Schema-Based Query Optimisation for Graph Databases

Chandan Sharma, Pierre Genevès, Nils Gesbert, Nabil Layaïda

Schema-Based Query Optimisation for Graph Databases

Chandan Sharma
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
Grenoble, France
chandan.sharma@inria.fr

Pierre Genevès
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
Grenoble, France
pierre.geneves@inria.fr

Nils Gesbert
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
Grenoble, France
nils.gesbert@inria.fr

Nabil Layaïda
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
Grenoble, France

Abstract

Recursive graph queries are increasingly popular for extracting information from interconnected data found in various domains such as social networks, life sciences, and business analytics. Graph data often come with schema information that describe how nodes and edges are organized. We propose a type inference mechanism that enriches recursive graph queries with relevant structural information contained in a graph schema. We show that this schema information can be useful in order to improve the performance when evaluating recursive graph queries. Furthermore, we prove that the proposed method is sound and complete, ensuring that the semantics of the query is preserved during the schema-enrichment process.

CCS Concepts

• **Information systems → Query languages**.

Keywords

Graph Schema, Query Optimisation, Graph Databases, Relational Algebra

1 Introduction

The creation, utilisation and, most importantly, analysis of highly interconnected data has become pervasive in various domains, including social media, astronomy, chemistry, bio-informatics, transportation networks and semantics associations (criminal investigation) [15, 16, 99, 112]. Graph databases have become appealing for modeling, managing and analysing highly interconnected data [5, 98].

Discovering complex relationships between graph-structured data requires expressive graph query languages to use recursion to navigate paths connecting nodes in a graph database [47, 70]. Therefore, the design of contemporary graph query languages is based on formalisms such as *regular path queries* (RPQ), *two-way regular path queries* (2RPQ) and *nested regular expressions* (NRE) along with their extensions such as *conjunctive two-way regular path queries* (C2RPQ) [37], *union of conjunctive two-way regular path queries* (UC2RPQ) [7, 25], *conjunctive nested regular expressions* (CNRE), *union of conjunctive nested two-way regular path queries* (UCN2RPQ) [14, 78, 91], XPath for graph databases (GXPath) [77, 106] and more recently *conjunctive queries and union of conjunctive queries extended with Tarski's algebra* (CQT/UCQT) [98]. These foundations form the basis of languages such as SPARQL [67], Cypher [54] and PGQL [104]. Furthermore, projects such as ISO/IEC 39075 [1] and Linked Data Benchmark Council [2] (LDBC) are working towards creating a standard query language for graph databases. At the core of all these graph query languages is *recursion*. It is essential to perform complex navigation and data extraction from the graph.

Furthermore, graph data often follow a certain organization, structural patterns, or even sets of constraints on the admissible graph shapes. Such knowledge may be left implicit or made explicit by the means of a so-called *graph schema* specification. In an ongoing standardization effort, *PG-Schema* [8] and *PG-Keys* [9] have been proposed to serve as a reference graph schema language for graph databases.

One fundamental idea of the work presented in this paper is to leverage the schema constraints in order to improve query evaluation. Intuitively, we conjecture that taking advantage of the constraints expressed in a schema can be useful to reduce the amounts of graph data involved when evaluating a query. This research proposes a schema-based query rewriting approach to optimize graph queries. We use a basic yet expressive graph schema formalism (based on [8]) to express graph constraints. We consider graph queries expressed in the formalism of UCQT. We propose a type inference method for injecting relevant schema information into the query and produce a schema-aware UCQT variant that is semantically equivalent.

[1] GQL: https://www.iso.org/standard/76120.html
[2] LDBC: https://ldbcouncil.org

The optimization of recursive queries (even independently from schema constraints) is already known to be significantly more challenging than in the non-recursive setting [70, 88, 111]. Extensions to classical relational algebra have been proposed to support recursion [4, 57, 75, 111]. As a result, many graph database engines use the query optimisation techniques developed for relational databases [71, 96, 101, 109]. Numerous optimisation techniques have been developed [30, 70, 83, 86] to optimise recursive graph queries independently from the schema.

However, due to the schema optional nature of contemporary graph databases [97], earlier schema-based methods that have been researched in other settings for recursive queries have been largely ignored. Examples of such earlier methods include static query analyses for Datalog [20, 36, 38], semi-structured [36, 46, 51] databases and XML [18, 55, 56, 73] databases. While these works are essentially of theoretical nature, they can still bring useful insights for schema-based graph query analyses and optimization.

The main contributions of this research work are:

Schema-based approach for query rewriting. We propose a type inference mechanism capable of leveraging structural information of a graph schema in order to rewrite UCQT queries. In particular, the type inference mechanism enriches the edge label-based navigational graph queries with additional information related to node labels. This approach aims at reducing the size of intermediate subquery results in order to improve the overall query performance. The approach automatically optimises recursive graph queries and is capable of eliminating costly transitive closure operations using schema information. The soundness and completeness of type inference ensure that the semantics of the query is preserved during optimization.

Prototype implementation. We have developed a prototype implementation to demonstrate the practical application of the schema-based approach. The prototype translates schema-based rewritten queries into graph patterns for graph databases and into recursive relational algebra for relational databases. We demonstrate the efficiency of the approach on graph and relational database systems.

Experimental evaluation. In order to empirically evaluate the effectiveness of the approach, we use the LDBC social network benchmark dataset which is a property graph, and the YAGO dataset which is a knowledge graph. We use third-party recursive and non-recursive queries. Experiments show that the proposed schema-based approach is effective, especially for recursive queries.

Organisation: We present preliminary concepts related to graph databases in Sec. 2, including schemas and the UCQT graph query language. The main contribution to schema-based query rewriting is presented in Sec. 3, where we present an inference system to enrich queries with annotations deduced from the schema. The prototype implementation is presented in Sec. 4, where we describe the overall system architecture. We then report on empirical evaluation results in Sec. 5, where we also consider several (relational and graph-based) database systems. Finally, we discuss related works in Sec. 6 before concluding in Sec. 7.

2 Graph Databases

We present a few basic definitions related to graph databases. In particular, we define the notions of graph schema, graph database, schema-database consistency and graph queries. These notions are vital for schema-based query rewriting. Furthermore, we use the graph database representation of the YAGO dataset as a running example to illustrate various definitions present in subsequent sections.

2.1 Graph Schema

A *graph schema* is a directed pseudo multigraph: a graph in which loops on nodes and multiple directed edges between two nodes are permitted. A graph schema captures the structural and properties-based restrictions in a graph database, with nodes representing entities and edges representing relationships between entities. In graph schemas, nodes and edges are labeled, and properties associated with nodes are expressed in the form of *key-type* pairs. Let $\mathsf{L_N}$ be a finite set of node labels and $\mathsf{L_E}$ be a finite set of edge labels such that $\mathsf{L_N} \cap \mathsf{L_E} = \emptyset$. Let $\mathsf{K_S}$ be a set of keys (for example: id, name, age), and T be a finite set of data types (for example: String, Integer, Date). We define a finite set of properties $\mathsf{P_S}$ such that $\mathsf{P_S} \subseteq (\mathsf{K_S} \times \mathsf{T})$.

Definition 1 (Graph Schema). *A graph schema is a tuple* $\mathsf{S} = (\mathsf{N_S}, \mathsf{E_S}, \mathsf{L_N}, \mathsf{L_E}, \mathsf{P_S}, \lambda_\mathsf{S}, \eta_\mathsf{S}, \xi_\mathsf{S}, \Delta_\mathsf{S})$ *where,*

- $\mathsf{N_S}$ *and* $\mathsf{E_S}$ *are finite set of nodes and edges such that* $\mathsf{N_S} \cap \mathsf{E_S} = \emptyset$.
- $\lambda_\mathsf{S} : \mathsf{E_S} \rightarrow \mathsf{N_S} \times \mathsf{N_S}$ *is a function which maps all edges to source and target nodes.*
- $\eta_\mathsf{S} : \mathsf{N_S} \rightarrow \mathsf{L_N}$ *is a function which maps all nodes to the set of node labels.*
- $\xi_\mathsf{S} : \mathsf{E_S} \rightarrow \mathsf{L_E}$ *is a function which maps all edges to the set of edge labels.*
- $\Delta_\mathsf{S} : \mathsf{N_S} \rightarrow 2^{\mathsf{P_S}}$ *is a function which maps all nodes to all possible subsets of the property set* $\mathsf{P_S}$.

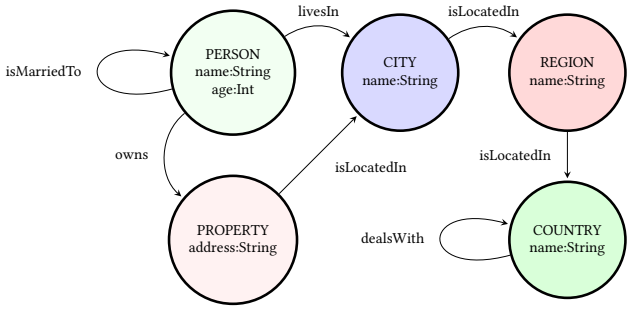

Figure 1: Sample graph schema for YAGO dataset.

Example 1. *Fig. 1 shows a graph schema for the YAGO dataset [66] consisting of five nodes and seven edges. All edges are labeled and directed; some edges can represent loops on the same nodes. For instance, the edge labeled as* isMarriedTo *has the same source and target node labeled as* PERSON. *All nodes are labeled and have properties associated with them. For instance, a node is labeled as* REGION *and has a property* name:String *as a* key:type *pair.*

2.2 Graph Database

A graph database is a graph instance that allows modeling real-world entities as labeled nodes and edges, with properties associated with nodes. Let $\mathsf{K_D}$ be an infinite set of keys (for example, id, name, age), V be a finite set of values (for example, 345, `James`). We define a function $\Upsilon : \mathsf{V} \rightarrow \mathsf{T}$ that maps values in V to their respective data types in T. The set of properties associated with the nodes of a graph database are defined as $\mathsf{P_D}$ such that $\mathsf{P_D} \subseteq (\mathsf{K_D} \times \mathsf{V})$ where each $p_d \in \mathsf{P_D}$ is a *key-value* pair and each value has a data type. Formally, a graph database is defined as follows:

Definition 2 (Graph database). *A graph database is a tuple* $\mathsf{D} = (\mathsf{N_D}, \mathsf{E_D}, \mathsf{L_N}, \mathsf{L_E}, \mathsf{P_D}, \lambda_\mathsf{D}, \eta_\mathsf{D}, \xi_\mathsf{D}, \Delta_\mathsf{D})$ *where,*

- $\mathsf{N_D}$ *and* $\mathsf{E_D}$ *are finite set of nodes and edges such that* $\mathsf{N_D} \cap \mathsf{E_D} = \emptyset$.
- $\lambda_\mathsf{D} : \mathsf{E_D} \rightarrow \mathsf{N_D} \times \mathsf{N_D}$ *is a function which maps all edges to source and target nodes.*
- $\eta_\mathsf{D} : \mathsf{N_D} \rightarrow \mathsf{L_N}$ *is a function which maps all nodes to the set of node labels.*
- $\xi_\mathsf{D} : \mathsf{E_D} \rightarrow \mathsf{L_E}$ *is a function which maps all edges to the set of edge labels.*
- $\Delta_\mathsf{D} : \mathsf{N_D} \rightarrow 2^{\mathsf{P_D}}$ *is a function which maps all nodes to all possible subsets of the property set* $\mathsf{P_D}$.

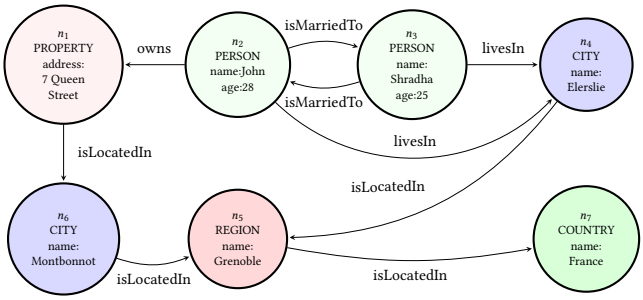

Figure 2: An example of YAGO graph database.

Example 2. *Fig. 2 shows an example of a YAGO graph database consisting of seven nodes and nine edges. All nodes are labeled, and have optional properties associated with them. All edges are labeled and can be identified by unique source and target node identifiers using function* λ_D. *For instance, edge* (n_2, n_1) *is labeled as* `owns`, *with* n_2 *as the source and* n_1 *as the target node identifier. The node with identifier* n_2 *is labeled as* `PERSON` *and has* `name:John` *and* `age:28` *as properties.*

2.3 Schema-database consistency

The notion of schema-database consistency implies that a graph database follows the structural and property based restrictions established by a graph schema. We define a function $\mathcal{SD} : \mathsf{D} \rightarrow \mathsf{S}$ that maps elements in the graph database to at most one element in the graph schema (also known as strict graph schema [8]).

Definition 3 (Schema database consistency). *Given a graph database* D *and a graph schema* S, *we say that* D *is* consistent *with* S *when there exists a schema-database mapping* $\mathcal{SD}$ *such that:*

- *For every node* $n_i \in \mathsf{D}$ *there exists a corresponding node in the schema:* $\mathcal{SD}(n_i) \in \mathsf{S}$, *and we have* $\eta_\mathsf{D}(n_i) = \eta_\mathsf{S}(\mathcal{SD}(n_i))$.
- *For every edge* $e_i \in \mathsf{D}$ *there exists a corresponding edge in the schema:* $\mathcal{SD}(e_i) \in \mathsf{S}$. *Let* $(n_i, n_j) = \lambda_\mathsf{D}(e_i)$, *then we have* $\lambda_\mathsf{S}(\mathcal{SD}(e_i)) = (\mathcal{SD}(n_i), \mathcal{SD}(n_j))$; *furthermore, we have* $\eta_\mathsf{D}(n_i) = \eta_\mathsf{S}(\mathcal{SD}(n_i))$, $\eta_\mathsf{D}(n_j) = \eta_\mathsf{S}(\mathcal{SD}(n_j))$ *and* $\xi_\mathsf{D}(e_i) = \xi_\mathsf{S}(\mathcal{SD}(e_i))$.
- *For each* $n_i \in \mathsf{N_D}$ *and for each* $(k, v) \in \Delta_\mathsf{D}(n_i)$, *we have* $(k, \Upsilon(v)) \in \Delta_\mathsf{S}(\mathcal{SD}(n_i))$.

Example 3. *By using Def. 3, we can observe that the graph database presented in Fig. 2 is consistent with the graph schema shown in Fig. 1. For instance, edge* (n_2, n_3) *is labeled as* `isMarriedTo`; *moreover, source and target nodes are labeled as* `PERSON` *and follow the property-based restrictions imposed in the YAGO graph schema.*

In this study, we only consider graph databases that are consistent with the graph schema; that is, we exclude any database that does not conform to Def. 3. Furthermore, we consider graph databases with restrictions, including each node/edge can have at most one node/edge label associated with it, and edges do not have properties associated with them. However, as mentioned in [106] our graph data model can easily be modified to accommodate properties over edges. In our graph data model we allow zero or more properties associated with nodes. Moreover, we consider that properties are atomic entities and cannot have maps nor lists as data types[3].

2.4 Querying graph databases

Graph patterns are essential in graph query languages as they assist in defining the structure of data to be extracted from a graph database [7, 52, 98]. The core component of a graph pattern is a *path expression* corresponding to specifying directed edges and/or paths defined over the edge labels of a graph database. To specify path expressions we use the formalism of *Tarski's algebra* which is strictly more expressive than other graph query formalisms such as *two-way regular path queries* (2RPQ) and *nested regular expressions* (NRE) [60, 63]. The grammar of Tarski's algebra used to formulate a path expression ϕ is presented in Fig. 3.

ϕ ::=		path expression
	l_e	single edge label ($l_e \in \mathsf{L_E}$)
|	ϕ_1/ϕ_2	concatenation
|	$\phi_1 \cup \phi_2$	union
|	$\phi_1 \cap \phi_2$	conjunction
|	$\phi_1[\phi_2]$	branch (right)
|	$[\phi_1]\phi_2$	branch (left)
|	$-l_e$	reverse
|	ϕ^+	transitive closure

Figure 3: Tarski's algebra grammar (adapted from [60, 98]).

In our adaptation of Tarski's algebra, the *reverse operation* can be used with single-edge labels, as shown in Fig. 3. Notice that it is possible to use the reverse operator in front of general path

[3] Interested readers may refer to [10, 97, 98] for detailed discussions on graph data model restrictions.

expressions ϕ, as this does not offer any additional expressive power compared to the reverse operation on single-edge labels [60].

In the sequel, we define and use the formalism of *conjunctive queries and union of conjunctive queries extended with Tarski's algebra* (CQT/UCQT) to express graph patterns based on path expressions. We choose the UCQT query language formalism for its high expressive power as it encompasses most existing query language formalisms. In particular, UCQT subsumes both *union of conjunctive two-way regular path queries* (UC2RPQ) and *union of conjunctive nested two-way regular path queries* (UCN2RPQ) which are used in many practical graph query languages such as Cypher, SPARQL and PGQL [91, 98, 106].

Additionally, in UCQT, the conjunction operator is closed under transitive closure [60, 98], which adds the ability to express recursive queries that search for arbitrary-length paths with repeating cyclic structures between the start and end nodes. Such queries cannot be expressed in the formalisms of UC2RPQ and UCN2RPQ [98].

2.4.1 Syntax of CQT/UCQT. Given a graph schema S and a graph database D, let $\mathcal{V}_\mathsf{N}$ be a finite set of node variables and $\mathcal{P}_\phi$ be a set of path expressions $\{\phi_1, \ldots, \phi_n\}$, such that each path expression $\phi_i \in \mathcal{P}_\phi$ is defined over the set of edge labels $\mathsf{L_E}$.

Definition 4 (CQT). *A conjunctive query with Tarski's algebra is a logical formula in the $\exists, \wedge$-fragment of first order logic, of the form* $\mathsf{C} = \{(h_1, \ldots, h_i) \mid \exists\ (b_1, \ldots, b_j)\ \ r_1 \wedge \ldots \wedge r_l \wedge\ \ a_1 \wedge \ldots \wedge a_k\ \}$ *where,*

- $\mathsf{H} = \{h_1, \ldots, h_i\}$ *is a finite set of head variables.* $\mathsf{B} = \{b_1, \ldots, b_j\}$ *is a finite set of body variables such that* $(\mathsf{B} \cup \mathsf{H}) \subseteq \mathcal{V}_\mathsf{N}$ *and* $(\mathsf{B} \cap \mathsf{H}) = \emptyset$.
- $\mathsf{A} = \{a_1, \ldots, a_k\}$ *is a finite set of atomic formulas formed by a labeling function* $\eta_\mathsf{A} : \mathcal{V}_\mathsf{N} \rightarrow (\mathsf{L_N} \cup \epsilon)$ *that maps all node variables to node labels in* $\mathsf{L_N}$ *or to the empty label. These formulas can be of the form* e. g. $\eta_\mathsf{A}(Y) =$ PERSON *to specify that nodes represented by the variable* Y *must be labeled as* PERSON.
- $\mathsf{Rel} \subseteq (\mathcal{V}_\mathsf{N} \times \mathcal{P}_\phi \times \mathcal{V}_\mathsf{N})$ *is a finite set of relations where each relation* $r_i \in \mathsf{Rel}$ *either represent directed edge(s) or path(s) connecting two nodes.*

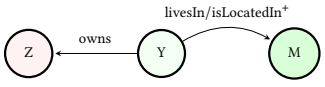

Figure 4: Example of a graph pattern.

Example 4. *The graph pattern in Fig. 4 identifies people living in regions and countries who also own properties. It consists of two relations: (*Y, owns, Z*) searches for people who own properties. While (*Y, livesIn/isLocatedIn+, M*) specifies a path expression to search for paths that have edges labeled as* livesIn *followed by an unbounded number of edges labeled as* isLocatedIn *returning node that either correspond to regions or countries. Both relations share the same node variable* Y*, specifying that both relations must search for the same person.*

$\mathsf{C}_1 = \{\mathtt{Y} \mid \exists(\mathtt{Z},\mathtt{M})\ \ (\mathtt{Y},(\mathtt{livesIn}\ /\mathtt{isLocatedIn+}),\mathtt{M}) \wedge (\mathtt{Y},\mathtt{owns},\mathtt{Z})\}$

Example 5. *Query* C_1*, presents a graph pattern in Fig. 4 expressed as a* CQT *where* Z,M *are body variables and* Y *is a head variable. Relations* (Y,owns,Z) *and* (Y,livesIn /isLocatedIn+,M) *describe the structure of the graph pattern.*

We extend the CQT query language formalism to the formalism of the union of conjunctive queries with Tarski's algebra (UCQT) which is analogous to extending the formalism of C2RPQ to UC2RPQ. The formalism of UCQT represents the disjunction of conjunctive queries with Tarski's algebra. A UCQT query is written as $\mathsf{UC} = \mathsf{H} \leftarrow \{\mathsf{C}_1 \cup \ldots \cup \mathsf{C}_n\}$, where all C_i are *union compatible* CQT queries, that is, they share the same set of head variables H [70, 89].

2.4.2 Semantics of CQT/UCQT. As discussed in Sec. 2.4, path expressions form the core components for syntactically describing graph patterns as CQT/UCQT. We first present the semantics of path expressions ϕ that are based on Tarski's algebra [4]. The interpretation of the path expression ϕ when evaluated over a graph database D (represented as $[\![\phi]\!]_\mathsf{D}$) is defined in Fig. 5. The output produced after evaluating a path expression ϕ consists of all pairs of nodes (source and target nodes) that are connected by the path ϕ in the graph database.

$$
\begin{aligned}
[\![l_e]\!]_\mathsf{D} &= \{(n,m) | n \xrightarrow{l_e} m \in \mathsf{E_D} \wedge n, m \in \mathsf{N_D} \wedge l_e \in \mathsf{L_E}\} \\
[\![\phi_1/\phi_2]\!]_\mathsf{D} &= \{(n,m) | \exists z \in \mathsf{N_D} \wedge (n,z) \in [\![\phi_1]\!]_\mathsf{D} \wedge (z,m) \in [\![\phi_2]\!]_\mathsf{D}\} \\
[\![\phi_1 \cup \phi_2]\!]_\mathsf{D} &= [\![\phi_1]\!]_\mathsf{D} \cup [\![\phi_2]\!]_\mathsf{D} \\
[\![\phi_1 \cap \phi_2]\!]_\mathsf{D} &= [\![\phi_1]\!]_\mathsf{D} \cap [\![\phi_2]\!]_\mathsf{D} \\
[\![\phi_1[\phi_2]]\!]_\mathsf{D} &= \{(n,m) | \exists z \in \mathsf{N_D} \wedge (n,m) \in [\![\phi_1]\!]_\mathsf{D} \wedge (m,z) \in [\![\phi_2]\!]_\mathsf{D}\} \\
[\![[\phi_1]\phi_2]\!]_\mathsf{D} &= \{(n,m) | \exists z \in \mathsf{N_D} \wedge (n,z) \in [\![\phi_1]\!]_\mathsf{D} \wedge (n,m) \in [\![\phi_2]\!]_\mathsf{D}\} \\
[\![-l_e]\!]_\mathsf{D} &= \{(m,n) | (n,m) \in [\![l_e]\!]_\mathsf{D}\} \\
[\![\phi^+]\!]_\mathsf{D} &= \bigcup_{i \geq 1} [\![\phi^i]\!]_\mathsf{D}, \phi^k = (\underbrace{\phi/\ldots/\phi}_{k-\text{times}}) \text{ where } 1 \leq k \leq i.
\end{aligned}
$$

Figure 5: Semantics of Tarski's algebra (adapted from [98]).

Example 6. *Given a graph schema (Fig. 1) and a graph database (Fig. 2) for the YAGO dataset, consider* ϕ_1 = [owns]([isMarriedTo]livesIn) *a path expression used to search for all married property owners living in cities. The path expression* ϕ_1 *first searches for people living in cities using the* livesIn *edge label. After that, the branching operation on* isMarriedTo *edge label only serves as an existential node test to select married people. Finally, the branching operation on the* owns *edge label only selects married people living in cities and owning properties.*

Based on the semantics in Fig. 5, the query returns nodes $\{(n_2, n_4)\}$ *as the final result set, where node* n_2 *represents a person "*John*" who lives in a city named "*Elerslie*" (represented by node* n_4*). Furthermore,* John *is married and owns a property as shown in Fig. 2.*

The formalism of CQT/UCQT expresses queries of two types: *non-recursive graph queries (NQ)* and *recursive graph queries (RQ)*. Non-recursive graph queries are restricted CQT/UCQT that do not allow transitive closure, whereas recursive graph queries allow general path expressions with transitive closure.

[4] Interested readers may refer [98] for detailed discussion on the semantics of CQT/UCQT

The semantics of graph query language formalisms are broadly of two types *(i)* evaluation semantics and *(ii)* output semantics [7, 98]. The formalism of CQT/UCQT uses homomorphism-based evaluation semantics for non-recursive graph queries (NQ), arbitrary path semantics for recursive graph queries (RQ) and set-based output semantics [98].

3 Schema-Based Query Rewriting

We introduce a method that leverages schema information for query evaluation. Specifically the method rewrites a query so that structural schema information is injected in relevant part of the queries, namely in path expressions. As a preliminary step, we first transform path expressions into a form where redundancies are eliminated. We then present how schema information can be injected into path expressions.

Preliminary path simplifications. The purpose of rewrite rules presented in Fig. 6 is to eliminate redundancies from path expressions in order to simplify queries. These rewrite rules are general in that they apply independently from any particular schema.

$$(\phi^+)^+ \to \phi^+ \text{ (R1)} \qquad \phi_1^+[\phi_2^+] \to \phi_1^+[\phi_2] \text{ (R2)} \qquad \phi_1[\phi_2/\phi_3] \to \phi_1[\phi_2[\phi_3]] \text{ (R3)}$$
$$[\phi_2^+]\phi_1^+ \to [\phi_2]\phi_1^+ \text{ (R4)} \qquad [\phi_2/\phi_3]\phi_1 \to [\phi_2[\phi_3]]\phi_1 \text{ (R5)}$$

Figure 6: Rewrite rules for path expression simplification.

Rule R1 removes the redundant use of transitive closures. Rules R2 and R4 simplify transitive closures within the branching operator since computing full transitive closure in branching expressions is not required [60, 63]. Rules R3 and R5 turn path compositions into branching operations when possible, as also done in prior works [60–63, 69]. Correctness of these rules follows immediately from the formal semantics of path expressions, and in particular from the existential semantics of the branching operator defined in Figure 5.

$$\phi_{red} = \quad (((\text{owns}[\text{isMarriedTo}^+/\text{livesIn}/\text{dealsWith}^+])/(\text{isLocatedIn}^+)^+)^+)^+$$
$$\phi_{opt} = \quad ((\text{owns}[\text{isMarriedTo}[\text{livesIn}[\text{dealsWith}]]]/\text{isLocatedIn}^+)^+$$

Figure 7: Application of path simplification rules.

Example 7. *In Fig 7, ϕ_{red} is a path expression with redundant plus operations and uses the concatenation operation within the branched path expressions. Rule* R1 *removes redundant plus operations, rule* R2 *removes plus operation inside branched expressions, and rule* R3 *replaces concatenation with branching operation in branched expressions. The optimised path expression is presented in Fig. 7 as the path expression ϕ_{opt}.*

3.1 Schema-based query analysis

Path expressions in Tarski's algebra do not contain any information about node labels. However, given a graph schema, it is possible to use the structure of a path expression to infer information about node labels. This information can then be used to rewrite the original query into a more precise query. The rewritten query generates fewer intermediate results in general, but is always equivalent to the original one in terms of the final result set, when the queried database conforms strictly to the schema.

Example 8. *In the graph database of the YAGO dataset shown in Fig. 2, suppose we need to search for a path such as* owns/isLocatedIn. *To answer this query, we only need to look for paths that start from a node labeled* PERSON *with an outgoing edge labeled* owns *to a node labeled* PROPERTY, *followed by an outgoing edge labeled* isLocatedIn *to nodes labeled* CITY. *This path traversal information can be inferred from the graph schema to formulate a precise path traversal query. When we do not use this schema information, the query considers all edges labeled* isLocatedIn *that start from nodes labeled* PROPERTY, CITY, *and* REGION, *respectively.*

3.1.1 Annotated path expressions. We first extend the grammar of Tarski's algebra to annotate concatenations with node labels: *annotated path expressions* ψ follow the same grammar as described in Fig. 3 except that concatenation / can be replaced by its annotated version $/_{l_n}$ where l_n is a node label. The expression $\psi_1/_{l_n}\psi_2$ represents paths which follow ψ_1, arrive at a node labeled l_n, and go on from there following ψ_2. Formally, we define:

$$[\![\psi_1/_{l_n}\psi_2]\!]_{\mathsf{D}} = \{(n,m) \mid \exists z \in \mathsf{N}_{\mathsf{D}}\ \eta_{\mathsf{D}}(z) = l_n \wedge (n,z) \in [\![\psi_1]\!]_{\mathsf{D}} \wedge (z,m) \in [\![\psi_2]\!]_{\mathsf{D}}\}$$

where $[\![\cdot]\!]$ is defined as in Fig. 5 for the other cases.

Our idea here is that, when querying a database, replacing a plain path expression with a set of annotated ones can help reduce the size of intermediary results by keeping only the relevant data and thus improve efficiency. To determine how these annotations can be added, we use the schema.

3.1.2 Graph schema triples. Given a graph schema S as in Def. 1, we have that for each edge $e_i \in \mathsf{E}_{\mathsf{S}}$, there exists a pair of source and target nodes $\lambda_{\mathsf{S}}(e_i) = (n_i, n_j)$. For each such edge, we consider the *basic graph schema triple* $t_i = (l_n, l_e, l'_n)$ constituted by the source label, the edge label and the target label, without any information about properties. Formally:

Definition 5. *The set of* basic graph schema triples *of* S, $\mathcal{T}_b(\mathsf{S})$, *is defined as follows:*

$$\mathcal{T}_b(\mathsf{S}) = \{(l_n, l_e, l'_n)) \mid \exists n_i, n_j \in \mathsf{N}_{\mathsf{S}}\ \exists e_i \in \mathsf{E}_{\mathsf{S}}\ \xi_{\mathsf{S}}(e_i) = l_e \wedge \lambda_{\mathsf{S}}(e_i) = (n_i, n_j) \wedge \eta_{\mathsf{S}}(n_i) = l_n \wedge \eta_{\mathsf{S}}(n_j) = l'_n\}$$

Example 9. *The graph schema as shown in Fig. 1 contains seven edges; therefore, the set associated with the schema contains seven basic graph schema triples $\mathcal{T}_b(\mathsf{S}) = \{t_1, \ldots, t_7\}$. For instance, the triple t_1 = (*PERSON, owns, PROPERTY*) has* PERSON *as source node label,* PROPERTY *as target node label and* owns *as an associated edge label. Similarly, the triple t_2 = (*PROPERTY, isLocatedIn, CITY*) has* PROPERTY *as source node label,* CITY *as target node label and* isLocatedIn *as an associated edge label.*

Consider the path expression owns. *The only triple containing this label is t_1. If we query a database conforming to our schema, we are able to know that this path expression will only return results conforming to t_1, in the sense that their source node will be labeled* PERSON *and their target node* PROPERTY.

Basic graph schema triples correspond to path expressions consisting of a single edge label. More generally, we define graph schema triples as follows:

Definition 6. *A* graph schema triple *is a triple (l_n, ψ, l'_n) where l_n and l'_n are node labels and ψ an annotated path expression. For a*

graph schema triple t, we write $\mathtt{sc}(t), \mathtt{eT}(t)$ and $\mathtt{tr}(t)$ respectively the source node label, annotated path expression and target node label.

Given a plain path expression and a graph schema, we can compute a number of graph schema triples which are *compatible* with the path expression. For example, if we consider the path expression owns/isLocatedIn, considering that triples t_1 and t_2 from Exp. 9 share a common node label PROPERTY, we can build the triple (PERSON, $\mathtt{owns}/_{\mathtt{PROPERTY}}\mathtt{isLocatedIn}$, CITY). In that example, it is the only triple compatible with the expression.

3.1.3 Path expression and triple compatibility. The set of triples compatible with a given path expression ϕ according to a schema S can be computed inductively from the sets of triples compatibles with the parts of ϕ. We do this using inference rules and an auxiliary function.

Definition 7 (path expression-triple compatibility). *Let ϕ be a path expression, S a graph schema and t a graph schema triple. The judgement $\vdash_S \phi : t$ means that under schema S, ϕ is* compatible *with t. It is defined inductively by the rules in Fig. 8, where $\mathcal{T}_S(\phi)$ represents the set of all triples compatible with ϕ:*

$$\mathcal{T}_S(\phi) = \{t \mid \vdash_S \phi : t\}$$

The last rule, TPlus, relies on the auxiliary function PIC, which works on the whole set of triples compatible with ϕ at once. This function, defined below, generates two kinds of triples: (a) triples where the path expression is ϕ^+ itself, with all annotations dropped, and (b) triples where the path expression does not contain +, the transitive closure operator, anymore. The rationale is that if the schema information allows us to avoid the costly[5] transitive closure, we prefer to do so. However, we do not want to overcomplicate the expression if we cannot avoid the transitive closure operation.

To determine when transitive closure can be avoided, we associate to the set $\mathcal{T}_S(\phi)$ the *directed graph* whose vertices are the node labels and whose edges are the triples. Any path of length n in that graph yields a triple compatible with ϕ^n (as we can infer by repeated application of TConcat). Since the meaning of ϕ^+ is the union of the ϕ^n for all $n \geqslant 1$, the set of all (non-empty) paths in the graph corresponds to triples compatible with ϕ^+. If this set is finite, we actually define it as *the* set of triples compatible with ϕ^+. If it is infinite, then transitive closure cannot be removed. Since the existence of infinitely many paths is equivalent to the presence of cycles in the graph, we propose the following algorithm for computing PIC:

Definition 8 (Plus Compatibility). $\mathsf{PIC}(\phi, \mathcal{T})$ *is the set of triples resulting of the following:*

(1) Let G be the directed graph associated with $\mathcal{T}$;
(2) Let K be the set of vertices of G which are part of a cycle;
(3) Let R be the initially empty result set;
(4) For each path p from A to B without cycles in G:
- *if any of the vertices of p (including A and B) is in K, then add to R the triple (A, ϕ^+, B)*
- *else*
  - *let ψ be the annotated path expression resulting of concatenating all triples in p;*
  - *add to R the triple (A, ψ, B)*

(5) Return R.

Example 10. *Consider the path expression $\phi_4 = \mathtt{livesIn}/\mathtt{isLocatedIn}^+/\mathtt{dealsWith}^+$, and let S be the schema of Fig. 1. Table 1 shows the sets of triples associated to ϕ_4 and its three sub-terms $\phi_1 = \mathtt{livesIn}(\mathtt{lvIn})$, $\phi_2 = \mathtt{isLocatedIn}^+(\mathtt{isL}^+)$ and $\phi_3 = \mathtt{dealsWith}^+(\mathtt{dw}^+)$.*

For ϕ_3, we have $\mathcal{T}_S(\mathtt{dealsWith}) = \{(\mathtt{COUNTRY}, \mathtt{dealsWith}, \mathtt{COUNTRY})\}$. The graph associated to that singleton set has one vertex and one edge which forms a cycle, therefore the transitive closure cannot be eliminated: we have $\mathcal{T}_S(\mathtt{dealsWith}^+) = (\mathtt{COUNTRY}, \mathtt{dealsWith}^+, \mathtt{COUNTRY})$.

For ϕ_2, the set $\mathcal{T}_S(\mathtt{isLocatedIn})$ contains 3 triples; the associated graph has 4 vertices (PROPERTY, CITY, REGION and COUNTRY) and 3 edges corresponding to the 3 triples. It contains no cycle, therefore $\mathcal{T}_S(\mathtt{isLocatedIn}^+)$ contains 6 triples, corresponding to the 6 non-empty paths in the graph.

Notice that, while TPlus can yield many triples when it is possible to remove the transitive closure, TConcat on the other hand can drastically reduce their number, so that in the end there is only one triple compatible with ϕ_4.

Table 1: Application of the inference rules of Fig. 8 to the term $\phi_4 = \mathtt{livesIn}/\mathtt{isLocatedIn}^+/\mathtt{dealsWith}^+$.

TERM	TRIPLES	RULE
lvIn	(PER, lvIn, CITY)	TBasic
$\mathtt{isL}^+$	(PRO, isL, CITY), (CITY, isL, REG), (REG, isL, CUN), (PRO, $\mathtt{isL}/_{\mathtt{CITY}}\mathtt{isL}$, REG), (PRO, $\mathtt{isL}/_{\mathtt{CITY}}\mathtt{isL}/_{\mathtt{REG}}\mathtt{isL}$, CUN), (CITY, $\mathtt{isL}/_{\mathtt{REG}}\mathtt{isL}$, CUN)	TPlus
$\mathtt{dw}^+$	(CUN, $\mathtt{dw}^+$, CUN)	TPlus
$\mathtt{lvIn}/\mathtt{isL}^+$	(PER, $\mathtt{lvIn}/_{\mathtt{CITY}}\mathtt{isL}$, REG), (PER, $\mathtt{lvIn}/_{\mathtt{CITY}}\mathtt{isL}/_{\mathtt{REG}}\mathtt{isL}$, CUN)	TConcat
$\mathtt{lvIn}/\mathtt{isL}^+/\mathtt{dw}^+$	(PER, $\mathtt{lvIn}/_{\mathtt{CITY}}\mathtt{isL}/_{\mathtt{REG}}\mathtt{isL}/_{\mathtt{CUN}}\mathtt{dw}^+$, CUN)	TConcat
CUN = COUNTRY	isL = isLocatedIn, lvIn = livesIn	REG = REGION
PRO = PROPERTY	dw = dealsWith	PER = PERSON

3.1.4 Properties of the compatibility relation. The path expression-triple compatibility relation is defined relative to a schema. Consider now a database conforming to this schema. The compatibility relation enjoys the following properties relative to any such database:

- Soundness, meaning that whenever our path expression ϕ is compatible with some triple (l_n, ψ, l'_n), then all pairs of a node labeled l_n and a node labeled l'_n linked, in the database, by a path conforming to ψ, are part of the result of ϕ;
- Completeness, meaning that whenever ϕ returns a pair of a node labeled l_n and a node labeled l'_n, there exists a triple (l_n, ψ, l'_n), compatible with ϕ, such that these nodes are linked, in the database, by a path conforming to ψ.

These two properties make the initial path expression semantically equivalent to its set of compatible triples, so long as we only consider databases conforming to the schema. Formally:

Theorem 3.1. *Let S be a graph schema, let D be a graph database conforming to S via the schema-database mapping $\mathcal{SD}$, and let ϕ be a path expression.*

[5] In terms of time and computing resources required for querying graph databases [60, 61].

$$\frac{(l_n, l_e, l'_n) \in \mathcal{T}_b(\mathsf{S})}{\vdash_{\mathsf{S}} l_e : (l_n, l_e, l'_n)}\,(\textsc{TBasic}) \qquad \frac{\vdash_{\mathsf{S}} \phi_1 : (l_n, \psi_1, l'_n) \quad \vdash_{\mathsf{S}} \phi_2 : (l'_n, \psi_2, l''_n)}{\vdash_{\mathsf{S}} \phi_1/\phi_2 : (l_n, \psi_1/_{l'_n}\psi_2, l''_n)}\,(\textsc{TConcat}) \qquad \frac{\vdash_{\mathsf{S}} \phi : (l_n, \psi, l'_n)}{\vdash_{\mathsf{S}} -(\phi) : (l'_n, -(\psi), l_n)}\,(\textsc{TMinus})$$

$$\frac{\vdash_{\mathsf{S}} \phi_2 : t}{\vdash_{\mathsf{S}} \phi_1 \cup \phi_2 : t}\,(\textsc{TUnionR}) \qquad \frac{\vdash_{\mathsf{S}} \phi_1 : (l_n, \psi_1, l'_n) \quad \vdash_{\mathsf{S}} \phi_2 : (l'_n, \psi_2, l''_n)}{\vdash_{\mathsf{S}} \phi_1[\phi_2] : (l_n, \psi_1[\psi_2], l'_n)}\,(\textsc{TBranchR}) \qquad \frac{\vdash_{\mathsf{S}} \phi_1 : (l_n, \psi_1, l'_n) \quad \vdash_{\mathsf{S}} \phi_2 : (l_n, \psi_2, l'_n)}{\vdash_{\mathsf{S}} \phi_1 \cap \phi_2 : (l_n, \psi_1 \cap \psi_2, l'_n)}\,(\textsc{TConj})$$

$$\frac{\vdash_{\mathsf{S}} \phi_1 : t}{\vdash_{\mathsf{S}} \phi_1 \cup \phi_2 : t}\,(\textsc{TUnionL}) \qquad \frac{\vdash_{\mathsf{S}} \phi_1 : (l_n, \psi_1, l'_n) \quad \vdash_{\mathsf{S}} \phi_2 : (l_n, \psi_2, l''_n)}{\vdash_{\mathsf{S}} [\phi_1]\phi_2 : (l_n, [\psi_1]\psi_2, l''_n)}\,(\textsc{TBranchL}) \qquad \frac{t \in \mathsf{PlC}(\phi, \mathcal{T}_{\mathsf{S}}(\phi))}{\vdash_{\mathsf{S}} \phi^+ : t}\,(\textsc{TPlus})$$

Figure 8: Inference rules for the path expression-graph schema triple compatibility relation.

Soundness. Assume $\vdash_{\mathsf{S}} \phi : (l_n, \psi, l'_n)$ *holds for some triple* (l_n, ψ, l'_n). *Let* $(s,t) \in [\![\psi]\!]_{\mathsf{D}}$ *such that* $\eta_{\mathsf{D}}(s) = l_n$ *and* $\eta_{\mathsf{D}}(t) = l'_n$. *Then* $(s,t) \in [\![\phi]\!]_{\mathsf{D}}$.

Completeness. Let $(s,t) \in [\![\phi]\!]_{\mathsf{D}}$. *Then there exists* ψ *such that* $(s,t) \in [\![\psi]\!]_{\mathsf{D}}$ *and* $\vdash_{\mathsf{S}} \phi : (\eta_{\mathsf{D}}(s), \psi, \eta_{\mathsf{D}}(t))$.

Proof. Both properties are proved by structural induction on ϕ. The base cases, where ϕ is a single edge label, are a direct consequence of the consistency between D and S (def. 3) and of the definition of basic graph schema triples (def. 5). The inductive cases other than transitive closure are straightforward since the inference rules of Fig. 8 follow exactly the structure of the semantics defined in Fig. 5. Finally, we detail the case of transitive closure:

Soundness: assume the property holds for ϕ. Let $(l_n, \psi, l'_n) \in \mathsf{PlC}(\phi, \mathcal{T}_{\mathsf{S}}(\phi))$ and let $(s,t) \in [\![\psi]\!]_{\mathsf{D}}$ such that $\eta_{\mathsf{D}}(s) = l_n$ and $\eta_{\mathsf{D}}(t) = l'_n$. We have two cases: if $\psi = \phi^+$ then the result is immediate. Otherwise, according to Def. 8, it means that there exists a sequence of triples $t_1, \ldots, t_n \in \mathcal{T}_{\mathsf{S}}(\phi)$ such that: $\mathsf{sc}(t_1) = l_n$, $\mathsf{tr}(t_i) = \mathsf{sc}(t_{i+1})$ for all $i < n$, $\mathsf{tr}(t_n) = l'_n$, and ψ is the concatenation of $t_1 \ldots t_n$ (i. e. $\psi = \mathsf{eT}(t_1)/_{\mathsf{tr}(t_1)}\mathsf{eT}(t_2)/_{\mathsf{tr}(t_2)}\cdots/_{\mathsf{tr}(t_{n-1})}\mathsf{eT}(t_n)$). Since (s,t) is in $[\![\psi]\!]_{\mathsf{D}}$, it means that there is a path in D from s to t which conforms to ψ. Since ψ is the concatenation of $t_1 \ldots t_n$, that path has to be the concatenation of n sub-paths conforming respectively to the triples $t_1 \ldots t_n$. Since each of those triples is compatible with ϕ, we can use the induction hypothesis for each of them and see that the source-target pair for each one is in $[\![\phi]\!]_{\mathsf{D}}$. It results that (s,t) is in $[\![\phi^n]\!]_{\mathsf{D}}$, and therefore in $[\![\phi^+]\!]_{\mathsf{D}}$.

Completeness: assume the property holds for ϕ. Let $(s,t) \in [\![\phi^+]\!]_{\mathsf{D}}$. By definition, there exists $k \geqslant 1$ such that $(s,t) \in [\![\phi^k]\!]_{\mathsf{D}}$. This means that there is a sequence $n_0 \ldots n_k$ of nodes in D with $n_0 = s$ and $n_k = t$ such that we have $(n_{i-1}, n_i) \in [\![\phi]\!]_{\mathsf{D}}$ for all i between 1 and k. By induction hypothesis, for each i there is a triple $t_i = (\eta_{\mathsf{D}}(n_{i-1}), \psi_i, \eta_{\mathsf{D}}(n_i))$ such that $\vdash_{\mathsf{S}} \phi : t_i$ and $(n_{i-1}, n_i) \in [\![\psi_i]\!]_{\mathsf{D}}$. Let G and K be the graph and set of vertices defined in Def. 8. The sequence $t_1 \ldots t_k$ forms a path from $\eta_{\mathsf{D}}(s)$ to $\eta_{\mathsf{D}}(t)$ in G. If this path contains no cycle, then we see from step (4) of Def. 8 that $\mathsf{PlC}(\phi, \mathcal{T}_{\mathsf{S}}(\phi))$ contains a triple $(\eta_{\mathsf{D}}(s), \psi, \eta_{\mathsf{D}}(t))$ with ψ equal either to ϕ^+ or to the concatenation of $\psi_1 \ldots \psi_k$. In both cases, we have $(s,t) \in [\![\psi]\!]_{\mathsf{D}}$. If the path $t_1 \ldots t_k$ does contain a cycle, say $\eta_{\mathsf{D}}(n_i) = \eta_{\mathsf{D}}(n_j)$ with $j > i$, then the sequence $t_1 \ldots t_i, t_{j+1} \ldots t_k$ is also a path in G without that cycle, and we have $\eta_{\mathsf{D}}(n_j) \in K$ since $t_{i+1} \ldots t_j$ is a cycle in G. If that path still contains a cycle, we can reiterate this 'shortcutting' until there are none left, and obtain a path from $\eta_{\mathsf{D}}(s)$ to $\eta_{\mathsf{D}}(t)$ without cycles and containing at least one vertex in K. Therefore, $\mathsf{PlC}(\phi, \mathcal{T}_{\mathsf{S}}(\phi))$ must contain the triple $(\eta_{\mathsf{D}}(s), \phi^+, \eta_{\mathsf{D}}(t))$, which allows us to conclude since we already have $(s,t) \in [\![\phi^+]\!]_{\mathsf{D}}$. □

3.2 From triples to rewritten query

3.2.1 Merging triples. The compatibility relation allows us to obtain, from a path expression ϕ and a schema S, a set $\mathcal{T}_{\mathsf{S}}(\phi)$ of triples which represent all the paths, annotated with node labels, which can possibly contribute to the query result. In principle, we could then convert each individual triple from $\mathcal{T}_{\mathsf{S}}(\phi)$ into a (very specific) query of its own, combine all those queries with a union and obtain a query which, provided the database conforms to the schema, is equivalent to the initial path expression, while avoiding the computation of intermediary results we know from the schema are useless. However, when several triples have the same underlying path expression and differ only by the node labels, it would not be efficient to compute their results separately and do the union afterwards. So, we merge such triples by replacing all single node labels with sets of possible node labels.

Definition 9. *Let* T *be a set of graph schema triples with the same underlying path expression* ϕ. *The* merged triple *of* T *is the triple* $M(T) = (L_1, \Psi, L_2)$ *where* $L_1 = \{l \mid (l, \_, \_) \in T\}$, $L_2 = \{l \mid (\_, \_, l) \in T\}$, *and* Ψ *is the path expression* ϕ *where each concatenation step is annotated with the set of all labels annotating the same concatenation step in* $\{\psi \mid (\_, \psi, \_) \in T\}$.

Example 11. *Consider an initial path expression* $\phi_1 = a^+/b/d$. *Suppose the set of triples* $\mathcal{T}_{\mathsf{S}}(\phi)$ *contains the following:* $(m, a^+/_n b/_l d, p)$ *and* $(m, a^+/_q b/_r d, l)$. *These triples can be merged into* $(\{m\}, a^+/_{\{n,q\}} b/_{\{l,r\}} d, \{p, l\})$.

From the set $\mathcal{T}_{\mathsf{S}}(\phi)$, we compute the set of merged triples of ϕ, $\mathcal{M}_{\mathsf{S}}(\phi)$, by: 1. partitioning $\mathcal{T}_{\mathsf{S}}(\phi)$ in subsets having the same underlying path expression, and 2. taking the merged triple of each subset. Note that in many cases, this will result in one single merged triple, whose underlying path expression is ϕ itself, but it is not the case whenever ϕ contains a union, or a transitive closure which could be removed.

3.2.2 Removing redundant annotations. In some cases, the source and target node labels assigned to a specific edge label in an annotated path expression are the only source and target node labels associated with that same edge label in the graph schema. In such a case, the annotation would not be useful for minimizing intermediate result size since the whole dataset already conforms to the annotation: on the contrary, it would introduce a useless filtering step. Therefore, as a last step before constructing the rewritten query, we detect and remove such annotations.

Example 12. *Let us assume that in a graph schema, source and target nodes associated with edges labeled as* d *and* b *are* l *and* r *respectively. The node labels* l *and* r *can be eliminated from the annotated path expression in Ex. 11, resulting in* $(\{m\}, a^+/_{\{n,q\}} b/d, \{p, l\})$.

$Q(\alpha, \beta, \phi) = (\emptyset, \emptyset, \{(\alpha, \phi, \beta)\})$
$Q(\alpha, \beta, \psi_1 /_L \psi_2)$ = let γ be a fresh variable,
  let $(B_1, A_1, Rel_1) = Q(\alpha, \gamma, \psi_1)$ and $(B_2, A_2, Rel_2) = Q(\gamma, \beta, \psi_2)$
  in $(B_1 \cup B_2 \cup \{\gamma\}, A_1 \cup A_2 \cup \{\eta_A(\gamma) \in L\}, Rel_1 \cup Rel_2)$
$Q(\alpha, \beta, \psi_1[\psi_2])$ = let γ be a fresh variable,
  let $(B_1, A_1, Rel_1) = Q(\alpha, \beta, \psi_1)$ and $(B_2, A_2, Rel_2) = Q(\beta, \gamma, \psi_2)$
  in $(B_1 \cup B_2 \cup \{\gamma\}, A_1 \cup A_2, Rel_1 \cup Rel_2)$
$Q(\alpha, \beta, [\psi_1]\psi_2)$ = let γ be a fresh variable,
  let $(B_1, A_1, Rel_1) = Q(\alpha, \gamma, \psi_1)$ and $(B_2, A_2, Rel_2) = Q(\alpha, \beta, \psi_2)$
  in $(B_1 \cup B_2 \cup \{\gamma\}, A_1 \cup A_2, Rel_1 \cup Rel_2)$
$Q(\alpha, \beta, \psi_1 \cap \psi_2)$ =
  let $(B_1, A_1, Rel_1) = Q(\alpha, \beta, \psi_1)$ and $(B_2, A_2, Rel_2) = Q(\alpha, \beta, \psi_2)$
  in $(B_1 \cup B_2, A_1 \cup A_2, Rel_1 \cup Rel_2)$

Figure 9: Translation of annotated path expressions into CQTs.

Note that, after merging triples and removing redundant annotations, some queries may revert to the initial path expression. This means that the schema did not contain information which would allow optimizing the query.

3.2.3 Rewritten queries. We now want to translate merged triples into CQT queries. For this, we first observe that the annotated path expressions ψ generated by the rules of Fig. 8 and Def. 8 always obey some syntactic restrictions, namely: no annotations appear under the transitive closure operator, and the union operator never appears, except possibly under transitive closure. Furthermore, the reverse operation only occurs in front of edge labels. These observations allow us to reduce the number of cases in the translation: ψ is either a plain path expression, a concatenation, a branching or a conjunction.

Definition 10 (CQT of a merged triple). *The translation is defined using a function $Q(\alpha, \beta, \psi)$, where α and β are CQT variables and ψ an annotated path expression, which returns a triple (B, A, Rel) corresponding to a CQT representing ψ with $H = \{\alpha, \beta\}$ (see Def. 4). This function is defined in Fig. 9.*

For a given merged triple $t = (L, \psi, L')$, we then define the associated CQT as $C(t) = (\{\alpha, \beta\}, B, A \cup \{\eta_A(\alpha) \in L, \eta_A(\beta) \in L'\}, Rel)$ where $(B, A, Rel) = Q(\alpha, \beta, \psi)$.

This allows us to associate to any path expression, given a schema S, a UCQTrepresenting the query enriched with schema information:

Definition 11 (schema-enriched query). *Given a path expression ϕ and a schema S, the schema-enriched query of ϕ, $\mathcal{R}_S(\phi)$, is the following UCQT: $\mathcal{R}_S(\phi) = \{\alpha, \beta\} \leftarrow \{\bigcup_{t \in \mathcal{M}_S \phi} C(t)\}$*

Thanks to Theorem 3.1, we have that ϕ is equivalent to $\mathcal{R}_S(\phi)$ on any database conforming strictly to S.

Example 13. *Consider the path expression $\phi_4 = \texttt{lvIn/isL}^+/\texttt{dw}^+$ of Example 10. We saw that our inference system derives exactly one triple compatible with it in schema S: $(\texttt{PER}, \texttt{lvIn}/_{\texttt{CITY}}\texttt{isL}/_{\texttt{REG}}\texttt{isL}/_{\texttt{CUN}}\texttt{dw}^+, \texttt{CUN})$. Then, since there is only one triple, there is nothing to merge. We then remove redundant labels: the schema S of Fig. 1 only allows livesIn from PERSON to CITY and dealsWith from COUNTRY to COUNTRY, so the final unique merged triple is: $(\emptyset, \texttt{lvIn/isL}/_{\{\texttt{REG}\}}\texttt{isL/dw}^+, \emptyset)$. Therefore, we can rewrite this expression into:*

$$\mathcal{R}_S(\phi_4) = \{\alpha, \beta \mid \exists \gamma \quad (\alpha, \texttt{lvIn/isL}, \gamma) \wedge (\gamma, \texttt{isL/dw}^+, \beta) \wedge \eta_A(\gamma) \in \{\texttt{REG}\}\}$$

We can notice that the rewritten query allows the database engine to avoid computing the transitive closure of isL *at all.*

4 System Implementation

We analyse the performance of queries augmented with schema information. The approach has been implemented as a system whose architecture is depicted in Figure 10. Our system architecture consists of three main modules *(i) Rewriter*, *(ii) Translator* and *(iii) Backend*.

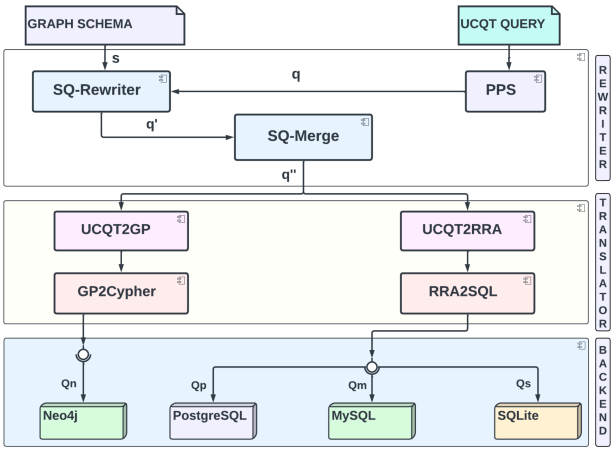

Figure 10: System architecture.

Rewriter. The rewriter module consists of three components *Preliminary Path Simplifier* (PPS), *Schema-based Query Rewriter* (SQ-Rewriter) and *Schema-based Query Merge* (SQ-Merge). The PPS component takes a UCQT query as input, then applies the preliminary path simplification rules presented in Fig. 6 and produces a simplified UCQT query q as output. The SQ-Rewriter component implements the query rewritings leveraging schema information presented in Sec. 3 for the UCQT query language (defined in Section 2.4). It takes a UCQT query q and a graph schema s described in the graph schema formalism (using Def. 1) as input and generates a schema-aware UCQT query q' as output. The SQ-Merge component uses the graph schema to remove redundant annotations by merging path expressions as described in Sec. 3.2. It takes a schema-aware UCQT query q' as input and produces merged UCQT query q'' as output.

Translator. The translator module compiles an UCQT query into a graph query that can be executed by a graph database management system (GDBMS) and a recursive SQL query that can be executed by a relational database management system (RDBMS). The translator module uses the UCQT2GP component to convert the UCQT query into a union of graph patterns, which is then transformed into a Cypher query by the GP2Cypher component. Due to the limited expressive power offered by Cypher [98], only UC2RPQ graph patterns (not UCQT) can be converted into Cypher. Since we use the UCQT

graph query language formalism to express graph patterns, our approach extends beyond Cypher: it can also be applied to other practical graph query languages such as SPARQL and PGQL [98]. If one starts from a query written in any of these formalisms, it can easily be translated into UCQT before being given as input to our system.

The translator module uses the UCQT2RRA component to translate UCQT query into a term in recursive relational algebra, and that is further optimised using the μ-RA system based on the rewritings proposed in [70]. Compared to the path expressions considered in [70], we consider additional rules for translating *conjunction* and *branching* operations as shown in Tab. 2.

Table 2: Conjunction and branching translation to RRA.

Path expression	RRA term
$(\!\|\phi_1 \cap \phi_2\|\!) =$	$\{\rho_n^{\mathrm{Tr}}\left(\rho_m^{\mathrm{Sr}}\left(\rho_{\mathrm{Tr}}^n\left(\rho_{\mathrm{Sr}}^m(\varphi_1)\right) \bowtie \rho_{\mathrm{Tr}}^n\left(\rho_{\mathrm{Sr}}^m(\varphi_2)\right)\right)\right) \mid \varphi_1 \in (\!\|\phi_1\|\!) \wedge \varphi_2 \in (\!\|\phi_2\|\!)\}$
$(\!\|\phi_1[\phi_2]\|\!) =$	$\{\rho_m^{\mathrm{Tr}}\left(\rho_{\mathrm{Tr}}^m(\varphi_1) \bowtie \rho_{\mathrm{Sr}}^m(\pi_{\mathrm{Sr}}(\varphi_2))\right) \mid \varphi_1 \in (\!\|\phi_1\|\!) \wedge \varphi_2 \in (\!\|\phi_2\|\!)\}$
$(\!\|[\phi_1]\phi_2\|\!) =$	$\{\rho_m^{\mathrm{Sr}}\left(\rho_{\mathrm{Sr}}^m(\pi_{\mathrm{Sr}}(\varphi_1)) \bowtie \rho_{\mathrm{Sr}}^m(\varphi_2)\right) \mid \varphi_1 \in (\!\|\phi_1\|\!) \wedge \varphi_2 \in (\!\|\phi_2\|\!)\}$

Our system architecture is modular, allowing for components such as UCQT2RRA (currently based on the μ-RA system) to be replaced with other systems such as α-extended RA [4], β-RA [57], WAVEGUIDE [111], AVANTGRAPH [75] and Datalog based RRA systems [11, 76, 103]. However, as discussed in [49, 70], the μ-RA system is superior in query optimisation capabilities.

The optimised RRA term is transformed into a concrete recursive SQL syntax for each RDBMS using the RRA2SQL component where the fixpoints are translated into *recursive view* statements[6].

Backend. To evaluate our approach, we conducted experiments on one GDBMS and three RDBMS. Specifically, we used Neo4j, PostgreSQL, MySQL, and SQLite. Graph databases can be directly stored as property graphs on Neo4j. However, in the relational data model, we represent the nodes and edges of the graph database as relational tables.

For instance, in the YAGO dataset's graph representation (as shown in Fig. 2), there are four graph edges labeled isLocatedIn connecting different pairs of nodes labeled as (PROPERTY, CITY), (CITY, REGION), and (REGION, COUNTRY) respectively. In the relational representation of the YAGO dataset, a table is created for each type of edge label, each with at least two columns Sr and Tr, which are foreign keys pointing to the source and target nodes. Similarly, a table is created for each type of node label, with a specific column Sr serving as a primary key and potentially many other columns for properties. Fig. 11 shows two node tables (*PROPERTY, CITY*) and two edge tables (*owns, isLocatedIn*). Overall, each row in node or edge tables represents a node or an edge of the graph database.

5 Experiments

We assess the impact on performance of the schema-based approach experimentally with a complete prototype implementation. Concretely, we try to answer the following questions: (i) Does the schema-based approach improve the performance of query evaluation on real and synthetic benchmark datasets? (ii) How does the approach behave on different database management systems?

[6]MySQL: CREATE OR REPLACE VIEW WITH RECURSIVE, SQLite: CREATE VIEW WITH RECURSIVE and PostgreSQL: CREATE TEMPORARY RECURSIVE VIEW

owns

Sr	Tr
n_2	n_1

PROPERTY

Sr	address
n_1	7 Queen Street

isLocatedIn

Sr	Tr
n_1	n_6
n_6	n_5
n_4	n_5
n_5	n_7

CITY

Sr	name
n_4	Elerslie
n_6	Montbonnot

Figure 11: Relational representation of nodes and edges.

5.1 Experimental Setup

5.1.1 Datasets. We consider datasets of different nature:

- YAGO [110], which is a real knowledge graph. We use a preprocessed and cleaned version of the real-world dataset YAGO2s [110] in which only nodes with unique identifiers are present. We split the set of RDF triples into multiple edge relations (tables), one for each predicate name. We create a node relation (table) for each node class. The dataset conforms to the YAGO schema constructed for this study. In other terms, the constructed schema does not exclude any existing triples from this YAGO dataset.
- the Social Network Benchmark (SNB) interactive workload from the Linked Data Benchmark Council (LDBC) [48] which is a synthetic reference benchmark for property graphs. Specifically, we used the LDBC-SNB dataset in CSV format provided from [102].

Table 3: Summary of dataset characteristics.

Name	SF	#NR	#ER	#Nodes	#Edges	Size
YAGO	N/A	7	88	98,582	150,391,592	26 GB
LDBC-SNB	0.1	8	16	416,311	2,034,983	0.3 GB
	0.3			1,154,108	6,235,570	0.9 GB
	1			3,966,203	23,056,025	3.3 GB
	3			11,407,480	69,482,982	9.9 GB
	10			36,485,994	231,532,873	33 GB
	30			88,789,972	541,279,759	82 GB

Tab. 3 summarises the characteristics of the pre-processed datasets. The YAGO dataset has 98k nodes and 150M edges, with 7 node relations and 88 edge relations. LDBC-SNB uses 8 node relations (**NR**) and 16 edge relations (**ER**). LDBC-SNB has property graphs of varying sizes measured by scale factors (**SF**), ranging from 0.1 to 30. We consider 6 of them shown in Tab. 3. LDBC property graph with SF 0.1 has 416k nodes and 2M edges, SF 30 has 88M nodes and 541M edges [7]. The size column in Tab. 3 shows the size on disk of the PostgreSQL database.

5.1.2 Schemas. YAGO does not have a graph schema, therefore we developed a basic one, as illustrated in Fig. 1 [8], inspired by the SHACL semantic constraints from [100] that specify the disjointness of certain classes. The LDBC-SNB dataset comes with a pre-defined property graph schema [48].

[7]LDBC-SNB contains large property graphs with scale factors 100, 1000

[8]Due to space constraints, we only provide a small schema for YAGO in Fig. 1. The complete schema is available as supplementary material (see Sec. 5.1.7).

Table 4: Queries for the LDBC-SNB Dataset.

Lab	Path expressions as CQT queries	Typ
IC1	x1, x2 ← (x1, knows1..3/(isL|(workAt|studyAt)/isL), x2)	NQ
IC2	x1, x2 ← (x1, knows/-hasC, x2)	NQ
IC6	x1, x2 ← (x1, knows1..2/(-hasC[hasT])[hasT], x2)	NQ
IC7	x1, x2 ← (x1, (-hasC/-likes)|((-hasC / -likes) ∩ knows), x2)	NQ
IC8	x1, x2 ← (x1, -hasC/-replyOf/hasC, x2)	NQ
IC9	x1, x2 ← (x1, knows1..2/-hasC, x2)	NQ
IC11	x1, x2 ← (x1, knows1..2/workAt/isL, x2)	NQ
IC12	x1, x2 ← (x1, knows/-hasC/replyOf/hasT/hasTY/isSubC+, x2)	RQ
IC13	x1, x2 ← (x1, knows+, x2)	RQ
IC14	x1, x2 ← (x1, (knows ∩ (-hasC/replyOf/hasC))+, x2)	RQ
Y1	x1, x2 ← (x1, knows+/studyAt/isL+/isP+, x2)	RQ
Y2	x1, x2 ← (x1, likes/hasC/knows+/isL+, x2)	RQ
Y3	x1, x2 ← (x1, likes/replyOf+/isL+/isP+, x2)	RQ
Y4	x1, x2 ← (x1, hasM/(studyAt|workAt)/isL+/isP+, x2)	RQ
Y5	x1, x2 ← (x1, -hasM/([cof]hasT)/hasTY/isSubC+, x2)	RQ
Y6	x1, x2 ← (x1, replyOf+/isL+/isP+, x2)	RQ
Y7	x1, x2 ← (x1, hasMod/hasI/hasTY/isSubC+, x2)	RQ
Y8	x1, x2 ← (x1, ([cof/hasC]hasM)/isL/isP+, x2)	RQ
IS2	x1, x2 ← (x1, -hasC/replyOf+/hasC, x2)	RQ
IS6	x1, x2 ← (x1, replyOf+/-cof/hasM, x2)	RQ
IS7	x1, x2 ← (x1, (-hasC/replyOf/hasC)|((-hasC/replyOf/hasC) ∩ knows), x2)	NQ
BI11	x1, x2 ← (x1, (([isL/isP]knows)[isL/isP]) ∩ (knows/([isL/isP]knows))) ,x2)	NQ
BI10	x1, x2 ← (x1, (knows+[isL/isP])/(-hasC[hasT])/hasT/hasTY, x2)	RQ
BI3	x1, x2 ← (x1, -isP/-isL/-hasMod/cof/-replyOf+/hasT/hasTY, x2)	RQ
BI9	x1, x2 ← (x1, replyOf+/hasC, x2)	RQ
BI20	x1, x2 ← (x1, (knows ∩ (studyAt/-studyAt))+, x2)	RQ
LSQB1	x1, x2 ← (x1, -isP/-isL/-hasM/cof/-replyOf+/hasT/hasTY, x2)	RQ
LSQB4	x1, x2 ← (x1, ((likes[hasT])[-replyOf])/hasC, x2)	NQ
LSQB5	x1, x2 ← (x1, -hasT/-replyOf/hasT, x2)	NQ
LSQB6	x1, x2 ← (x1, knows/knows/hasI, x2)	NQ

isL=isLocatedIn, hasT=hasTag, isP=isPartOf, isSubC=isSubClassOf, Symbol | = ∪
hasTY=hasType, coF=containerOf, hasMod=hasModerator, hasC=hasCreator,
Lab=LDBC query label, **Shp**=Shape, **Typ**=Type, hasM=hasMember, hasI=hasInterest

5.1.3 Queries. We selected 30 queries from the LDBC-SNB workload [48], divided into two categories: non-recursive graph queries (**NQ**) and recursive graph queries (**RQ**). As shown in Tab. 4 out of the 30 queries, 12 are non-recursive, and 18 are recursive. Out of the 30 queries of Tab. 4, 22 are third-party queries. Queries labeled as (**IC** and **IS**) are extracted from the LDBC interactive workload, while queries labeled as (**BI**) are from the business intelligence workload, and queries labeled as (**LSQB**) are extracted from the large-scale subgraph query benchmark [84]. Finally, we proposed queries labeled as (**Y**) as complementary queries, inspired by YAGO-style queries found in [70].

We consider 18 queries on the YAGO dataset. These queries were previously used in studies such as [3, 58, 111]. All the queries selected for the YAGO dataset are recursive graph queries.

5.1.4 Baseline. All experiments conducted using the schema-based approach are systematically compared to the initial (non-schema enriched) query considered as a baseline.

5.1.5 Measures and timeout. We set a 30-minute computation time limit for each query to evaluate schema-based and baseline approaches. If the computation exceeded 30 minutes, we stopped the process and deemed the query infeasible with the given approach, allowing us to measure the system's performance in a reasonable time frame. Each reported query execution time corresponds to the average of 5 runs.

In experiments, some queries result in timeouts at a given scale factor but not with smaller dataset sizes. The primary focus is to measure the relative performance improvements brought by the proposed optimisations rather than conducting pure performance comparisons between systems. In particular, we want to check if such optimisations are capable of turning infeasible queries to feasible ones at some scale factors on the various systems.

5.1.6 Hardware and software setup. All experiments have been conducted by using a Macbook Air laptop with Apple M2 chip, 8 cores (4 performance and 4 efficiency), 24 GB of RAM and 1 TB hard disk. The main backend used for evaluation is PostgreSQL 15.4. We also provide performance comparisons with the graph database system Neo4j 5.21.2 community edition. We do not report on comparisons with MySQL and SQLite because both systems are significantly less efficient than PostgreSQL in executing queries for large LDBC datasets. No queries could be succesfully executed on MySQL and SQLite beyond scale factor 1.

5.1.7 Availability. The datasets, schemas, initial and rewritten queries are available[9].

5.2 Query feasibility

For the YAGO dataset, 17 queries out of 18 did run successfully with both approaches within the allowed timeout period. Only one query (query 7) timed out. Furthermore, during the process of schema enrichment and query factorization, query 7 automatically reverted to the initial query form since no schema-based optimization was found. The LDBC dataset consists of ten queries (*IC2, IC6, IC7, IC9, IC13, Y7, BI11, BI9, BI20*, and *LSQB6*) that returned to their initial path expressions after going through the schema enrichment and query factorization process. Among these, six are non-recursive, while the remaining four are recursive.

These queries highlight an advantage of our approach, which is to automatically avoid schema enrichment when it does not detect any significant performance gain. This allows us to prevent query performance degradation. For example, query 7 for YAGO and ten LDBC queries did not benefit from schema enrichment, but their performance was not made worse.

Out of ten LDBC queries, only three queries *Y7, BI11* and *BI20* could be executed on all six scale factors. Tab. 5 summarizes the number and percentage of successful runs for the various scale factors (data size) on the LDBC dataset.

Table 5: LDBC query feasibility across six scale factors (SF).

SF	Recursive				Non Recursive			
	Baseline		Schema		Baseline		Schema	
	Count	Percentage	Count	Percentage	Count	Percentage	Count	Percentage
0.1	18	**100**	18	**100**	12	**100**	12	**100**
0.3	16	**88.9**	18	**100**	9	**75**	9	**75**
1	14	**77.8**	15	**83.3**	9	**75**	9	**75**
3	11	**61.1**	13	**72.2**	7	**58.3**	7	**58.3**
10	10	**55.6**	11	**61.1**	7	**58.3**	7	**58.3**
30	5	**27.8**	7	**38.9**	4	**33.3**	4	**33.3**

As shown in Tab. 5, the percentage of queries that did timeout increase with the scale factor for both approaches. For scale factor 30, the baseline approach could only execute 27.8% of recursive graph queries, whereas the schema-based approach could execute 38.9% recursive graph queries. Both approaches successfully executed the same number of non-recursive graph queries across all scale factors. Using the schema-based approach, more recursive graph queries could be executed compared to the baseline across all scale factors, as shown in Tab. 5.

We now evaluate the impact of the proposed approach on performance.

[9] https://anonymous.4open.science/r/Schema_Graphs_Dataset-E883/

5.3 Performance results on YAGO

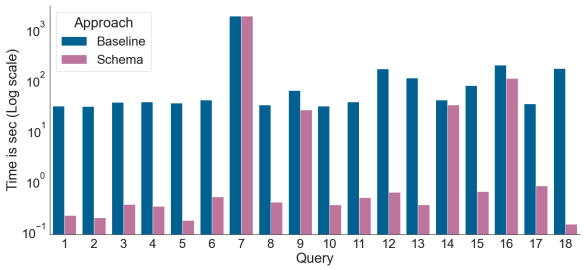

Figure 12: Query runtime for YAGO dataset.

Fig. 12 presents the evaluation times for the schema-based approach compared to the baseline. We observe that the schema-based approach outperforms the baseline for all YAGO queries; on average, YAGO queries run 6.1 times faster using the schema-based approach compared to the baseline. Notice that the time scale is logarithmic.

As mentioned in Sec. 3, the schema-enrichment approach enables the removal of the transitive closure operation from certain queries. Tab. 6 illustrates the replacement of the sub-path expression `isLocatedIn+` by fixed-length paths in YAGO queries.

Table 6: Statistics on generated fixed-length paths

YAGO Queries	#Paths	Min	Avg	Max
1,2,3,4,5	1	2	2	2
12	2	1	1.5	2
6,8,10,11,14,15,16,17,18	3	1	2	3
9	8	1	2.5	4

Tab. 6 displays the total number of fixed length paths (**#Paths**) along with minimum (**Min**), average (**Avg**) and maximum (**Max**) lengths of paths generated as replacement for transitive closure. Specifically, the transitive closure operation can be eliminated in 16 out of 18 queries for the YAGO dataset.

All (third-party) queries considered in these YAGO experiments happen to be recursive (**RQ**) graph queries. In order to further experimentally assess the impact of the approach on performance, we next investigate with graphs of different topology and varying sizes, in combination with recursive and non-recursive queries.

5.4 Performance results on LDBC

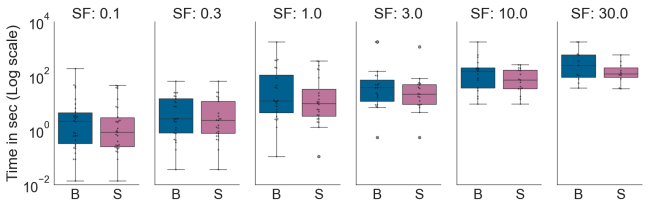

Figure 13: Box plot of query runtime for six scale factors (SF=Scale Factor, B=Baseline, S=Schema)

We execute the 30 queries of Tab. 4 for the six different scales of the LDBC-SNB datasets, for both the schema-based approach and the baseline. Therefore each query is run $6 \times 2 = 12$ times, which yields 360 query runs. Some queries timed out. Among the successful executions, times spent in evaluations can significantly vary from a query to another. In order to extract useful insights from these measurements, we resort to statistical measures and aggregations on successful executions.

Performance analysis when dataset size varies. We test the hypothesis that performance of query execution is improved by the schema-based approach as compared to the baseline. We conduct this test across all scale factors. Fig. 13, presents a summary of statistics based on box plots for all the successful runs of the 30 queries of Tab. 4 for the two approaches for graphs of increasing sizes. Running times reported on the y-axis use a logarithmic scale. This suggests that the schema-based approach is more efficient for executing queries, especially when the dataset size increases. Notably, the performance of the schema-based approach improved after scale factor 0.3, as depicted in Fig. 13.

Performance analysis between recursive and non recursive queries. To further analyze the benefits of the schema-based approach, we categorize queries into recursive and non-recursive. Then, we collectively analyze both approaches across all six scale factors. Tab. 7 compares schema-based and baseline (non-schema-based) approaches in query runtime. Here, `Count`, `Min`, `Q1`, `Q2`, `Q3`, `Max`, and `Mean` represent the total number of queries, minimum, 25^{th} percentile, median, 75^{th} percentile, maximum and mean query runtime in seconds, respectively.

Table 7: Query runtime summary statistics (in seconds)

	Recursive		Non Recursive	
	Baseline	Schema	Baseline	Schema
Count	78	78	48	48
Min	**0.0137**	**0.0137**	**0.087**	**0.087**
Q1	**2.839**	**1.968**	1.211	1.271
Q2	**16.254**	**11.284**	**10.574**	**9.515**
Q3	**140.446**	**46.759**	**45.397**	**44.938**
Max	**1800.0**	**1171.876**	**360.887**	**353.044**
Mean	**213.282**	**65.244**	**46.727**	**45.398**

As reported in Tab. 7 comparing the schema-based approach and the baseline for recursive queries across all six scale factors, the results demonstrate that the schema-based approach outperforms the baseline. Specifically, the schema-based approach is 3.26 times faster on average than the baseline for recursive queries. Both approaches exhibit comparable performance when examining non-recursive queries. However, it is worth noticing that the schema-based approach generally demonstrates faster median, 75th percentile, maximum and mean query runtimes compared to the baseline.

Table 8: Overall analysis of query runtime (in seconds)

	Count	Min	Q1	Median	Q3	Max	Mean
Baseline	126	**0.0137**	**2.6**	**14.45**	**80.37**	**1800.0**	**149.833**
Schema	126	**0.0137**	**1.38**	**10.45**	**46.76**	**1171.87**	**58.066**

Measurements in Tab. 8 suggest that the schema-based approach consistently outperforms the baseline, with an average improvement of 2.58 times for feasible queries at all six scale factors.

Experimental statistics show that the effect of the schema-enrichment process is even more significant for YAGO compared to LDBC. First, YAGO queries provide more optimisation opportunities for transitive closure removal. In contrast, the transitive closure operation can only be removed in 5 out of the 30 LDBC queries. Second, YAGO queries also provide more opportunities for semi-join insertion as they use less branching and conjunction operators. Overall, 10 out of 30 LDBC queries returned to their original UCQT form. In contrast, only 1 out of 18 YAGO queries, reverted to its initial form. When the query reverts to its original form, the schema-based approach obviously has the same runtime than the baseline. This affects the overall performance impact and explains the more favorable aggregate statistics for YAGO queries.

5.5 Evaluation on other database systems

We now conduct experiments to evaluate the schema-enrichment approach on PostgreSQL relational database system and Neo4j graph database system. We conducted experiments on the LDBC-SNB dataset, focusing on *chain-shaped* queries [27]. These queries are expressed without branching and conjunction operations and correspond to the formalism of the union of two-way conjunctive regular path queries (UC2RPQ). The Cypher query language used in the Neo4j graph database only supports a restricted form of UC2RPQ [98]. In particular, only 15 out of 30 queries from the LDBC-SNB dataset shown in Tab. 4 are expressible in Cypher and hence supported by Neo4j.

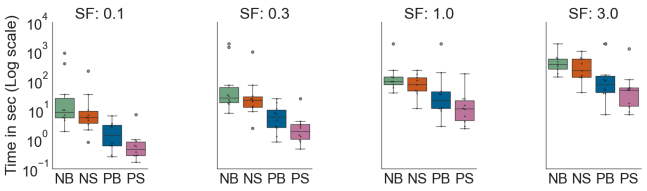

Figure 14: Query runtimes on Neo4j (N) and PostgreSQL (P) for LDBC-SNB. (SF=Scale Factor, B=Baseline, S=Schema, PB=PostgreSQLBaseline)

Fig. 14, provides a comprehensive overview of the summary statistics for query runtimes using box plots specifically for the LDBC-SNB dataset at scale factors of 0.1, 0.3, 1, and 3. For scale factors 10, and 30, Neo4j could not complete the query evaluations within 30 minutes. Results shown in Fig. 14 suggest that the schema-based approach improves the performance on each individual system. Overall, the median query runtime of the schema-based approach is consistently better than the baseline across both database systems considered in this study. From our results we observed that PostgreSQL offers more scalability as it is capable of handling scale factors 10 and 30 (See Fig. 13), and yet its performance still benefits from the schema-based enrichment approach. These performance improvements are particularly important at scale factor 30.

Measurements with Neo4j are not reported for YAGO queries as it is by far the most inefficient system for evaluating YAGO queries. This was also noticed by [70].

Plan-level impact of annotated path expressions. To illustrate the plan-level impact of the schema enrichment process on annotated path expressions, we consider a query Q_1 and its schema-enriched version as query Q_2 in the LDBC dataset.

$$S, T \leftarrow (S, \texttt{knows/workAt/isLocatedIn}, T) \quad Q_1$$

$$S, T \leftarrow (S, \texttt{knows/workAt/}_{\texttt{Organisation}}\texttt{isLocatedIn}, T) \quad Q_2$$

The translated versions of queries Q_1 and Q_2 in SQL and Cypher are presented in Figures 15 and 16, respectively.

```
1 // SCHEMA-ENRICHED (Q2)
2 SELECT DISTINCT kw.Sr AS S, isLin.Tr AS T
3   FROM knows  AS kw
4   JOIN workAt AS wr ON kw.Tr=wr.Sr
5   JOIN (SELECT loc.Sr AS Sr, loc.Tr AS Tr
6           FROM (SELECT Sr FROM Organisation) AS org
7           JOIN isLocatedIn AS loc ON loc.Sr=org.Sr
8       ) AS isLin ON wr.Tr=isLin.Sr;
```

```
1 // BASELINE (Q1)
2 SELECT DISTINCT kw.Sr AS S, isLin.Tr AS T
3   FROM knows  AS kw
4   JOIN workAt AS wr ON kw.Tr=wr.Sr
5   JOIN isLocatedIn AS isLin ON wr.Tr=isLin.Sr;
```

Figure 15: Schema-enriched and baseline queries in SQL

In Fig. 15, the schema-enriched version of the SQL query contains an additional *semi-join* between the Organisation node relation and isLocatedIn edge relation (lines 5-7). The result of this *semi-join* is then combined with the workAt edge relation (line 8). On the other hand, in the baseline version of the query, additional node relation-based *semi-join* is not performed.

```
1 // SCHEMA-ENRICHED (Q2)
2 MATCH (S)-[:knows]->()-[:workAt]->(:org)-[:isLin]->(T)
3 RETURN DISTINCT S, T;
```

```
1 // BASELINE (Q1)
2 MATCH (S)-[:knows]->()-[:workAt]->()-[:isLin]->(T)
3 RETURN DISTINCT S, T;
```

Figure 16: Schema-enriched and baseline queries in Cypher (isLin=isLocatedIn, org=Organisation)

In Fig. 16, the schema-enriched version of the Cypher query includes an extra node label in the graph pattern (line 2). The schema-enriched graph pattern indicates the selection of isLocatedIn labeled edges that start from nodes labeled as Organisation in the graph database. In contrast, the baseline version of the query in Cypher does not impose such constraints on the graph pattern.

In order to concretely illustrate the reduction of intermediate results enabled by the schema-enrichment process, Fig. 17 presents the schema-enriched and baseline query execution plans for queries Q_2 and Q_1, annotated with costs and cardinalities as estimated by PostgreSQL. In Fig. 17, the schema-enriched execution plan generated for query Q_2, indicates that the number of intermediate rows significantly decreased after a *semi-join* is performed between the Organisation node relation and isLocatedIn edge relation (lines 12-16). The isLocatedIn edge relation contains 11 million rows, however, after the *semi-join*, the number of rows reduced to approximately 8 thousand. In the schema-enriched execution plan, the result of the *semi-join* is combined with the workAt edge relation

```
1 // SCHEMA-ENRICHED (Q2)
2  HashAggregate (cost=215,265.50 rows=2,085,899)
3   Group Key: knows.Sr, islocatedin.Tr
4   Planned Partitions: 32
5    Hash Join (cost=40,408.50 rows=2,085,899)
6     Hash Cond: (knows.Tr = workat.Sr)
7      Seq Scan on knows (cost=12,221.46 rows=704,246)
8     Hash (cost=2,190.11 rows=58,912)
9      Hash Join (cost=2,190.11 rows=58,912)
10      Hash Cond: (workat.Tr = Organisation.Sr)
11       Seq Scan on workat (cost=965.12 rows=58,912)
12      Hash (cost=315.51 rows=7,955)
13       Merge Join (cost=315.51 rows=7,955)
14        Merge Cond: (Organisation.Sr = islocatedin.Sr)
15         Index Scan on Organisation(cost=215.61 rows=7,955)
16         Index Scan on islocatedin(cost=337,785.74 rows=11M)
```

```
1 // BASELINE (Q1)
2  HashAggregate (cost=219,592.34 rows=2,085,899)
3   Group Key: knows.Sr, islocatedin.Tr
4   Planned Partitions: 32
5    Hash Join (cost=44,735.33 rows=2,085,899)
6     Hash Cond: (knows.Tr = workat.Sr)
7      Seq Scan on knows (cost=12,221.46 rows=704,246)
8     Hash (cost=6,516.95 rows=58,912)
9      Merge Join (cost=6,516.95 rows=58,912)
10      Merge Cond: (islocatedin.Sr = workat.Tr)
11       Index Scan on islocatedin (cost=337,785.74 rows=11M)
12      Sort (cost=5,780.08 rows=58,912)
13       Sort Key: workat.Tr
14        Seq Scan on workat (cost=965.12 rows=58,912)
```

Figure 17: Execution plans for schema-enriched and baseline queries (`where 11M = 11,118,487 rows`)

(lines 9-11), where the total number of estimated rows is 58,912 for an estimated cost of 2,190.11 (line 9). Conversely, in Fig. 17, the baseline execution plan generated for query Q_1, the `isLocatedIn` edge relation does not undergo filtering. It is directly joined with the `workAt` edge relation (line 10), for an estimated higher cost of 6,516.95 (line 9). The same number of rows is estimated for both the baseline and the schema-enriched execution plans (line 9).

Overall, as shown in line 2 of Fig. 17, the schema-enriched plan has a smaller estimated cost compared to the baseline for the same number of rows in the final result set.

6 Related Work

Query rewriting techniques that rely on the structural information of the schema have been proposed [39]. Authors of [2, 33, 35, 105] emphasize the schema's significance in the query rewriting. Additionally, authors in [32, 50] suggest that knowing the structure of the database can help reduce the query search space, leading to a significant improvement in overall query runtime. We now briefly discuss schema-based query rewriting techniques proposed for databases following different data models.

Semi-structured databases. Schema-based query rewriting techniques for queries over *semi-structured databases* is proposed in [2, 33, 35]. In semi-structured databases, schemas are expressed as *path constraints* that are regular expressions defined over edge labels [34]. The use of path constraints to rewrite queries expressed in the formalism of C2RPQ and UC2RPQ is proposed in [36, 46, 51]. However, a significant limitation of using path constraints as a schema language is that the schema database consistency can only be established when path constraints are defined without using the Kleene star operator [35].

XML databases. Authors in [87, 108] propose rewriting XPath queries using the structural information stored in the schema of XML databases. For XML databases, schemas are expressed as *Document Type Definition* (DTDs), which are essentially regular expressions defined over the edge labels [81]. Authors [18, 43, 55, 65, 73, 85] study the *satisfiability* of XPath queries in presence of DTDs and suggest that satisfiability is undecidable for XPath queries in presence of recursive DTDs (DTDs defined using Kleene star operator). Authors in [31, 80] propose the creation of smaller XML documents by using the XPath query and structure of XML schema. Authors in [19] suggest a type system-based approach for XML document pruning; however, they highlight that creating pruned XML documents can be time-consuming and may take a similar amount of time as running the original query.

Datalog. Schema-based query rewriting of *conjunctive queries* (CQ) and *union of conjunctive queries* (UCQ) in the presence of schema expressed as Datalog rules is proposed in [82]. Furthermore, authors [13] suggest that query containment of CQ and UCQ is decidable in the presence of schema expressed as non-recursive Datalog. Authors [12, 44, 95, 105] study the *containment* of Datalog queries in the presence of schema and suggest that the containment is decidable in the presence of non-recursive schemas. Regarding graph query language formalism, authors in [20, 36, 38] express the formalisms of C2RPQ and UC2RPQ as Datalog queries and suggest that the query containment is decidable in the presence of non-recursive Datalog schema.

Graph databases. Graph query and schema languages have been extensively researched in the context of knowledge graphs [17, 68, 107] with RDFS and OWL in particular [64, 79], and the standard query language SPARQL [28, 40, 41, 59]. Query rewriting based on the structure of the *Resource Description Framework* (RDF) has been proposed in [1, 72] for non-recursive SPARQL queries.

For property graphs, significant research and standardisation effort are still in progress [6, 21, 22, 29, 45, 53, 92–94]. So far, the lack of standard schema language hinders the application of schema-based query rewriting techniques. Contemporary research in property graph data models mainly focuses on property graph schema design and inference techniques [5, 23, 24, 26, 74, 90, 94]. The design of our graph schema is motivated by existing works such as *PG-Schema* and *PG-Keys* proposed in [8, 9].

In [42], a type inference approach is proposed to rewrite queries expressed in the formalisms of RPQ and 2RPQ using a recursive graph schema. However, the type inference system presented in [42] is neither sound nor complete for graph query language formalisms containing branching and conjunction operations. Additionally, the type inference system is only explored theoretically.

Compared to the state-of-the-art, our approach can take a UCQT query and a graph schema as input and generate a UCQT query as output, which is enhanced with structural schema information.

The rewritten schema-aware query preserves the initial query semantics under the graph schema. We experimentally demonstrate that for *recursive* UCQTs, the generated UCQT can be executed more efficiently using our approach.

7 Conclusion and Perspectives

We propose a graph query rewriting method aimed at leveraging the structural information of a graph schema. The purpose is to enrich an initial query with schema information in order to improve query performance. To this end, we introduce inference rules capable of incorporating schema constraints in the path expressions contained in a query. The difficulty comes from the fact that pushing constraints through regular path expressions is complex. Our approach automates a process which would be tricky and error-prone if done manually by a developer. The soundness and completeness of the approach are proved to ensure that the initial query semantics is preserved under the schema. Furthermore, our approach is opportunistic in that it is applied only when performance gains are expected. We conducted extensive experiments on real and synthetic datasets, with several database systems. Experimental results show that schema-based query rewriting provides significant performance gains for recursive path queries in graphs. A perspective for further work is to extend the approach by considering queries with aggregrations.

References

[1] Abdullah Abbas, Pierre Genevès, Cécile Roisin, and Nabil Layaïda. 2017. Optimising SPARQL query evaluation in the presence of shex constraints. In *BDA 2017-33ème conférence sur la «Gestion de Données—Principes, Technologies et Applications»*. 1–12.

[2] Serge Abiteboul and Victor Vianu. 1997. Regular path queries with constraints. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 122–133.

[3] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Shadi Ghajar-Khosravi, and Mark H Chignell. 2017. Tasweet: optimizing disjunctive regular path queries in graph databases. In *EDBT/ICDT 2017 Joint Conference 20th International Conference on Extending Database Technology*. 470–473.

[4] Rakesh Agrawal. 1988. Alpha: An extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering* 14, 7 (1988), 879–885.

[5] Rana Alotaibi, Chuan Lei, Abdul Quamar, Vasilis Efthymiou, and Fatma Özcan. 2021. Property graph schema optimization for domain-specific knowledge graphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 924–935.

[6] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*. 1421–1432.

[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–40.

[8] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. 2023. PG-Schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[9] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W Hare, Jan Hidders, Victor E Lee, Bei Li, Leonid Libkin, Wim Martens, et al. 2021. Pg-keys: Keys for property graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 2423–2436.

[10] Renzo Angles, Harsh Thakkar, and Dominik Tomaszuk. 2020. Mapping RDF databases to property graph databases. *IEEE Access* 8 (2020), 86091–86110.

[11] Molham Aref, Balder ten Cate, Todd J Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L Veldhuizen, and Geoffrey Washburn. 2015. Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1371–1382.

[12] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*. 1–15.

[13] Pablo Barceló, Diego Figueira, Georg Gottlob, and Andreas Pieris. 2020. Semantic optimization of conjunctive queries. *Journal of the ACM (JACM)* 67, 6 (2020), 1–60.

[14] Pablo Barceló, Jorge Pérez, and Juan Reutter. 2013. Schema mappings and data exchange for graph databases. In *Proceedings of the 16th International Conference on Database Theory*. 189–200.

[15] Chris Barrett, Riko Jacob, and Madhav Marathe. 2000. Formal-language-constrained path problems. *SIAM J. Comput.* 30, 3 (2000), 809–837.

[16] Gordon Bell, Tony Hey, and Alex Szalay. 2009. Beyond the data deluge. *Science* 323, 5919 (2009), 1297–1298.

[17] Luigi Bellomarini, Andrea Gentili, Eleonora Laurenza, and Emanuel Sallinger. 2022. Model-Independent Design of Knowledge Graphs-Lessons Learnt From Complex Financial Graphs.. In *EDBT*. 2–524.

[18] Michael Benedikt, Wenfei Fan, and Floris Geerts. 2008. XPath satisfiability in the presence of DTDs. *Journal of the ACM (JACM)* 55, 2 (2008), 1–79.

[19] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguyen. 2006. Type-Based XML Projection.. In *VLDB*, Vol. 6. 271–282.

[20] Piero A Bonatti. 2004. On the decidability of containment of recursive datalog queries-preliminary report. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 297–306.

[21] Angela Bonifati. 2021. Graph processing systems back to the future. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. 1–1.

[22] Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Matthias Hofer, Wim Martens, Filip Murlak, Joshua Shinavier, Slawek Staworko, and Dominik Tomaszuk. 2023. Threshold Queries. *ACM SIGMOD Record* 52, 1 (2023), 64–73.

[23] Angela Bonifati, Stefania Dumbrava, Emile Martinez, Fatemeh Ghasemi, Malo Jaffré, Pacôme Luton, and Thomas Pickles. 2022. DiscoPG: property graph schema discovery and exploration. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3654–3657.

[24] Angela Bonifati, Stefania Dumbrava, Emile Martinez, and Nicolas Mir. 2023. The Quest for Schemas in Graph Databases. *Looking Ahead* 4 (2023), 5.

[25] Angela Bonifati, George Fletcher, Hannes Voigt, Nikolay Yakovets, and HV Jagadish. 2018. *Querying graphs*. Vol. 10. Springer.

[26] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema validation and evolution for graph databases. In *Conceptual Modeling: 38th International Conference, ER 2019, Salvador, Brazil, November 4–7, 2019, Proceedings 38*. Springer, 448–456.

[27] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *The VLDB Journal* 29, 2-3 (2020), 655–679.

[28] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. SHARQL: Shape analysis of recursive SPARQL queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2701–2704.

[29] Angela Bonifati and Hannes Voigt. 2022. Special issue on big graph data management and processing. *The VLDB Journal* 31, 2 (2022), 201–202.

[30] Mihaela A Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an efficient RDF store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 121–132.

[31] Stéphane Bressan, Barbara Catania, Zoé Lacroix, Ying Guang Li, and Anna Maddalena. 2005. Accelerating queries by pruning XML documents. *Data & Knowledge Engineering* 54, 2 (2005), 211–240.

[32] Peter Buneman, Susan Davidson, Mary Fernandez, and Dan Suciu. 1997. Adding structure to unstructured data. In *Database Theory—ICDT'97: 6th International Conference Delphi, Greece, January 8–10, 1997 Proceedings 6*. Springer, 336–350.

[33] Peter Buneman, Wenfei Fan, and Scott Weinstein. 1998. Path constraints on semistructured and structured data. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 129–138.

[34] Peter Buneman, Wenfei Fan, and Scott Weinstein. 2000. Path constraints in semistructured databases. *J. Comput. System Sci.* 61, 2 (2000), 146–193.

[35] Peter Buneman, Wenfei Fan, and Scott Weinstein. 2000. Query optimization for semistructured data using path constraints in a deterministic data model. In *Research Issues in Structured and Semistructured Database Programming: 7th International Workshop on Database Programming Languages, DBPL'99 Kinloch Rannoch, UK, September 1–3, 1999 Revised Papers 7*. Springer, 208–223.

[36] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. 1998. On the decidability of query containment under constraints. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 149–158.

[37] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y Vardi. 1999. Rewriting of regular expressions and regular path queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 194–204.

[38] Diego Calvanese, Giuseppe De Giacomo, and Moshe Y Vardi. 2005. Decidable containment of recursive queries. *Theoretical Computer Science* 336, 1 (2005), 33–56.

[39] Upen S Chakravarthy, John Grant, and Jack Minker. 1990. Logic-based approach to semantic query optimization. *ACM Transactions on Database Systems (TODS)* 15, 2 (1990), 162–207.

[40] Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. 2012. SPARQL query containment under SHI axioms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 26. 10–16.

[41] Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès, and Nabil Layaïda. 2018. SPARQL query containment under schema. *Journal on Data Semantics* 7, 3 (2018), 133–154.

[42] Dario Colazzo and Carlo Sartiani. 2015. Typing regular path query languages for data graphs. In *Proceedings of the 15th Symposium on Database Programming Languages*. 69–78.

[43] Wojciech Czerwiński, Wim Martens, Pawel Parys, and Marcin Przybylko. 2015. The (almost) complete guide to tree pattern containment. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 117–130.

[44] Oege de Moor, Damien Sereni, Pavel Avgustinov, and Mathieu Verbaere. 2008. Type inference for datalog and its application to query optimisation. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 291–300.

[45] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. 2022. Graph pattern matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International Conference on Management of Data*. 2246–2258.

[46] Alin Deutsch and Val Tannen. 2001. Optimization properties for classes of conjunctive regular path queries. In *International Workshop on Database Programming Languages*. Springer, 21–39.

[47] Oliver M Duschka, Michael R Genesereth, and Alon Y Levy. 2000. Recursive query plans for data integration. *The Journal of Logic Programming* 43, 1 (2000), 49–73.

[48] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[49] Amela Fejza, Pierre Genevès, Nabil Layaïda, and Sarah Chlyah. 2023. The μ-RA System for Recursive Path Queries over Graphs. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 5041–5045.

[50] Mary Fernandez and Dan Suciu. 1998. Optimizing regular path expressions using graph schemas. In *Proceedings 14th International Conference on Data Engineering*. IEEE, 14–23.

[51] Daniela Florescu, Alon Levy, and Dan Suciu. 1998. Query containment for conjunctive queries with regular expressions. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 139–148.

[52] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. GPC: A pattern calculus for property graphs. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 241–250.

[53] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. A Researchers Digest of GQL. In *The 26th International Conference on Database Theory, 2023*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 1–1.

[54] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*. 1433–1445.

[55] Floris Geerts and Wenfei Fan. 2005. Satisfiability of XPath queries with sibling axes. In *International Workshop on Database Programming Languages*. Springer, 122–137.

[56] Pierre Genevès and Nabil Layaïda. 2006. A system for the static analysis of XPath. *ACM Trans. Inf. Syst.* 24, 4 (2006), 475–502. https://doi.org/10.1145/1185877.1185882

[57] Luiz Gomes-Jr, Bernd Amann, and André Santanchè. 2015. Beta-algebra: Towards a relational algebra for graph analysis. In *EDBT/ICDT 2015 Joint Conference*, Vol. 157.

[58] Andrey Gubichev, Srikanta J Bedathur, and Stephan Seufert. 2013. Sparqling kleene: fast property paths in RDF-3X. In *First International Workshop on Graph Data Management Experiences and Systems*. 1–7.

[59] David Haller. 2023. A Query-Driven Approach for SHACL Type Inference. In *Conference on Very Large Data Bases (VLDB 2023)*.

[60] Jelle Hellings. 2018. On Tarski's Relation Algebra: querying trees and chains and the semi-join algebra.

[61] Jelle Hellings, Marc Gyssens, Jan Van Den Bussche, and Dirk Van Gucht. 2023. Expressive Completeness of Two-Variable First-Order Logic with Counting for First-Order Logic Queries on Rooted Unranked Trees. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–13.

[62] Jelle Hellings, Marc Gyssens, Yuqing Wu, Dirk Van Gucht, Jan Van den Bussche, Stijn Vansummeren, and George HL Fletcher. 2020. Comparing the expressiveness of downward fragments of the relation algebra with transitive closure on trees. *Information Systems* 89 (2020), 101467.

[63] Jelle Hellings, Catherine L Pilachowski, Dirk Van Gucht, Marc Gyssens, and Yuqing Wu. 2017. From relation algebra to semi-join algebra: An approach for graph query optimization. In *Proceedings of the 16th International Symposium on Database Programming Languages*. 1–10.

[64] Jakob Henriksson and Jan Małuszyński. 2004. Static type-checking of Datalog with ontologies. In *Principles and Practice of Semantic Web Reasoning: Second International Workshop, PPSWR 2004, St. Malo, France, September 6-10, 2004. Proceedings 2*. Springer, 76–89.

[65] Jan Hidders. 2004. Satisfiability of XPath expressions. In *Database Programming Languages: 9th International Workshop, DBPL 2003, Potsdam, Germany, September 6-8, 2003. Revised Papers 9*. Springer, 21–36.

[66] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. 2013. YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia. *Artif. Intell.* 194 (2013), 28–61. https://doi.org/10.1016/J.ARTINT.2012.06.001

[67] Aidan Hogan and Aidan Hogan. 2020. SPARQL query language. *The Web of Data* (2020), 323–448.

[68] Zhiwei Hu, Víctor Gutiérrez-Basulto, Zhiliang Xiang, Xiaoli Li, Ru Li, and Jeff Z Pan. 2022. Type-aware embeddings for multi-hop reasoning over knowledge graphs. *arXiv preprint arXiv:2205.00782* (2022).

[69] Reijo Jaakkola and Antti Kuusisto. 2023. Complexity Classifications via Algebraic Logic. In *31st EACSL Annual Conference on Computer Science Logic, CSL 2023*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[70] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaïda. 2020. On the optimization of recursive relational queries: Application to graph queries. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 681–697.

[71] Alekh Jindal, Praynaa Rawlani, Eugene Wu, Samuel Madden, Amol Deshpande, and Mike Stonebraker. 2014. Vertexica: your relational friend for graph analytics! (2014).

[72] HyeongSik Kim, Padmashree Ravindra, and Kemafor Anyanwu. 2017. Type-based semantic optimization for scalable RDF graph pattern matching. In *Proceedings of the 26th International Conference on World Wide Web*. 785–793.

[73] Laks VS Lakshmanan, Ganesh Ramesh, Hui Wang, and Zheng Zhao. 2004. On testing satisfiability of tree pattern queries. In *VLDB*, Vol. 4. 120–131.

[74] Hanâ Lbath, Angela Bonifati, and Russ Harmer. 2021. Schema inference for property graphs. In *EDBT 2021-24th International Conference on Extending Database Technology*. 499–504.

[75] Wilco v Leeuwen, Thomas Mulder, Bram van de Wall, George Fletcher, and Nikolay Yakovets. 2022. Avantgraph query processing engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3698–3701.

[76] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic (TOCL)* 7, 3 (2006), 499–562.

[77] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2013. Querying graph databases with XPath. In *Proceedings of the 16th International Conference on Database Theory*. 129–140.

[78] Leonid Libkin, Juan L Reutter, Adrián Soto, and Domagoj Vrgoč. 2018. TriAL: A navigational algebra for RDF triplestores. *ACM Transactions on Database Systems (TODS)* 43, 1 (2018), 1–46.

[79] Jing Lu, Li Ma, Lei Zhang, Jean-Sébastien Brunner, Chen Wang, Yue Pan, and Yong Yu. 2007. SOR: A Practical System for Ontology Storage, Reasoning and Search.. In *VLDB*, Vol. 7. 1402–1405.

[80] Amélie Marian and Jérôme Siméon. 2003. Projecting XML documents. In *Proceedings 2003 VLDB Conference*. Elsevier, 213–224.

[81] Wim Martens. 2022. Towards theory for real-world data. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 261–276.

[82] Michael Meier, Michael Schmidt, Fang Wei, and Georg Lausen. 2010. Semantic query optimization in the presence of types. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 111–122.

[83] Marios Meimaris, George Papastefanatos, Nikos Mamoulis, and Ioannis Anagnostopoulos. 2017. Extended characteristic sets: graph indexing for SPARQL query optimization. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 497–508.

[84] Amine Mhedhbi, Matteo Lissandrini, Laurens Kuiper, Jack Waudby, and Gábor Szárnyas. 2021. LSQB: a large-scale subgraph query benchmark. In *Proceedings of the 4th ACM SIGMOD Joint International Workshop on Graph Data Management*

Experiences & Systems (GRADES) and Network Data Analytics (NDA). 1–11.
[85] Gerome Miklau and Dan Suciu. 2004. Containment and equivalence for a fragment of XPath. *Journal of the ACM (JACM)* 51, 1 (2004), 2–45.
[86] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 984–994.
[87] Frank Neven and Thomas Schwentick. 2006. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science* 2 (2006).
[88] Van-Quyet Nguyen and Kyungbaek Kim. 2017. Estimating the evaluation cost of regular path queries on large graphs. In *Proceedings of the 8th International Symposium on Information and Communication Technology*. 92–99.
[89] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2006. Semantics and Complexity of SPARQL. In *International semantic web conference*. Springer, 30–43.
[90] Mohamed Ragab. 2020. Large Scale Querying and Processing for Property Graphs PhD Symposium. (2020).
[91] Juan L Reutter, Miguel Romero, and Moshe Y Vardi. 2017. Regular queries on graph databases. *Theory of computing Systems* 61 (2017), 31–83.
[92] Christopher Rost, Riccardo Tommasini, Angela Bonifati, Emanuele Della Valle, Erhard Rahm, Keith W Hare, Stefan Plantikow, Petra Selmer, and Hannes Voigt. 2024. Seraph: Continuous Queries on Property Graph Streams. (2024).
[93] Sherif Sakr, Angela Bonifati, Hannes Voigt, and Alexandru Iosup. 2021. Ensuring the success of big graph processing for the next decade and beyond. *Commun. ACM* 64, 9 (2021).
[94] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A Boncz, et al. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71.
[95] Max Schäfer and Oege de Moor. 2010. Type inference for datalog with complex type hierarchies. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 145–156.
[96] J Shanmugasundaram, K Tufte, C Zhang, G He, DJ DeWitt, and JF Naughton. 1999. Relational Databases for Querying XML Documents: Limitations and Opportunities, in 'VLDB'99: Proc. of the 25th Int. Conf. on Very Large Data Bases'.
[97] Chandan Sharma and Roopak Sinha. 2022. FLASc: a formal algebra for labeled property graph schema. *Automated Software Engineering* 29, 1 (2022), 37.
[98] Chandan Sharma, Roopak Sinha, and Kenneth Johnson. 2021. Practical and comprehensive formalisms for modelling contemporary graph query languages. *Information Systems* 102 (2021), 101816.
[99] Amit Sheth, Boanerges Aleman-Meza, I Budak Arpinar, Clemens Bertram, Yashodhan Warke, Cartic Ramakrishanan, Chris Halaschek, Kemafar Anyanwu, David Avant, F Sena Arpinar, et al. 2005. Semantic association identification and knowledge discovery for national security applications. *Journal of Database Management (JDM)* 16, 1 (2005), 33–53.
[100] Fabian Suchanek, Mehwish Alam, Thomas Bonald, Pierre-Henri Paris, and Jules Soria. 2023. Integrating the Wikidata Taxonomy into YAGO. arXiv:2308.11884 [cs.AI]
[101] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. Sqlgraph: An efficient relational-based property graph store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1887–1901.
[102] Gábor Szárnyas. 2023. LDBC Social Network Benchmark graphs. https://hdl.handle.net/11112/e6e00558-a2c3-9214-473e-04a16de09bf8. https://doi.org/10.25606/SURF.8f3ac424d6694282
[103] Jacopo Urbani, Ceriel JH Jacobs, and Markus Krötzsch. 2016. VLog: A Column-Oriented Datalog System for Large Knowledge Graphs.. In *ISWC (Posters & Demos)*.
[104] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. 1–6.
[105] Laurent Vielle. 1989. Recursive query processing: The power of logic. *Theoretical computer science* 69, 1 (1989), 1–53.
[106] Domagoj Vrgoc. 2014. *Querying graphs with data*. Ph. D. Dissertation. University of Edinburgh, UK. https://hdl.handle.net/1842/8953
[107] Kemas Wiharja, Jeff Z Pan, Martin J Kollingbaum, and Yu Deng. 2020. Schema aware iterative Knowledge Graph completion. *Journal of Web Semantics* 65 (2020), 100616.
[108] Peter T Wood. 2003. Containment for XPath fragments under DTD constraints. In *Database Theory—ICDT 2003: 9th International Conference Siena, Italy, January 8–10, 2003 Proceedings 9*. Springer, 300–314.
[109] Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol Deshpande. 2017. Graphgen: Adaptive graph processing using relational databases. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. 1–7.
[110] YAGO Yago. 2019. A high-quality knowledge base. https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago.
[111] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2015. WAVEGUIDE: Evaluating SPARQL Property Path Queries.. In *EDBT*, Vol. 2015. 525–528.
[112] Fang Yang, Kunjie Fan, Dandan Song, and Huakang Lin. 2020. Graph-based prediction of protein-protein interactions with attributed signed graph embedding. *BMC bioinformatics* 21, 1 (2020), 1–16.