatch) on average takes 0.34ms (resp. 1.1s), and outperforms the baseline Drewer (resp. PAGOdA) by 43× (resp. 2 orders of magnitude).

Since the ontology depth primarily affects the number of rule applications in the procedure CondDeduction of GenOGP (see Section IV-B), we further considered GenOGP_{depk}, a variant of GenOGP, which limits the depth of applied rules to not exceed k, *i.e.,* given a concept C (resp. property P), GenOGP_{depk} only inspects the concepts (resp. properties) that



(a) DBpedia: Varying d_O (rewriting) (b) LUBM: Varying d_O (rewriting)

(c) DBpedia: Varying d_O (evaluation) (d) LUBM_{100}: Varying d_O (evaluation)

Fig. 5. Impact of ontology depth

are at most k hops from concept C (resp. property P). When k increases from 1 to 5, the running time of GenOGP$_{depk}$ on DBpedia (resp. OWL2Bench$_{100}$) increases from 0.059ms to 0.073ms (resp. 0.06ms to 0.062ms), and the runtime of the corresponding OMatch increases from 1s to 1.1s (resp. 1.2s to 2s), as expected. The deeper depth of applied rules results in more iterations for CondDeduction, and consequently, more conditions in the generated OGP, thus leading to longer runtimes for both GenOGP and OMatch.

D. Number of Unsolved Queries

We set a time limit of 30 minutes for query evaluation, so Figures 4(k)–4(l) also reveal the number of unsolved queries of all methods. As shown there, only our approach GenOGP+OMatch answers all queries without any failure. On DBpedia and LUBM$_{100}$, the best baselines are PAGOdA and Stardog, respectively, with 4 and 12 unsolved queries each.

E. Performance on Real-life Queries

To evaluate 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 [59], which includes SPARQL queries issued to DBpedia by users.

The results for the 14 benchmark queries from LUBM, 10 benchmark queries from OWL2Bench, and 10 real-life queries randomly selected from the LSQ dataset are listed in Table V, Table VI, and Table VII, respectively. As shown in these tables, real-life queries are generally simple. The sizes over 70% of them are between 1 and 3, and the largest one has size 6. So all methods can answer these queries efficiently, and only CLIPPER fails on Q_1 of DBpedia due to memory issues.

Rewriting size. We measured the rewriting sizes of all methods. Note that (1) PAGOdA has no rewriting phase; (2) Rapid and Graal are based on UCQ-rewriting, and we count the number of atoms of each CQ in the rewritten UCQ; (3) CLIPPER, Ontop and Drewer are based on Datalog-rewriting, and we count the number of atoms of each rule in the generated Datalog query; (4) GenOGP generates a single OGP, and we count its size, *i.e.,* the number of vertices and edges plus the number of conditions. Although their rewriting sizes cannot be directly compared, the number of conditions generated by GenOGP is comparable to others. More importantly, conditions are more lightweight than CQs and rules. CLIPPER has a very large rewriting size (greater than 1800) on DBpedia due to the large size of the DBpedia ontology, but the number of conditions in the OGP Q generated by GenOGP is small, since it is bounded by a polynomial of $|Q|$.

Running time. As shown in the tables, GenOGP is consistently the fastest, and it outperforms all the baselines by at least 2 orders of magnitude. OMatch is the fastest in over 60% of the cases, but is slower than gStore or Drewer in other cases. This is because such queries are simple, OMatch must build the auxiliary structure OMCS first, while gStore builds indices in advance, and Drewer directly evaluates the queries. For complex queries or queries with a large number of answers, OMatch is the fastest, *e.g.,* Q_1 and Q_5 of LUBM$_{100}$. In over 75% of the queries, GenOGP+OMatch is the fastest.

F. Scalability on Billion-Scale Graphs

To further test the scalability of OMatch, we generated LUBM$_{10000}$ with 0.33B vertices and 1.1B edges. To run this experiment within a reasonable time, we kept the time limit for query evaluation to 30 minutes. That is, if the query evaluation process runs more than 30 minutes or stops with an error, we recorded 30 minutes as its runtime.

We report the results on the query set $\mathcal{Q}_{12}$. (1) OMatch solves 58 queries, and on average takes 64.4s to answer each query, while the remaining 42 queries all failed due to memory issues. (2) Drewer, gStore and Ontop solve 13, 6 and 2 queries, and on average take 401s, 437s and 836s, respectively, to handle each query. (3) Graal, CLIPPER and PAGOdA fail to answer any query. (4) OMatch is feasible on large graphs; it is the fastest in 13 out of the 14 queries that can be solved by gStore, Drewer and Ontop, and it solves 44 additional queries.

The preprocessing time takes quite a portion of the end-to-end time for large graphs. On average, (1) OMatch takes 42.6 minutes to load the dataset into memory as a graph, (2) Ontop spends 1.8 hours storing the data into database, (3) Drewer requires 1.9 hours to load and saturate the dataset, while (4) gStore takes 7.7 hours to store the data onto SSD.

TABLE V
RESULTS ON LUBM_{100}

Query		Algorithm	Rewriting Size	Rewriting Time (ns)	Evaluation Time (ns)	Total Time (ns)	Answer Size
Q_0	SELECT ?X0 WHERE { ?X0 a GraduateStudent . ?X0 takesCourse GraduateCourse0 .}	GenOGP+OMatch	3	**4.6e+04**	3.1e+07	**3.1e+07**	
		Rapid+gStore	2	6.0e+06	**2.6e+07**	3.2e+07	
		Graal	1	5.9e+07	1.6e+08	2.2e+08	
		CLIPPER	2	5.5e+07	3.3e+10	3.3e+10	4
		Ontop	1	1.7e+7	8.2e+07	9.9e+07	
		Drewer	1	1.0e+06	4.4e+08	4.4e+08	
		PAGOdA	/	/	7.6e+08	7.6e+08	
Q_1	SELECT ?X0 ?X1 ?X2 WHERE { ?X0 a GraduateStudent . ?X1 a University . ?X2 a Department . ?X0 memberOf ?X2 . ?X2 subOrganizationOf ?X1 . ?X0 undergraduateDegreeFrom ?X1 . }	GenOGP+OMatch	15	**5.5e+04**	**1.4e+08**	**1.4e+08**	
		Rapid+gStore	20	9.0e+06	1.0e+09	1.0e+09	
		Graal	4	5.9e+07	4.8e+09	4.9e+09	
		CLIPPER	34	4.3e+07	3.3e+10	3.3e+10	528
		Ontop	13	1.2e+07	1.4e+12	1.4e+12	
		Drewer	6	5.0e+06	5.1e+08	5.2e+08	
		PAGOdA	/	/	1.1e+09	1.1e+09	
Q_2	SELECT ?X0 WHERE { ?X0 a Publication . ?X0 publicationAuthor AssistantProfessor0 .}	GenOGP+OMatch	18	**4.4e+04**	3.1e+07	**3.1e+07**	
		Rapid+gStore	1	7.0e+06	**2.5e+07**	3.2e+07	
		Graal	1	4.8e+07	1.2e+08	1.7e+08	
		CLIPPER	34	4.4e+07	3.2e+10	3.2e+10	6
		Ontop	1	9.0e+06	5.5e+09	5.5e+09	
		Drewer	1	1.0e+06	2.5e+08	2.5e+08	
		PAGOdA	/	/	1.1e+09	1.1e+09	
Q_3	SELECT ?X0 ?X1 ?X2 ?X3 WHERE { ?X0 a Professor . ?X0 worksFor Department0.University0 . ?X0 name ?X1 . ?X0 emailAddress ?X2 . ?X0 telephone ?X3 . }	GenOGP+OMatch	15	**5.7e+04**	1.4e+08	1.4e+08	
		Rapid+gStore	90	2.6e+07	8.2e+09	8.2e+09	
		Graal	18	8.4e+07	1.0e+08	1.8e+08	
		CLIPPER	23	4.2e+07	3.3e+10	3.3e+10	68
		Ontop	32	1.5e+07	6.0e+09	6.0e+09	
		Drewer	10	1.0e+06	**7.5e+07**	**7.6e+07**	
		PAGOdA	/	/	8.8e+08	8.8e+08	
Q_4	SELECT ?X0 WHERE { ?X0 a Person . ?X0 memberOf Department0.University0 . }	GenOGP+OMatch	21	**4.6e+04**	3.1e+07	3.1e+07	
		Rapid+gStore	4	1.9e+07	**1.2e+07**	**3.1e+07**	
		Graal	4	5.6e+07	1.2e+08	1.8e+08	
		CLIPPER	36	4.5e+07	3.0e+10	3.0e+10	719
		Ontop	10	1.4e+07	4.3e+09	4.3e+09	
		Drewer	6	4.0e+06	2.0e+08	2.0e+08	
		PAGOdA	/	/	1.3e+09	1.3e+09	
Q_5	SELECT ?X0 WHERE { ?X0 a Student . }	GenOGP+OMatch	3	**3.4e+04**	**2.2e+08**	**2.2e+08**	
		Rapid+gStore	3	5.0e+06	1.2e+09	1.2e+09	
		Graal	3	6.7e+07	2.3e+09	2.4e+09	
		CLIPPER	5	4.3e+07	3.3e+10	3.3e+10	1737794
		Ontop	7	7.0e+06	2.6e+09	2.6e+09	
		Drewer	3	2.0e+06	1.3e+09	1.3e+09	
		PAGOdA	/	/	2.7e+09	2.7e+09	
Q_6	SELECT ?X0 ?X1 WHERE { ?X0 a Student . ?X1 a Course . ?X0 takesCourse ?X1 . AssociateProfessor0 teacherOf ?X1 . }	GenOGP+OMatch	13	**5.3e+04**	4.0e+07	4.0e+07	
		Rapid+gStore	9	8.0e+06	**1.3e+07**	**1.4e+07**	
		Graal	3	6.5e+07	1.6e+08	2.3e+08	
		CLIPPER	16	4.3e+07	3.6e+10	3.6e+10	61
		Ontop	9	1.3e+07	5.7e+08	5.8e+08	
		Drewer	3	2.0e+06	5.2e+07	5.4e+07	
		PAGOdA	/	/	1.5e+09	1.5e+09	
Q_7	SELECT ?X0 ?X1 ?X2 WHERE { ?X0 a Student . ?X1 a Department . ?X0 memberOf ?X1 . ?X1 subOrganizationOf University0 . ?X0 emailAddress ?X2 . }	GenOGP+OMatch	11	**5.7e+04**	**5.8e+07**	**5.8e+07**	
		Rapid+gStore	60	9.0e+06	6.4e+09	6.5e+09	
		Graal	12	8.4e+07	3.2e+08	4.0e+08	
		CLIPPER	17	4.3e+07	3.8e+10	3.8e+10	12926
		Ontop	19	1.2e+07	1.4e+12	1.4e+12	
		Drewer	8	4.0e+06	2.0e+08	2.0e+08	
		PAGOdA	/	/	1.2e+09	1.2e+09	
Q_8	SELECT ?X0 ?X1 ?X2 WHERE { ?X0 a Student . ?X1 a Faculty . ?X2 a Course . ?X0 advisor ?X1 . ?X1 teacherOf ?X2 . ?X0 takesCourse ?X2 . }	GenOGP+OMatch	26	**5.6e+04**	**3.7e+08**	**3.7e+08**	
		Rapid+gStore	12	1.3e+07	1.8e+09	1.8e+09	
		Graal	3	5.3e+07	8.0e+09	8.1e+09	
		CLIPPER	42	4.5e+07	3.4e+10	3.4e+10	34465
		Ontop	10	1.1e+07	1.3e+08	1.4e+08	
		Drewer	3	2.0e+06	6.9e+08	6.9e+08	
		PAGOdA	/	/	1.7e+09	1.7e+09	
Q_9	SELECT ?X0 WHERE { ?X0 a Student . ?X0 takesCourse GraduateCourse0 . }	GenOGP+OMatch	5	**4.5e+04**	**7.6e+06**	**7.6e+06**	
		Rapid+gStore	6	9.0e+06	1.0e+07	1.9e+07	
		Graal	3	5.5e+07	8.6e+07	1.4e+08	
		CLIPPER	6	4.6e+07	3.1e+10	3.1e+10	0
		Ontop	10	1.4e+07	1.3e+08	1.4e+08	
		Drewer	3	2.0e+06	4.5e+08	4.5e+08	
		PAGOdA	/	/	9.3e+08	9.3e+08	
Q_{10}	SELECT ?X0 WHERE { ?X0 a ResearchGroup . ?X0 subOrganizationOf University0 . }	GenOGP+OMatch	4	**4.4e+04**	7.6e+06	**7.6e+06**	
		Rapid+gStore	4	5.0e+06	**3.3e+06**	8.3e+06	
		Graal	2	6.0e+07	5.0e+07	1.1e+08	
		CLIPPER	4	4.7e+07	3.1e+10	3.1e+10	0
		Ontop	6	1.1e+07	1.9e+08	2.0e+08	
		Drewer	2	1.0e+06	1.6e+07	1.7e+07	
		PAGOdA	/	/	6.7e+08	6.7e+08	
Q_{11}	SELECT ?X0 ?X1 WHERE { ?X0 a Chair . ?X1 a Department . ?X0 worksFor ?X1 . ?X1 subOrganizationOf University0 . }	GenOGP+OMatch	6	**5.0e+04**	8.0e+06	**8.0e+06**	
		Rapid+gStore	8	7.0e+06	**2.2e+06**	9.2e+06	
		Graal	2	6.6e+07	5.7e+07	1.2e+07	
		CLIPPER	6	4.4e+07	3.0e+10	3.0e+10	0
		Ontop	8	1.1e+07	1.9e+08	2.0e+08	
		Drewer	2	2.0e+06	1.6e+07	1.8e+07	
		PAGOdA	/	/	6.7e+08	6.7e+08	
Q_{12}	SELECT ?X0 WHERE { ?X0 a Person . University0 hasAlumnus ?X0 . }	GenOGP+OMatch	26	**4.5e+04**	3.1e+07	3.1e+07	
		Rapid+gStore	5	2.0e+07	**3.3e+06**	**2.3e+07**	
		Graal	5	5.5e+07	6.7e+07	1.2e+08	
		CLIPPER	28	4.5e+07	4.2e+10	4.2e+10	472
		Ontop	11	1.1e+07	4.0e+06	1.5e+07	
		Drewer	6	2.0e+06	1.5e+08	1.5e+08	
		PAGOdA	/	/	1.2e+09	1.2e+09	
Q_{13}	SELECT ?X0 WHERE { ?X0 a UndergraduateStudent . }	GenOGP+OMatch	1	**3.2e+04**	**2.0e+08**	**2.0e+08**	
		Rapid+gStore	1	5.0e+06	8.9e+08	8.9e+08	
		Graal	1	4.8e+07	2.0e+09	2.0e+09	
		CLIPPER	1	4.3e+07	3.1e+10	3.1e+10	1591940
		Ontop	1	6.0e+06	4.8e+11	4.8e+11	
		Drewer	1	1.0e+06	1.2e+09	1.2e+09	
		PAGOdA	/	/	2.4e+09	2.4e+09	

TABLE VI
RESULTS ON OWL2Bench$_{100}$

Query		Algorithm	Rewriting Size	Rewriting Time (ns)	Evaluation Time (ns)	Total Time (ns)	Answer Size
Q_0	SELECT ?X0 ?X1 WHERE { ?X0 knows ?X1 . }	GenOGP+OMatch	1	**4.0e+04**	**2.2e+08**	**2.2e+08**	
		Rapid+gStore	1	5.0e+06	2.9e+08	2.9e+08	
		Graal	1	9.6e+07	1.1e+09	1.1e+09	
		CLIPPER	1	8.7e+07	3.8e+10	3.8e+10	339067
		Ontop	1	1.0e+07	3.6e+10	3.6e+10	
		Drewer	1	2.0e+06	4.9e+08	4.9e+08	
		PAGOdA	/	/	1.4e+09	1.4e+09	
Q_1	SELECT ?X0 ?X1 WHERE { ?X0 hasAlumnus ?X1 . }	GenOGP+OMatch	5	**3.9e+04**	**2.2e+08**	**2.2e+08**	
		Rapid+gStore	5	5.0e+06	9.4e+08	9.4e+08	
		Graal	5	9.6e+07	1.5e+09	1.5e+09	
		CLIPPER	5	6.4e+07	3.9e+10	3.9e+10	476291
		Ontop	11	1.0e+07	1.2e+09	1.2e+09	
		Drewer	6	2.0e+06	1.1e+09	1.1e+09	
		PAGOdA	/	/	1.8e+09	1.8e+09	
Q_2	SELECT ?X0 ?X1 WHERE { ?X0 isAffiliatedOrganizationOf ?X1 . }	GenOGP+OMatch	1	**3.9e+04**	2.5e+07	2.5e+07	
		Rapid+gStore	1	1.6e+07	**2.0e+06**	**1.8e+07**	
		Graal	1	1.2e+08	1.6e+09	1.7e+09	
		CLIPPER	1	6.0e+07	3.7e+10	3.7e+10	1512
		Ontop	1	8.0e+06	2.9e+07	3.7e+07	
		Drewer	1	1.0e+06	2.6e+07	2.7e+07	
		PAGOdA	/	/	1.1e+09	1.1e+09	
Q_3	SELECT ?X0 WHERE { ?X0 hasCollegeDiscipline NonScience . }	GenOGP+OMatch	1	**4.0e+04**	5.3e+06	**5.3e+06**	
		Rapid+gStore	1	1.9e+07	**1.0e+06**	2.0e+07	
		Graal	1	1.2e+08	8.7e+07	2.1e+08	
		CLIPPER	1	6.6e+07	3.7e+10	3.7e+10	0
		Ontop	1	7.0e+06	1.5e+08	1.5e+08	
		Drewer	1	1.0e+06	1.1e+07	1.2e+07	
		PAGOdA	/	/	5.4e+08	5.4e+08	
Q_4	SELECT ?X0 ?X1 WHERE { ?X0 hasCollaborationWith ?X1 . }	GenOGP+OMatch	2	**4.0e+04**	**8.6e+07**	**8.6e+07**	
		Rapid+gStore	2	1.6e+07	3.5e+08	3.7e+08	
		Graal	1	1.1e+08	3.8e+08	4.9e+08	
		CLIPPER	5	6.3e+07	3.7e+10	3.7e+10	124343
		Ontop	1	9.0e+06	2.3e+09	2.3e+09	
		Drewer	2	1.0e+06	2.8e+08	2.8e+08	
		PAGOdA	/	/	9.2e+08	9.2e+08	
Q_5	SELECT ?X0 ?X1 WHERE { ?X0 isAdvisedBy ?X1 . }	GenOGP+OMatch	2	**3.9e+04**	**2.0e+08**	**2.0e+08**	
		Rapid+gStore	2	5.0e+06	6.0e+08	6.1e+08	
		Graal	2	9.3e+07	9.5e+08	1.0e+09	
		CLIPPER	5	6.4e+07	3.8e+10	3.8e+10	517007
		Ontop	1	9.0e+06	6.8e+10	6.8e+10	
		Drewer	3	2.0e+06	7.7e+08	7.7e+08	
		PAGOdA	/	/	1.7e+09	1.7e+09	
Q_6	SELECT ?X0 WHERE { ?X0 a Faculty . }	GenOGP+OMatch	38	**3.6e+04**	**4.7e+07**	**4.7e+07**	
		Rapid+gStore	34	1.0e+07	8.0e+08	8.1e+08	
		Graal	34	1.4e+08	5.9e+09	6.1e+09	
		CLIPPER	81	6.6e+07	3.7e+10	3.7e+10	242716
		Ontop	67	7.0e+06	1.5e+09	1.5e+09	
		Drewer	70	7.0e+06	7.1e+08	7.1e+08	
		PAGOdA	/	/	1.3e+09	1.3e+09	
Q_7	SELECT ?X0 ?X1 WHERE { ?X0 hasSameHomeTownWith ?X1 . }	GenOGP+OMatch	2	**4.0e+04**	**4.3e+08**	**4.3e+08**	
		Rapid+gStore	2	5.0e+06	7.8e+08	7.9e+08	
		Graal	1	1.0e+08	1.8e+09	1.9e+09	
		CLIPPER	5	6.5e+07	3.8e+10	3.8e+10	600138
		Ontop	1	7.0e+06	6.5e+08	6.6e+08	
		Drewer	2	1.0e+06	9.6e+08	9.6e+08	
		PAGOdA	/	/	1.7e+09	1.7e+09	
Q_8	SELECT ?X0 ?X1 WHERE { ?X0 a Student . ?X0 isStudentOf ?X1 . ?X1 isPartOf ?X2 . ?X2 hasCollegeDiscipline Engineering . }	GenOGP+OMatch	31	**5.6e+04**	**7.0e+06**	**7.0e+06**	
		Rapid+gStore	108	3.1e+07	2.7e+08	3.0e+08	
		Graal	36	1.6e+08	2.0e+08	3.6e+08	
		CLIPPER	68	7.2e+07	3.8e+10	3.8e+10	0
		Ontop	13	1.4e+07	2.6e+08	2.8e+08	
		Drewer	27	3.0e+06	1.8e+07	2.1e+07	
		PAGOdA	/	/	1.0e+09	1.0e+09	
Q_9	SELECT ?X0 ?X1 WHERE { ?X0 a Student . ?X2 a Organization . ?X2 hasDean ?X3 . ?X3 teachesCourse ?X1 . ?X4 takesCourse ?X1 . }	GenOGP+OMatch	117	**6.9e+04**	5.9e+08	5.9e+08	
		Rapid+gStore	12	3.3e+07	**2.7e+08**	**3.0e+08**	
		Graal	4	1.2e+08	9.0e+09	9.1e+09	
		CLIPPER	337	7.6e+07	3.8e+10	3.8e+10	14909
		Ontop	11	1.1e+07	3.8e+11	3.8e+11	
		Drewer	6	5.0e+06	4.3e+08	4.4e+08	
		PAGOdA	/	/	1.9e+09	1.9e+09	

TABLE VII
RESULTS ON DBpedia

Query		Algorithm	Rewriting Size	Rewriting Time (ns)	Evaluation Time (ns)	Total Time (ns)	Answer Size
Q_0	SELECT ?X0 ?X1 ?X2 WHERE { ?X0 birthPlace ?X1 . ?X1 country ?X2 . }	GenOGP+OMatch	2	**3.4e+04**	**4.3e+08**	**4.3e+08**	
		Rapid+gStore	2	5.0e+06	1.1e+10	1.1e+10	
		Graal	1	1.5e+08	7.5e+09	7.6e+09	
		CLIPPER	1832	5.7e+07	5.0e+10	5.0e+10	263500
		Ontop	2	8.0e+06	5.0e+09	5.1e+09	
		Drewer	1	2.0e+06	1.0e+09	1.0e+09	
		PAGOdA	/	/	9.2e+08	9.2e+08	
Q_1	SELECT ?X0 ?X1 ?X2 ?X3 WHERE { ?X0 birthPlace ?X1 . ?X1 country ?X2 . ?X2 leaderName ?X3 . }	GenOGP+OMatch	3	**3.4e+04**	**4.9e+08**	**4.9e+08**	
		Rapid+gStore	3	5.0e+06	2.7e+09	2.8e+09	
		Graal	1	1.7e+08	1.3e+10	1.3e+10	
		CLIPPER	1834	5.0e+07	×	×	493611
		Ontop	3	9.0e+06	3.0e+09	3.0e+09	
		Drewer	1	2.0e+06	2.4e+09	2.4e+09	
		PAGOdA	/	/	1.6e+09	1.6e+09	
Q_2	SELECT ?X0 ?X1 WHERE { ?X0 city ?X1 . }	GenOGP+OMatch	1	**3.0e+04**	1.7e+08	1.7e+08	
		Rapid+gStore	1	5.0e+06	**1.4e+08**	**1.5e+08**	
		Graal	1	1.7e+08	7.9e+08	8.0e+08	
		CLIPPER	1830	7.6e+07	5.0e+10	5.0e+10	93584
		Ontop	1	8.0e+06	6.0e+08	6.0e+08	
		Drewer	1	1.0e+06	2.9e+08	2.9e+08	
		PAGOdA	/	/	4.2e+08	4.2e+08	
Q_3	SELECT ?X0 ?X1 ?X2 WHERE { ?X0 spouse ?X1 . ?X1 birthPlace ?X2 . }	GenOGP+OMatch	2	**3.2e+04**	**1.2e+08**	**1.2e+08**	
		Rapid+gStore	2	6.0e+06	6.2e+08	6.2e+08	
		Graal	1	1.6e+08	1.2e+09	1.4e+09	
		CLIPPER	1832	4.9e+07	5.2e+10	5.2e+10	22512
		Ontop	2	1.1e+07	5.5e+08	5.6e+08	
		Drewer	1	2.0e+06	1.7e+08	1.7e+08	
		PAGOdA	/	/	1.7e+08	1.7e+08	
Q_4	SELECT ?X0 WHERE { ?X0 a Airport . }	GenOGP+OMatch	4	**2.7e+04**	**3.4e+07**	**3.4e+07**	
		Rapid+gStore	4	6.0e+06	4.1e+07	4.7e+07	
		Graal	4	1.4e+08	9.0e+08	9.0e+08	
		CLIPPER	1839	5.1e+07	5.0e+10	5.0e+10	12735
		Ontop	9	8.0e+06	3.6e+08	3.7e+08	
		Drewer	5	2.0e+06	7.9e+07	8.1e+07	
		PAGOdA	/	/	1.0e+08	1.0e+08	
Q_5	SELECT ?X0 WHERE { ?X0 a Film . }	GenOGP+OMatch	12	**2.7e+04**	**5.7e+07**	**5.7e+07**	
		Rapid+gStore	12	6.0e+06	4.3e+08	4.3e+08	
		Graal	12	1.7e+08	3.0e+09	3.2e+09	
		CLIPPER	1881	4.9e+07	5.0e+10	5.0e+10	79092
		Ontop	25	8.0e+06	1.1e+09	1.1e+09	
		Drewer	13	2.0e+06	2.2e+08	2.2e+08	
		PAGOdA	/	/	3.2e+08	3.2e+08	
Q_6	SELECT ?X0 ?X1 ?X2 WHERE { ?X0 place ?X1 . ?X1 district ?X2 . }	GenOGP+OMatch	2	**3.3e+04**	7.6e+07	7.6e+07	
		Rapid+gStore	2	5.0e+06	6.6e+08	6.7e+08	
		Graal	1	1.5e+08	2.5e+09	2.7e+09	
		CLIPPER	1832	5.6e+07	5.0e+10	5.0e+10	401
		Ontop	2	1.1e+07	4.0e+08	4.0e+08	
		Drewer	1	2.0e+06	**3.0e+07**	**3.0e+07**	
		PAGOdA	/	/	4.4e+07	4.4e+07	
Q_7	SELECT ?X0 ?X1 WHERE { ?X0 timeZone ?X1 . }	GenOGP+OMatch	1	**2.9e+04**	**1.7e+08**	**1.7e+08**	
		Rapid+gStore	1	5.0e+06	2.7e+08	2.8e+08	
		Graal	1	1.5e+08	8.1e+08	8.3e+08	
		CLIPPER	1830	4.7e+07	5.1e+10	5.1e+10	196811
		Ontop	1	8.0e+06	1.6e+09	1.6e+09	
		Drewer	1	1.0e+06	4.8e+08	4.8e+08	
		PAGOdA	/	/	5.8e+08	5.8e+08	
Q_8	SELECT ?X0 ?X1 WHERE { ?X0 a Book . ?X0 author ?X1 . }	GenOGP+OMatch	9	**3.5e+04**	**9.4e+07**	**9.4e+07**	
		Rapid+gStore	16	5.0e+06	2.2e+10	2.2e+10	
		Graal	8	1.7e+08	7.6e+09	7.8e+09	
		CLIPPER	1852	4.6e+07	5.1e+10	5.1e+10	29766
		Ontop	18	1.0e+07	8.5e+08	8.6e+08	
		Drewer	10	2.0e+06	1.7e+08	1.7e+08	
		PAGOdA	/	/	2.2e+08	2.2e+08	
Q_9	SELECT ?X0 ?X1 WHERE { ?X0 a Road . ?X0 county ?X1 . ?X0 routeStart ?X2 . }	GenOGP+OMatch	12	**4.0e+04**	**8.2e+07**	**8.2e+07**	
		Rapid+gStore	27	8.0e+06	1.5e+08	1.6e+08	
		Graal	9	1.5e+08	1.2e+09	1.4e+09	
		CLIPPER	1867	6.1e+07	5.1e+10	5.1e+10	13924
		Ontop	21	1.3e+07	7.5e+08	7.6e+08	
		Drewer	9	2.0e+6	1.0e+08	1.0e+08	
		PAGOdA	/	/	1.2e+08	1.2e+08	