Graph Queries: From Theory to Practice

Angela Bonifati, Stefania Dumbrava

Graph Queries: From Theory to Practice

Angela Bonifati
University of Lyon 1 & CNRS LIRIS
{angela.bonifati}@univ-lyon1.fr

Stefania Dumbrava*
ENS Rennes & CNRS IRISA & INRIA
{stefania-gabriela.dumbrava}@ens-rennes.fr

ABSTRACT

We review various graph query language fragments that are both theoretically tractable and practically relevant. We focus on the most expressive one that retains these properties and use it as a stepping stone to examine the underpinnings of graph query evaluation along graph view maintenance. Further broadening the scope of the discussion, we then consider alternative processing techniques for graph queries, based on graph summarization and path query learning. We conclude by pinpointing the open research directions in this emerging area.

1. INTRODUCTION

Graphs are semantically rich data models able to represent inherently complex object structures and their interconnectivity relationships. Due to their high expressivity, graphs are used in numerous domains, including Knowledge Representation and the Semantic Web, Linked Open Data, geolocation data, as well as life science repositories, such as those used in medicine, biology, and chemistry. Several graph datasets, such as DBPedia [11], Wikidata [40], and Bio2RDF [34], to name a few, are readily available and exhibit a continuous growth, as new user content is injected on a daily basis. Hence, such massive, graph-shaped data have tremendous potential to be queried and explored for knowledge extraction purposes [17].

In this paper, which summarizes our previous work in this area [16, 13, 23], we survey established theoretical graph query foundations and discuss their practical impact. To this end, we examine graph query evaluation and incremental maintenance techniques, along with implementation aspects. We also review alternative graph query processing techniques, such as approximate query evaluation and path query learning.

*Work mainly done while affiliated with University of Lyon 1 & CNRS LIRIS.

As opposed to their relational counterparts, graph queries are recursive in nature and need to inspect both the structure and the heterogeneity of the underlying data. We illustrate this aspect with the following user-specified query that we have taken from the online Wikidata query set, formulated by real users at the Wikidata SPARQL query service [1]. The query outputs the geolocation information of all hospitals in the Wikipedia ontology at a world-wide scale:

```
Q1: SELECT * WHERE {
        ?item wdt:P31*/wdt:P279* wd:Q16917;
              wdt:P625 ?geo .
}
```

Note that wd:Q16917 is a hospital item, wdt:P31 and wdt:P279 are the "instance of" and "subclass of" Wikidata properties, while wdt:P625 is a coordinate location property. Such a query relies on a recursive expression of the kind a^*/b^*, which drives the navigation of the Wikipedia ontology to find all possible occurrences of hospitals, as item instances or subclasses. More precisely, the above query retrieves a set of geolocation data points that represent the positions of hospitals in a map. As such, its result represents *a graph* of interconnected hospital locations. Concerning the language fragment to which this query belongs, we can classify it as a Conjunctive Regular Path query, belonging to C2RPQ, a notable query fragment that we discuss in Section 2. Due to the presence of recursion, such a query performs complex navigation on the Wikidata graph. An alternative example of a graph query is the following Wikidata one, which retrieves a single aggregate value, namely the total number of humans in Wikidata [2].

```
Q2: SELECT (COUNT(?item) AS ?count)
WHERE {
        ?item wdt:P31/wdt:P279* wd:Q5 .
}
```

[1] https://www.wikidata.org/wiki/Wikidata:SPARQL_query_service/queries/examples

[2] Amounting to 4531233 (on September 25, 2018).

In such a case, a property path of the type a/b^* allows to navigate the Wikidata ontology.

While the first query belongs to C2RPQ, the second query is a counting regular path query (a fragment henceforth named RPQ_C). Even though these fragments have remarkable differences in terms of their evaluation complexity (see Section 3), they are also significantly different in terms of their retrieved output. In fact, while the result of the first query is a geographical network, the second query retrieves a *semantically rich aggregate value*.

This wide array of possibilities, in terms of query input and output, along with the complexity of query evaluation, drives and motivates our current research in the area [23, 16, 22, 18, 8, 15, 7, 13, 12, 9], part of which we revisit in this paper.

We start by describing the *actual* expressivity and computational complexity of practical graph query fragments, as used in various modern graph query languages [3], and focus on how to efficiently process them. We also expand on ensuring the reliability of potentially security-critical applications that can leverage queries in the above languages. To this end, we illustrate in [16] the feasibility of employing formal methods to formalize the expressive regular query (RQ) language and to mechanize the implementation of a corresponding *inference and view maintenance* engine, whose correct behavior we establish through machine-checked proofs.

Hence, we turn to RPQs and study approximate query processing (AQP), which gives the users the agency to decide the tradeoff w.r.t efficiency for query fragments that are expensive to process. In particular, in [23], we investigate the effectiveness of storing pre-computed aggregates to approximate the result of RPQ_C queries, which have a high runtime evaluation cost. To this purpose, we illustrate a query-driven summarization algorithm that we introduced. As we will outline in Section 4, we tackle reachability preservation, in the presence of aggregates, with the explicit purpose of obtaining a small, *reusable graph summary* that can lend itself easily to in-database approximate evaluation.

Next, we illustrate *path query learning* as an alternative processing technique for graph queries [13]. Still on the RPQ fragment, we show how to infer a query statement from a set of positive and negative examples, the latter embodying the expected (or not) query results. As the consistency checking problem for RPQ queries is PSPACE-complete, we resort to lifting the soundness condition of the learning algorithm and propose a learning algorithm that selects paths of a given length. Finally, we discuss an interactive scenario, which leads to a learning algorithm that starts with an empty sample and continuously interacts with the user, in order to infer the goal query.

The paper is organized as follows. We start in Section 2 with an outline of the underlying graph data models and the fundamental graph query fragments that have been studied in the literature and identified as retaining practical interest. Section 3 expands on the complexity of query and incremental view evaluation and illustrates both evaluations for a highly expressive query fragment. Section 4 describes approximate analytical processing and path query learning. Finally, we conclude in Section 5, by highlighting open problems in this area and by providing future research directions.

2. PRELIMINARIES

Graph Database Models. Graph databases rely on *nodes*, to denote abstract entities, and on *edges*, to denote the relationships between them. Such is the structure of the basic edge-labeled model, which we consider in Section 3. This can be further enhanced, to account for *direction*, by taking edges to be ordered pairs of vertices, for *heterogeneity*, by allowing multiple edges between a given pair of vertices, as well as multiple labels, on both vertices and edges, and for *data storage*, by allowing an arbitrary number of properties, or key/value pairs, to be attached to both vertices and edges. Considering these extensions, we reach the expressivity level of the *property graph model* (PGM) [2, 17], on which we focus in Section 4 and which we define next.

Given a finite sets of symbols (*labels*) Σ, *property keys* $\mathcal{K}$, and *property values* $\mathcal{N}$, a *property graph instance* $\mathcal{G}$ over (Σ, $\mathcal{K}$, $\mathcal{N}$), is a structure $(\mathcal{V}, E, \eta, \lambda, \nu)$, such that $\mathcal{V}$ and E are finite sets of *vertex/edge* identifiers, $\eta : E \to \mathcal{V} \times \mathcal{V}$ associates a pair of vertex identifiers to each edge identifier, $\lambda : \mathcal{V} \cup E \to \mathcal{P}(\Sigma)$ [3] associates a set of labels to vertex/edges, and $\nu : (\mathcal{V} \cup E) \times \mathcal{K} \to \mathcal{N}$, associates a value to each vertex/edge property key.

Example 1. *We exemplify the base, edge-labeled model with the graph instance $\mathcal{G}_{SN}$ in Fig. 1, which represents a social network, whose schema is inspired by the LDBC benchmark [24]. Entities are customers (type* `Person`*, C_i),* connected *(l_0) and/or* following *(l_1) each other, that can purchase (l_4) merchandise (type* `Product`*, M_i). This is* promoted *(l_5) in ads (type* `Message`*, A_i), which are* posted *(l_3) on brand pages (type* `Forum`*, P_i),* moderated *(l_2) by specific persons. Additionally, customers can* endorse *(l_6) each other or* endorse *a brand.*

[3] i.e., $\mathcal{P}(\Sigma)$ denotes the set of finite subsets of Σ.

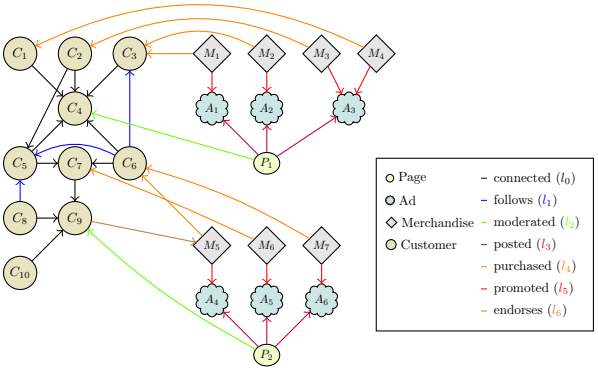


Figure 1: Example Social Graph $\mathcal{G}_{SN}$

Fundamental Query Fragments. *Regular expressions* over a finite alphabet Σ are defined as $e ::= \epsilon \mid s$, with $s \in \Sigma \mid e + e \mid e \cdot e \mid e^*$. A *regular language* $\mathcal{L}(\Sigma)$ of complex labels can thus be inductively built: $\mathcal{L}(\epsilon) = \{\epsilon\}$; $\mathcal{L}(s) = \{s|\ s \in \Sigma\}$; $\mathcal{L}(e_1 + e_2) = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$; $\mathcal{L}(e_1 \cdot e_2) = \mathcal{L}(e_1) \cdot \mathcal{L}(e_2)$; $\mathcal{L}(e^*) = \{e_1 \cdot e_2 \mid e_1 \in \mathcal{L}(e) \wedge e_2 \in \mathcal{L}(e^*)\}$. Next, given $\mathcal{V}$, a countably infinite set of variables, and $\mathcal{D}$, a domain of constant values, *terms* are elements of $\mathcal{V} \cup \mathcal{D}$. Based on these building blocks, the following prominent *query* fragments have emerged: 1) **regular path queries**, $RPQ = \{s(t_1, t_2) | s \in \mathcal{L}(\Sigma)\}$, 2) **counting** RPQ, $RPQ_C = \{s(t_1, t_2) | s \in \mathcal{L}(\Sigma \cup \{count\})\}$, 3) **2-way** $RPQs$, which allow backward navigation, $2RPQ = \{s(t_1, t_2) | s \in \mathcal{L}(\Sigma \cup \{s^- | s \in \Sigma\})\}$, 4) **conjunctive** $2RPQ$, $C2RPQ = \{\bigwedge_{i \in \mathbb{N}} s_i(t_1, t_2) | s_i \in 2RPQ\}$, 5) **union of** $C2RPQ$, $UC2RPQ = \{\bigvee_{i \in \mathbb{N}} s_i(t_1, t_2) | s_i \in C2RPQ\}$, 6) **nested** $UC2RPQ$, $nUC2RPQ = \{\bigvee_{i \in \mathbb{N}} s_i(t_1, t_2) | s_i \in \{s^* | s \in UC2RPQ\}\}$, 7) **union of conjunctive, nested** $2RPQ$, $UCN2RPQ = \{\bigvee_{i \in \mathbb{N}} \bigwedge_{j \in \mathbb{N}} s_{i,j}(t_1, t_2) | s_{i,j} \in \{s^* | s \in 2RPQ\}\}$, and the recent 8) **regular queries**, $RQ = \{s(t_1, t_2) | s \in \{s^* | s \in UCN2RPQ\}\}$.

The most expressive graph query fragment, RQ (regular queries) [39, 36], is an extension of unions of conjunctive 2-way regular path queries (UC2RPQs) and of unions of conjunctive nested 2-way regular path queries (UCN2RPQs). Regular queries support expressing complex regular patterns between graph nodes. They also correspond to Datalog with linear recursion [31], also known as non-recursive Datalog, extended, at the language-level, with transitive closures of binary predicates.

EXAMPLE 2. *Revisiting Example 1, we illustrate the above query fragments on the social graph $\mathcal{G}_{SN}$ from Figure 1:*

$$
\begin{aligned}
&Q_1 : \Omega_1(X, \mathcal{B}) \leftarrow (l_0 + l_1)^* \cdot l_6(X, \mathcal{B})\\
&Q_2 : \Omega_2(X, \mathcal{B}) \leftarrow l_4^- \cdot l_5 \cdot l_3^-(X, \mathcal{B})\\
&Q_3 : \Omega_3(C, \_) \leftarrow l_4^- \cdot l_5 \cdot l_3^-(X, \mathcal{B}), count(X, C)\\
&Q_4 : \Omega_4(X, \mathcal{B}) \leftarrow l_2^-(X, \mathcal{B}), l_6 \cdot l_5 \cdot l_3^-(X, \mathcal{B})\\
&Q_5 : \Omega_5(X, Y) \leftarrow l_4 \cdot \Omega_4{}^+ \cdot l_4(X, Y)\\
&Q_6 : \Omega_6(X, Y) \leftarrow l_3 \cdot l_5(\mathcal{B}, X), \Omega_5(X, Y), l_3 \cdot l_5(\mathcal{B}, Y)\\
&Q_7 : \Omega_7(X, Y) \leftarrow (l_6^- \cdot (l_0 + l_1)^+ + \Omega_6)(X, Y)\\
&Q_8 : \Omega_8(X, Y) \leftarrow (l_4^- \cdot \Omega_7 \cdot l_4)^+(X, Y)
\end{aligned}
$$

Consider $\mathcal{B}$ to be a brand in the social graph $\mathcal{G}_{SN}$. Ω_1 returns the customers connected, directly or indirectly, to an endorser*, while Ω_2 returns $\mathcal{B}$'s customers. Ω_3 counts the above, while Ω_4 returns $\mathcal{B}$'s* fans*, i.e., the customers that monitor a $\mathcal{B}$ page and endorse its merchandise. Ω_5 returns the products that are* viral*, i.e., purchased (or endorsed, for Ω_7) by connected brand fans (of brand $\mathcal{B}$, for Ω_6). Ω_8 returns all consumers that purchase viral $\mathcal{B}$ products. In terms of query expressivity, the following memberships hold: $Q_1 \in RPQ, Q_2 \in 2RPQ, Q_3 \in RPQ_c, Q_4 \in C2RPQ, Q_5 \in UC2RPQ, Q_6 \in nUC2RPQ, Q_7 \in UCN2RPQ, Q_8 \in RQ$.*

3. GRAPH QUERY PROCESSING

Our discussion on the evaluation and processing of graph queries begins in Section 3.1, with an brief overview of the respective complexity of the various query classes we discuss. Narrowing down on Regular Queries, which represent the most expressive fragment, we then proceed, in Section 3.2, to presenting the design of a *custom RQ evaluation and incremental maintenance algorithm*. In Section 3.3, we provide insights concerning its corresponding implementation and formal development, carried out with the Coq proof assistant [37]. To the best of our knowledge, this work constitutes the first *certified specification and implementation* of the RQ language and of its mechanized processing engine.

3.1 Complexity Results

In Table 1, we summarize the main complexity results [36, 39] regarding the evaluation and containment of the graph query classes in Section 2. Note that, at a foundational level, these fragments rely on *conjunctive queries* (CQ), whose evaluation is polynomial and whose containment is NP-complete [19]. Their common denominator is that they perform *edge traversals* (through join chains), while specifying and checking the existence of *constrained paths*.

The most expressive class we consider is RQ (*Regular Queries*) [36], already described in Section 2. Unlike full Datalog [1], with P-complete evaluation and undecidable containment, Regular Datalog is particularly well-behaved. First, its evaluation

$$
\begin{array}{lcll}
t & ::= & n \in \mathcal{D} \mid x \in \mathcal{V} & \text{(Term)} \\
A & ::= & s(t_1, t_2), s \in \Sigma \mid t_1 = t_2 & \text{(Atom)} \\
L & ::= & A \mid A^+ & \text{(Literal)} \\
B & ::= & L_1 \wedge \ldots \wedge L_n & \text{(Conj. Body)} \\
D & ::= & B_1 \vee \ldots \vee B_n & \text{(Disj. Body)} \\
C & ::= & (t_1, t_2) \leftarrow D & \text{(Clause)} \\
\Pi & ::= & \Sigma \rightarrow \{C_1, \ldots, C_n\} & \text{(Program)}
\end{array}
$$

Figure 2: Regular Datalog Grammar

has NLOGSPACE-complete data complexity and is hence included in NC, the class of *highly parallelizable* problems. Second, the containment of Regular Datalog queries is *decidable*, with an elementary tight bound (2EXPSPACE-complete) [36].

The behaviors of the other fragments presented in Table 1 resemble each other, with two exceptions. The evaluation of counting label-constrained reachability queries RPQ_C has $\#P-$complete data complexity [38] and its containment problem is undefined. The containment problem for RPQ and 2RPQ is PSPACE-complete [6]. For the sake of conciseness, we omit here further details on the complexity of evaluation of special classes of RPQ, boiling down to the trichotomy in [9] and to simple transitive expressions in [33].

3.2 Evaluation and Maintenance

We present the theoretical foundations of an evaluation and incremental view maintenance engine for regular queries (RQ). We begin with a high-level description of the basic algorithm underpinning evaluation and then extend the introduced constructs to support *incremental maintenance*.

Evaluation. As a subset of Datalog, RQs can consequently lend themselves to the same evaluation techniques employed by deductive reasoning engines. The adopted evaluation strategies of these engines can be classified as: 1) *bottom-up*, i.e., start from the extensional database and generate new facts by forward-chaining, 2) *top-down*, i.e., start from the query (part of the intensional database) and construct a proof tree or a refutation proof by back-chaining, or 3) *rewriting-based*, i.e., transform the query into one for which bottom-up evaluation emulates top-down information passing (*magic-sets* [10]) or pushdown automata (*chain-queries* [28]). Henceforth, we focus on the bottom-up approach, and build on it in order to construct the first inference engine. Our choice is motivated by the desirable properties of bottom-up inference, such as guaranteed termination for finite models and amenability to formalization. Specifically, we rely on its fundamentally set-theoretical nature to specify the RQ engine's behavior and to construct its machine-checked soundness proof in the Coq proof assistant.

To facilitate efficient mechanical reasoning, we represent RQ constructs as illustrated in Fig. 2. Notably, we formalize programs as mappings from indexing symbols to a *single pair* of source-target nodes and to a *normalized* disjunctive body. The normalized form is obtained through a *completion* procedure, uniformizing clause heads and regrouping their respective bodies. For example, the program $s(a,b).\ s(z,y) \leftarrow p(x,y), q^+(z,x)$ is normalized as $s(x,y) \leftarrow (a = x \wedge b = y) \vee (p(z,y) \wedge q^+(x,z))$ and represented by a *function* from s to the head and disjunctive body. Based on this representation, we define an RQ over a graph $\mathcal{G}$ as a *stratified*, Regular Datalog program Π, along with a distinguished query clause, whose head is the top-level *view*. We illustrate this in Example 3, with $l_r(X,Y)$ as RQ.

Example 3. *In $\mathcal{G}_{SN}$, let $\mathcal{B}$ be a brand wanting to determine if a customer pair (l_c) is in the same advertising reachability cluster (l_r). We say that $\mathcal{B}$'s advertising reaches a customer X either: 1) directly (l_d), if X endorses (l_6) or purchases (l_4^-) merchandise promoted (l_5) in ads posted (l_3) on the brand's page, or 2) indirectly (l_i), if X is linked via a follower/connection chain to another customer that is under the direct reach of $\mathcal{B}$.*

$$
\begin{array}{lcl}
l_r(X,Y) & \leftarrow & {l_c}^+(X,Y) \\
l_c(X,Y) & \leftarrow & l_i(X,\mathcal{B}), l_i(Y,\mathcal{B}) \\
l_i(X,\mathcal{B}) & \leftarrow & (l_1 + l_0)^+ \cdot l_d(X,\mathcal{B}) \\
l_d(X,\mathcal{B}) & \leftarrow & (l_6 + l_4^-) \cdot l_5 \cdot l_3(X,\mathcal{B})
\end{array}
$$

Figure 3: Advertising Reachability Clusters

The semantics of Regular Datalog programs follows standard term-model definitions. For optimization purposes, we model *interpretations* $\mathcal{G}$ as indexed relations $(\Sigma \times \{\emptyset, +\}) \rightarrow \mathcal{P}(\mathcal{D} \times \mathcal{D})$, which contain labeled graphs and their transitive closure. Given that closures are thus internalized, we also impose that interpretations be well-formed, i.e., that the information stored in $\mathcal{G}(s,+)$ corresponds to the actual transitive closure of $\mathcal{G}(s,\emptyset)$. Hence, we check if, for every node pair $(n_1,n_2) \in \mathcal{G}(s,+)$, there exists a path (vertex sequence) that starts with n_1 and ends with n_2. Hence, a ground literal $s^m(n_1,n_2)$ is satisfied by $\mathcal{G}$ iff $(n_1,n_2) \in \mathcal{G}(s,m)$, with $m \in \{\emptyset,+\}$, the transitive closure marker. Consequently, a clause, with index s and disjunctive body $D \equiv (L_{1,1} \wedge \ldots \wedge L_{1,n}) \vee \ldots \vee (L_{m,1} \wedge \ldots \wedge L_{m,n})$, is satisfied by $\mathcal{G}$, $\mathcal{G} \models_s (t_1,t_2) \leftarrow D$, iff $\forall \eta, \bigvee_{i=1..m} (\bigwedge_{j=1..n} \mathcal{G} \models \eta(L_{i,j})) \Rightarrow \mathcal{G} \models$

Query Fragment	*Evaluation*	*Containment*
RPQ	NLOGSPACE-complete	PSPACE-complete
RPQ_C	$\#P-$complete	*Undefined*
2RPQ	NLOGSPACE-complete	PSPACE-complete
C2RPQ	NLOGSPACE-complete	EXPSPACE-complete
UC2RPQ	NLOGSPACE-complete	EXPSPACE-complete
nUC2RPQ	NLOGSPACE-complete	EXPSPACE-complete
UCN2RPQs	NLOGSPACE-complete	EXPSPACE-complete
RQ	NLOGSPACE-complete	2EXPSPACE-complete

Table 1: Evaluation and containment data complexity for the language fragments studied in this paper.

$\eta(s(t_1, t_2))$, i.e, for all substitutions η, which ground body literals, if an instantiated body disjunct is satisfied, then so is the instantiated head. The latter is indeed a *ground literal*, as we impose the safety condition that all head variables appear in the body. A well-formed interpretation $\mathcal{G}$ is thus a model for a program Π w.r.t Σ iff $\mathcal{G}$ satisfies all clauses indexed by Σ symbols, i.e., $\mathcal{G} \models_\Sigma \Pi$ iff $\forall\ s \in \Sigma, \mathcal{G} \models_s \Pi(s)$.

To compute models, we implement bottom-up RQ evaluation based on the *consequence operator* [1]. This relies on a generic matching algorithm that, for an initial interpretation and a clause construct, computes the set of all satisfying substitutions. For example, given $\mathcal{G}$ and a literal l, the matching function $M^B_\mathcal{G}(l)$ outputs all substitutions σ, such that $\mathcal{G} \models \sigma(l)$. For a clause, $\Pi(s) \equiv (t_1, t_2) \leftarrow \bigvee_{i=1..n} B_i$, it extends to body matching straightforwardly, with $M^B_\mathcal{G}(B_i)$ traversing B_i and collecting the set of substitutions obtained from the individual matching. Substitutions for each disjunctive clause are thus accumulated and the resulting ground heads, newly inferred facts, are added to the interpretation. The consequence operator, encoding nested-loop join, is expressed set-theoretically as $T^{\Pi,s}(\mathcal{G}) \equiv \{\sigma(t_1, t_2) \mid \sigma \in \bigcup_{i=1..n} M^B_\mathcal{G}(B_i)\}$.

Maintenance. Given updates Δ to a base graph $\mathcal{G}$, the above evaluation procedure non-incrementally maintains the top-level Π view, without reusing or adjusting the previously computed maintenance information. This makes it especially inefficient when few nodes are added to a high-cardinality graph.

To remedy this situation, we extend our previous algorithm to take into account information from previously computed models. The key idea is to restrict matching to graph updates in the spirit of incremental view maintenance for relational databases [27]. For example, let V be a materialized view, defined as the path over two base edges, r and s, i.e., $V(X, Y) \leftarrow r(X, Z), s(Z, Y)$. Notice that this path can also be seen as a join between the binary relations r and s on the Z variable, abbreviated as $V = r \bowtie s$. For *base deltas*, Δr and Δs, we can compute the *view delta* as $\Delta V = (\Delta r \bowtie s) \cup (r \bowtie \Delta s) \cup (\Delta r \bowtie \Delta s)$, or, factoring, as $\Delta V = (\Delta r \bowtie s) \cup (r^\nu \bowtie \Delta s)$, with $r^\nu = r \cup \Delta r$. Hence, $\Delta V = \Delta V_1 \cup \Delta V_2$, with ΔV_1 and ΔV_2 computable via the following delta clauses:

$$\delta_1 : \Delta V_1 \leftarrow \Delta r(X, Z), s(Z, Y)$$
$$\delta_2 : \Delta V_2 \leftarrow r^\nu(X, Z), \Delta s(Z, Y)$$

In general, for $V \leftarrow L_1, \ldots, L_n$ and an additive update Δ, we can determine the view delta $\Delta V[\mathcal{G}; \Delta]$ as the set of facts such that $V[\mathcal{G} :+: \Delta] = V[\mathcal{G}] \cup \Delta V[\mathcal{G}; \Delta]$. To this end, we compute the *delta program* $\delta(V) = \{\delta_i \mid i \in [1, n]\}$, where each *delta clause* δ_i is $V \leftarrow L_1, \ldots, L_{i-1}, \Delta L_i, L^\nu_{i+1}, \ldots, L^\nu_n$.

Note that L^ν_j marks that we match L_j against atoms in $\mathcal{G} \cup \Delta G$ with the same symbol as L_j and ΔL_j marks that we match L_j against atoms in $\Delta\mathcal{G}$ with the same symbol as L_j. We revisit the schema in Example 1, on a slightly different graph instance, to illustrate this *incremental view computation*.

Example 4. *Consider Figure 4a, in which entity Y is* monitored *(l_m) by X, if X is its connection/follower/moderator, and* auto-referrals *(l_{ar}) are computed as cyclic endorsements.*

$$\begin{array}{lcl} l_m(X, Y) & \leftarrow & (l_0 + l_1 + l_2)(X, Y) \\ l_{ar}(X, Y) & \leftarrow & l_6(X, Y), l_6(Y, X) \end{array}$$

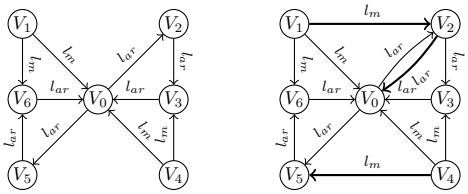
All detectable auto-referrals *are computable with RQ below, as* $[\![\Omega]\!]_\mathcal{G} = \{(V_6, V_0), (V_3, V_0))\}$

$$\Omega(X, Y) \leftarrow l_{ar}(X, Y), l_m(Z, X), l_m(Z, Y)$$

When updating *the previous graph in Figure 4b:* $[\![\Omega]\!]_{\mathcal{G}'} = \{(V_6, V_0), (V_3, V_0), (\mathbf{V_0}, \mathbf{V_2}), (\mathbf{V_2}, \mathbf{V_0}), (\mathbf{V_0}, \mathbf{V_5})\}$. *The delta update* $\Delta\Omega = \{(\mathbf{V_0}, \mathbf{V_2}), (\mathbf{V_2}, \mathbf{V_0}), (\mathbf{V_0}, \mathbf{V_5})\}$ *can be* incrementally *computed from* $\Pi_\Delta = \delta_1 \cup \delta_2 \cup \delta_3$, *as:* $\delta_1 = \emptyset$, $\delta_2 = \{(\mathbf{V_2}, \mathbf{V_0})\}$, $\delta_3 = \{(\mathbf{V_0}, \mathbf{V_2}), (\mathbf{V_0}, \mathbf{V_5})\}$.

3.3 Implementation and Certification

The implementation of the engine accounts for two modes of evaluation: *base* and *incremental*. Note that the former is still needed as, in some cases we identify, incremental evaluation is either not possible or not sensible, as full recomputation may be faster. The built-in engine heuristic is bottom-up and we leverage the non-recursive, stratified nature



(a) Initial Graph (b) Updated Graph

$$
\begin{array}{lrcl}
\delta_1: & \Delta\Omega(X,Y) & \leftarrow & \Delta\mathbf{l_{ar}}(X,Y), \mathbf{l_m}(Z,X), \mathbf{l_m}(Z,Y) \\
\delta_2: & \Delta\Omega(X,Y) & \leftarrow & \mathbf{l_{ar}}^{\nu}(X,Y), \Delta\mathbf{l_m}(Z,X), \mathbf{l_m}(Z,Y) \\
\delta_3: & \Delta\Omega(X,Y) & \leftarrow & \mathbf{l_{ar}}^{\nu}(X,Y), \mathbf{l_m}^{\nu}(Z,X), \Delta\mathbf{l_m}(Z,Y)
\end{array}
$$

(c) Delta Program for Detectable Auto-Referrals

Figure 4: Detectable Auto-Referrals

of the input programs to achieve single-pass, *fine-grained incremental* model computation. In the following, we outline the top-level engine interface and the main *soundness* theorem we formally prove.

Implementation. The *static* parameters of the engine are: a program Π, a graph $\mathcal{G}$, and a symbol set, or *support* supp, indicating the validity of a $\mathcal{G}$ subset, i.e., what information the incremental engine needs not recompute. Indeed, as we will see, a precondition of the engine is that the input graph is a model of Π *up to* supp, i.e., that $\mathcal{G} \models_{\mathsf{supp}} \Pi$. Note that in the database literature, the set of $\mathcal{G}$ symbols, $\mathbf{\Sigma}$, is often seen as a disjoint set pair, $(\mathbf{\Sigma}_E, \mathbf{\Sigma}_I)$, corresponding to the *extensional* and *intensional* program parts. For our engine, this distinction is "dynamic", as the already-processed strata-level is "extensional", or immutable, for the rest of the execution. Thus, typical cases for supp are $\mathsf{supp} \equiv \mathbf{\Sigma}_E$, when the engine has never been run before, or $\mathsf{supp} \equiv \mathbf{\Sigma}$, where $\mathcal{G}$ is the output of a previous run, and thus the consequences for all clauses have been computed. The *dynamic* parameters, capturing the current execution state, are: Δ, the current update, modified at each call, and the already and to-be processed strata, $\Sigma_{\triangleright}$ and $\Sigma_{\triangleleft}$.

Relying on an *incrementality-aware consequence operator*, $T^{\Pi,s}_{\mathcal{G},\mathsf{supp}}(\Delta)$, the engine iterates over $\Sigma_{\triangleleft}$ and, for each unprocessed symbol s, computes its corresponding closure. The algorithm then calls itself recursively, adding both s and s^+ to $\Sigma_{\triangleright}$.

Before discussing the implementation of $T^{\Pi,s}_{\mathcal{G},\mathsf{supp}}(\Delta)$, we explain the modifications made to *base* matching, in order to accommodate delta clauses and programs. Specifically, for each body to be processed incrementally, we generate a *mask*, B_Δ, by marking each of its literals with $m \in \{\mathbf{B}, \mathbf{D}, \mathbf{F}\}$. This indicates whether the engine should match against the base interpretation, the update, or both. We then define incremental atom matching as:

$$
\begin{aligned}
M^{A,m}_{\mathcal{G},\Delta}(a) \quad &= (\mathsf{if}\ m \in \{\mathbf{B},\mathbf{F}\}\ \mathsf{then}\ M^A_{\mathcal{G}}(a)\ \mathsf{else}\ \emptyset) \cup \\
&\quad\ (\mathsf{if}\ m \in \{\mathbf{D},\mathbf{F}\}\ \mathsf{then}\ M^A_{\Delta}(a)\ \mathsf{else}\ \emptyset)
\end{aligned}
$$

Incremental body matching, $M^B_{\mathcal{G},\Delta}$, proceeds as in Section 3.2, but additionally takes into account B_Δ, generated following the *diagonal factoring* below, where each row corresponds to a mask element:

$$
\begin{bmatrix}
{L_1}^{\mathbf{D}} & {L_2}^{\mathbf{F}} & \dots & {L_{n-1}}^{\mathbf{F}} & {L_n}^{\mathbf{F}} \\
{L_1}^{\mathbf{B}} & {L_2}^{\mathbf{D}} & \dots & {L_{n-1}}^{\mathbf{F}} & {L_n}^{\mathbf{F}} \\
\dots & \dots & \dots & \dots & \dots \\
{L_1}^{\mathbf{B}} & {L_2}^{\mathbf{B}} & \dots & {L_{n-1}}^{\mathbf{B}} & {L_n}^{\mathbf{D}}
\end{bmatrix}
$$

Finally, the last piece to complete the incremental engine is the top-level clausal maintenance operator $T^{\Pi,s}_{\mathcal{G},\mathsf{supp}}(\Delta)$ itself. This is more complex than its *base* counterpart, as it must take into account which incrementality heuristics to apply, distinguishing between two cases. If $s \notin \mathsf{supp}$, or Δ contains deletions for any of the literals in the body of $\Pi(s)$, it uses the base operator $T^{\Pi,s}(\mathcal{G} :+: \Delta)$, as we either cannot reuse the previous model or cannot support deletions through our incremental strategy. Otherwise, it generates a body mask, B_Δ, for each of the bodies B, and returns $\bigcup_{B_m \in B_\Delta} M^B_{\mathcal{G},\Delta}(B_m)$.

Certification. Before stating the key result with regard to the correct behavior of our engine, we mention the pre-conditions imposed. First, we require our input programs Π be *stratified*, i.e., that none of its head symbols depend on other that have not been previously defined. Second, as we reason about satisfaction *up to* a given symbol set, Σ, we say that Π is a well-formed slice of Σ, if, for every s in Σ, the symbols defining s in Π are contained in Σ. We establish that the engine operates over well-formed slices, which allows us to isolate reasoning about the current iteration. Finally, we formally prove that the incremental graph view maintenance engine is sound, as stated below.

THEOREM 1. *Let* Π *be a* safe, stratifiable, *Regular Datalog program;* Σ*, its symbols;* $\mathcal{G}$*, a graph instance;* Δ*, an update. The incremental view maintenance engine cumulatively processes each strata symbol, such that, if the already processed symbols,* $\Sigma_{\triangleright}$*, are a well-formed slice, if* Δ *only modifies* $\Sigma_{\triangleright}$*, and if the updated graph is a model of* Π *under* $\Sigma_{\triangleright}$*, then it outputs an* incremental update*, which, when applied to* $\mathcal{G}$*, forms a model of* Π *under* Σ*.*

The proof follows by structural induction on $\Sigma_{\triangleleft}$, relying on results we establish regarding modular satisfaction. These are paramount, as they allow us to reason about satisfaction locally, within each

well-formed slice. Note that the corresponding Coq proof of Theorem 1 is about 25 lines long and, thus, comparable to its paper-version. In total, the library we developed amounts to $\sim 1K$ lines of definitions, specifying our mechanized theory, and ~ 700 lines of proofs. Its compactness is mostly due to the fact that we rely on a library fine-tuned for the computer-aided theorem proving of finite-set theory results. This was built to carry out the mechanized proof of the Feit-Thompson theorem [26] on finite group classification. We leveraged the finite reasoning support, by giving a high-level, mathematical representation of the core engine components, as exemplified with the definition of the consequence operator in Section 3.2. This leads to *composable* lemmas that boil down to set-theoretic statements and, ultimately, to a condensed development, avoiding the proof-complexity explosion characteristic of formal verification efforts.

4. OTHER PROCESSING TECHNIQUES

We present alternative approaches that seek to mitigate the challenges posed by the evaluation of complex RPQ, as discussed in Section 3.1. First, in Section 4.1, we focus on leveraging the expressivity of the property graph model to develop efficient *approximate query evaluation* techniques for the RPQ_C fragment. Second, in Section 4.2, we highlight the promise shown by *path query learning* approaches, in the basic RPQ setting.

4.1 Query Approximation

In the following, we outline a newly introduced algorithm for graph summarization, and its application to the approximate evaluation of RPQ_C queries.

Graph Summarization. Sampling approaches, typically used for approximating relational queries, are not directly applicable to graph processing, due to the lack of the linearity assumption in graph-oriented data [30]. Indeed, the linear relationship between the sample size and execution time typical of relational query processing falls apart in graph query processing. For this reason, we focus on query-driven graph summarization as a baseline technique for untangling approximate graph query processing.

Our effort targets the *efficient*, *high-accuracy*, estimation of RPQ_C analytical queries, known to be costly in terms of runtime. We tackle both challenges, in an effort to achieve an optimal trade-off. First, we seek to obtain a *compact* (yet informative) summary, by explicitly inspecting the query workload and partitioning the graph according to the connectivity of the labels identified as most important. Second, we rely on the expressiveness of the *property graph* model to store *pertinent*, approximation-relevant, data, in the property lists of both nodes and edges. Specifically, these recorded statistics serve the purpose of preserving label-constrained reachability information.

Since both the original and summarized graphs adhere to the property graph data model (as presented in Section 2), the approximate evaluation can be done directly inside the graph database itself, thanks to a seamless query translation we provide.

We now focus on explaining and illustrating the underlying summarization algorithm. Let $\mathcal{G} = (\mathcal{V}, E)$ be a graph with edge labels $\Lambda(\mathcal{G})$. We introduce a summarization algorithm that compresses $\mathcal{G}$ to an AQP-amenable property graph, $\hat{\mathcal{G}}$, tailored for counting label-constrained reachability queries, with labels in Λ_Q, where $\Lambda_Q \subseteq \Lambda(\mathcal{G})$.

The summarization algorithm consists of the following three phases. First, the grouping phase computes Φ, a label-driven partitioning of $\mathcal{G}$ into *groupings*, following the label connectivity on the most frequent labels in $\Lambda(\mathcal{G})$. Next, the evaluation phase refines the previous step, further isolating into *supernodes* the grouping components that satisfy a custom property concerning label-connectivity. The merge phase then coalesces supernodes into *hypernodes*, based on label-reachability similarity conditions, as specified the heuristic mode m.

The *grouping phase* returns a partitioning Φ of $\mathcal{G}$, such that $|\Phi|$ is *minimized* and, for each $\mathcal{G}_i \in \Phi$, the number of occurrences of the most frequent edge label in $\Lambda(\mathcal{G}_i)$, $\max_{l \in \Lambda(\mathcal{G}_i)} (\#l)$, is *maximized*. Hence, we first sort the edge label set $\Lambda(\mathcal{G})$ into a *frequency list*, $\overrightarrow{\Lambda(\mathcal{G})}$. For each $l_i \in \overrightarrow{\Lambda(\mathcal{G})}$, in descending frequency order, we identify the largest $\mathcal{G}$-subgraphs that are weakly-connected on l_i.

EXAMPLE 5. *Let $\mathcal{G}$ be the graph from Figure 1. It holds that: $\#l_0 = 11$, $\#l_1 = 3$, $\#l_2 = 2$, $\#l_3 = 6$, $\#l_4 = 7$, $\#l_5 = 7$, $\#l_6 = 1$. Hence, we can take $\overrightarrow{\Lambda(\mathcal{G})} = [l_0, l_5, l_4, l_3, l_1, l_2, l_6]$. Note that, as $\#l_4 = \#l_5$, we can choose an arbitrary order for the labels in $\overrightarrow{\Lambda(\mathcal{G})}$. We first add $\mathcal{G}_1$ to Φ, as it regroups the maximal weakly-label components on l_0. Hence, $\mathcal{V} = \{R_1 - R_7, M_1 - M_6, F_1, F_2\}$. Next, we add $\mathcal{G}_2$ to Φ, as it regroups the maximally weakly-label component on l_5. We obtain $\mathcal{V} = \{F_1, F_2\}$ and $\Phi = \{\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3\}$, as shown in Figure 5a.*

The *evaluation phase* takes as input Φ, the previously obtained $\mathcal{G}$-partitioning, together with Λ_Q, and outputs $\mathcal{G}^* = (\mathcal{V}^*, E^*)$, an AQP-amenable com-

pression of $\mathcal{G}$. The phase computes $\mathcal{V}^*$, the set of *supernodes* (SN), and E^*, the set of *superedges* (SE). After each step, $\mathcal{G}^*$ is enriched with AQP-relevant properties, such as: $\mathcal{V}\,Weight$ and $EWeight$, the number of inner vertices and edges; *LPercent*, the percentage-wise label occurrence; and *LReach*, the number of vertex pairs connected by an edge with a given label. We also record pairwise label-traversal information, such as: *EReach*, the number of paths between two cross-edges with given labels, directions, and common node; and δ, the number of *traversal edges*, i.e., inner/cross-edge pairs, with given labels, directions, and common endpoint. Finally, we compute $\mathcal{V}_F$, the number of *frontier vertices*, given a fixed label and direction, as well as δ, the *relative label participation*, i.e., the number of cross-edges on a given label, relative to that of frontier vertices on another label.

The *merge phase* takes as input the $\mathcal{G}^*$ graph and Λ_Q and outputs a compressed graph, $\hat{\mathcal{G}} = (\hat{\mathcal{V}}, \hat{E})$. The phase proceeds in two steps, corresponding to the creation of $\hat{\mathcal{V}}$, the set of *hypernodes* (HN), and, respectively, to that of $\hat{E}$, the set of *hyperedges* (HE). HNs are computed by merging together supernodes based on two criteria. The primary, *inner-merge*, condition for candidate supernodes requires them to be maximal weakly label-connected on the same label. The *source-merge* heuristic additionally requires that they share the same set of outgoing labels, while the *target-merge* heuristic requires that they share the same set of ingoing labels. HEs are obtained by merging superedges that share the same label and endpoints. Finally, $\hat{\mathcal{G}}$ is enriched with previous AQP-relevant properties, with the addition of $\mathcal{V}^*Weight$, the average SN weight in each HN.

The three phases of the summarization algorithm are illustrated on our running example in Figure 5.

Optimal Summarization: NP-Completeness. We prove the intractability of the optimal graph summarization problem, under the conditions of our algorithm. Specifically, let $\mathcal{G} = (\mathcal{V}, E)$ and $\Phi = \{\mathcal{G}_i = (\mathcal{V}_i, E_i) \mid i \in [1, |\mathcal{V}|]\}$, a $\mathcal{G}$-partitioning. Each HN in $\mathcal{G}_i \in \Phi$ contains HN-subgraphs, $\mathcal{G}_i^k$, that are all *maximal weakly label-connected* on a label $l \in \Lambda(\mathcal{G})$. A *summarization function* $\chi_\Lambda : \mathcal{V} \to \mathbb{N}$ assigns to each vertex, v, a unique HN identifier $\chi_\Lambda(v) \in [1, k]$. χ_Λ is *valid*, if, for any v_1, v_2, where $\chi_\Lambda(v_1) = \chi_\Lambda(v_2)$, v_1, v_2 are in the following cases.
Case 1: part of the same HN-subgraph, $\mathcal{G}_i^k$, that is maximal weak label-connected on l.
Case 2: part of different HN-subgraphs, $\mathcal{G}_i^{k_1}$, $\mathcal{G}_i^{k_2}$, each maximal label-connected on l and not connected by an l-labeled edge in $\mathcal{G}$.

THEOREM 2. *Let* MinSummary *be the problem that, for a graph $\mathcal{G}$ and an integer $k' \geq 2$, decides if there exists a label-driven partitioning Φ of $\mathcal{G}$, $|\Phi| \leq k'$, such that χ_Λ is a* valid summarization. *MinSummary is NP-complete, even for undirected graphs, $|\Lambda(\mathcal{G})| \leq 2$ and $k' = 2$.*

Approximate Query Evaluation. For a graph $\mathcal{G}$ and a counting reachability query Q, we approximate the result $[\![Q]\!]_\mathcal{G}$ of evaluating Q over $\mathcal{G}$. Hence, we translate Q into a query Q^T, evaluated over the summarization $\hat{\mathcal{G}}$ of $\mathcal{G}$, such that $[\![Q^T]\!]_{\hat{\mathcal{G}}} \approx [\![Q]\!]_\mathcal{G}$, as discussed next.

Simple and Optional Label Queries. There are two configurations in which a label l can occur in $\hat{\mathcal{G}}$: either within a HN or on a cross-edge. Thus, we either cumulate the number of l-labeled HN inner-edges or the l-labeled cross-edge weights. To account for the potential absence of l, we also estimate, in the optional-label queries, the number of nodes in $\hat{\mathcal{G}}$, by cumulating those in each HN.

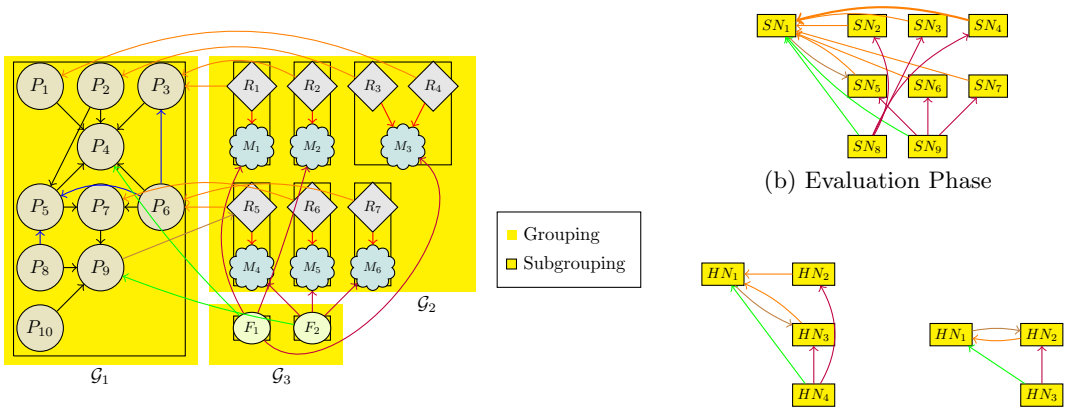
Kleene Plus and Kleene Star Queries. To estimate l^+, we cumulate the counts within HNs containing l-labeled inner-edges and, as above, the weights on l-labeled cross-edges. For the first part, we use the statistics gathered during the *evaluation phase*. We distinguish three scenarios, depending on whether the l_+ reachability is due to: 1) inner-edge connectivity – hence, we use the corresponding property counting the inner l-paths; 2) incoming cross-edges – hence, we cumulate the l-labeled in-degrees of HN vertices; or 3) outgoing cross-edges – hence, we cumulate the number of outgoing l-paths. To handle the ϵ-label in l^*, we additionally estimate, as before, the number of nodes in $\hat{\mathcal{G}}$.

Disjunction. We treat each possible configuration, on both labels. Hence, depending on each case, we cumulate the number of HN inner-edges, on either label, or the cross-edge weights with either label.

Binary Conjunction. We consider all cases, depending on whether: 1) the concatenation label $l_1 \cdot l_2$ appears on a path *within* a HN, 2) one of the labels l_1, l_2 occurs on a HN inner-edge and the other, as a cross-edge, or 3) both labels occur on cross-edges.

EXAMPLE 6. *We evaluate the AQP-translation of example queries of each of the types mentioned above:*

$$[\![l_5]\!]_{\hat{\mathcal{G}}} = Q_L^T(l_5) = \sum_{\hat{v} \in \hat{\mathcal{V}}} EWeight(\hat{v}, l_5) * LPercent(\hat{v}, l_5)$$
$$= EWeight(HN_2, l_5) * LPercent(HN_2, l_5) = 7$$
$$[\![l_2?]\!]_{\hat{\mathcal{G}}} = Q_L^T(l_2) + \sum_{\hat{v} \in \hat{\mathcal{V}}} \mathcal{V}^*Weight(\hat{v}) * \mathcal{V}Weight(\hat{v}) = 27$$
$$[\![l_0^+]\!]_{\hat{\mathcal{G}}} = \sum_{\hat{v} \in \hat{V}} LReach(\hat{v}, l_0) + \sum_{\hat{e} \in \hat{E}} EWeight(\hat{e}, l_0) = 15$$
$$[\![l_0^*]\!]_{\hat{\mathcal{G}}} = [\![l_0^+]\!]_{\hat{\mathcal{G}}} + \sum_{\hat{v} \in \hat{\mathcal{V}}} \mathcal{V}^*Weight(\hat{v}) * \mathcal{V}Weight(\hat{v}) = 40$$
$$[\![l_4 + l_1]\!]_{\hat{\mathcal{G}}} = [\![l_4]\!]_{\hat{\mathcal{G}}} + [\![l_1]\!]_{\hat{\mathcal{G}}} = 14 \text{ and } [\![l_4^- \cdot l_1]\!]_{\hat{\mathcal{G}}} = 7.$$



(a) Grouping Phase

(b) Evaluation Phase

(c) Source and Target Merge Phases

Figure 5: Summarization Phases for $\mathcal{G}_{SN}$

4.2 Query Learning

The problem of learning a regular path query $q \in$ RPQ consists of deriving a query statement from a set of user examples, specified under the form of positive and negative labels on the nodes of an input graph instance. A positive example is thus a positively labeled node n_+ in the input graph $\mathcal{G}$ if n should be present in the query result, while a negative example n_- is the opposite. We denote by S_+ the set of positive examples and by S_- the set of negative ones and their union by $S = S_+ \cup S_-$. In this work, we exemplify the query to be learned as an automaton. Regular languages can alternatively be represented by automata. We refer to [28] for standard definitions of *nondeterministic finite word automaton* (*NFA*) and *deterministic finite word automaton* (*DFA*). We rely on a Gold-style learning algorithm [25], thus on the standard framework of *language identification in the limit*. We aim at a polynomial query learning algorithm that is at the same time sound and complete. Soundness means that given the set of positive and negative examples, the algorithm will correctly return a consistent query with respect to the input positive and negative examples. The algorithm should also be complete, in the sense that it should be capable of learning any query from the set of input examples. It is easy to see that soundness is difficult to achieve, due to the intractability of consistency checking [13, 12], which is PSPACE-complete for queries in RPQ and NP-complete for concatenations of symbols.

Example 7. *Consistency checking for a query of the kind* $Q(l_4, l_5) \equiv l_4 \cdot l_5(\_,\_)$ *corresponding to the* direct reach *of a company via its page ads (i.e., the node pairs of customers and product advertisements in Figure 1) is already NP-complete.*

Due to this intractability, in our work [13, 12] we lifted the soundness condition of the algorithm and resorted to query learning with an abstain condition. If a consistent query cannot be efficiently found, the algorithm abstains from answering. The learning model with abstain is guaranteed to return, in polynomial time, either a consistent query or a null value, if such a query cannot be found. In particular, if a polynomial characteristic sample is provided, the learning algorithm is guaranteed to return the goal query.

The learning algorithm works by selecting the *smallest consistent paths* (SCPs) of length bounded by k (in order to avoid the enumeration of infinite paths). It then generalizes the SCP by states merge on the automaton [35].

The learnability of our query class corresponding to RPQ$^{\leq n}$ (denoting the RPQ of size at most n) is stated by the following result.

Theorem 3. *The query class* RPQ$^{\leq n}$ *is learnable with abstain in polynomial time and data, using the algorithm learner with the parameter* k *set to* $2 \times n + 1$.

In order to illustrate the underpinnings of our learning algorithm, we define the notion of a *consistent path* as follows. A path is consistent if it can be selected by the algorithm, for each positive node, and it does not cover any negative node. One can enumerate consistent paths (according to the canonical order $\leq$) by identifying the paths of each node labeled as positive and stopping when a consistent path is found, for each node. We refer to the obtained set of paths as the set of *smallest consistent paths* (SCPs).

EXAMPLE 8. *For example, given the graph in Figure 1 and a sample s.t.* $S_+ = \{C_2, C_{10}\}$ *and* $S_- = \{M_5\}$*, we obtain the SCPs* $l_0 \cdot l_6$*, for* C_{10}*, and, respectively,* $l_0 \cdot l_0 \cdot l_0 \cdot l_6$*, for* C_2*.*

Notice that in this case the disjunction of the SCPs (i.e., the query $l_0 \cdot l_6 + l_0 \cdot l_0 \cdot l_0 \cdot l_6$) is consistent with the input sample and one may think that a learning algorithm should return such a query. The shortcoming of such an approach is that the learned query would be always very simple, in the sense that it uses only concatenation and disjunction. Since we want a learning algorithm that covers all the expressibility of RPQ (in particular including the Kleene star), we need to extend the algorithm with a further step, namely the generalization. The PTA (Prefix Tree Acceptor) [21] of the previous SCPs can be constructed and its states are tentatively merged, if the obtained DFA does not select any negative node. In our example, it is easy to see that the generalized path expression should correspond to $l_0^+ \cdot l_6$.

Although Theorem 3 provides a theoretical k in order to guarantee learnability of queries of a certain size, our practical evaluation [13, 12] showed that small values of k (ranging between 2 and 4) are enough to cover many notable cases of graph query learning.

The above setting is static since the set of positive and negative examples is provided beforehand and no interaction with the user takes place during the learning process. An alternative, *interactive scenario*, can be envisioned that leads to a learning algorithm that starts with an empty sample and continuously interacts with the user during the construction of the input sample. The user provides positive and negative labels on the nodes of the input graph $\mathcal{G}$ until she is satisfied with the output of the learned query. Thus, the sample keeps growing until at most one query consistent with the user's labels is found. This scenario is inspired by the Angluin's model *learning with membership queries* [4].

Let S be a sample over a graph $\mathcal{G}$, the set of all queries consistent with S over $\mathcal{G}$ is defined as:

$$\mathcal{C}(\mathcal{G}, S) = \{q \in \text{RPQ} \mid S_+ \subseteq q(\mathcal{G}) \wedge S_- \cap q(\mathcal{G}) = \emptyset\}.$$

Assuming that the user labels the nodes consistently with some goal query q, the set $\mathcal{C}(\mathcal{G}, S)$ always contains q, where, initially, $S = \emptyset$.

Therefore, an ideal strategy of presenting nodes to the user is able to get us quickly from $S = \emptyset$ to a sample S s.t. $\mathcal{C}(\mathcal{G}, S) = \{q\}$. In particular, a good strategy should not propose to the user the *certain nodes* i.e., nodes not yielding new information when labeled by the user. Formally, given a graph $\mathcal{G}$, a sample S, and an unlabeled node $\nu \in \mathcal{G}$, we say that ν is *certain* (w.r.t. S) if it belongs to one of the following sets:

$$Cert_+(\mathcal{G}, S) = \{\nu \in \mathcal{G} \mid \forall q \in \mathcal{C}(\mathcal{G}, S).\ \nu \in q(\mathcal{G})\},$$
$$Cert_-(\mathcal{G}, S) = \{\nu \in \mathcal{G} \mid \forall q \in \mathcal{C}(\mathcal{G}, S).\ \nu \notin q(\mathcal{G})\}.$$

In other words, a node is certain with a label α if labeling it explicitly with α does not eliminate any query from $\mathcal{C}(\mathcal{G}, S)$.

The notion of certain nodes is inspired by possible world semantics and certain answers [29], and already employed for XML querying for non-expert users [20] and for the inference of relational joins [14, 15]. Additionally, given a graph $\mathcal{G}$, a sample S, and a node ν, we say that ν is *informative* (w.r.t. S), if it is neither labeled by the user nor certain.

An intelligent strategy should propose to the user only informative nodes. Since deciding the informativeness of a node is intractable, we need to explore *practical strategies* that efficiently compute the next node to label. The basic idea behind these is to avoid the intractability of deciding the informativeness of a node, by only looking at a small number of paths of that node. More precisely, we say that a node is *k-informative*, if it has at least one path of length at most k that is not covered by a negative example. If a node is k-informative, then it is also informative, otherwise we are not able to establish its informativeness w.r.t. the current k.

5. CONCLUSION AND PERSPECTIVES

In addition to our overview of topics we addressed concerning graph query evaluation, approximate processing and learning, we also would like to briefly outline the challenges encountered when tackling the problems of query benchmarking and log analysis, as reported in our previous work [8, 18]. In gMark [8], we addressed the graph and query workload generation problems concerning edge-labeled graph instances and UC2RPQ query workloads. Such a generator has been employed in the experimental evaluation of [16, 23] to generate varied test cases of these engines. Benchmarking is a long-standing question in our community, which serves, at the same time, the purpose of both system-driven and theoretical research. On the other hand, empirical analysis of real-world SPARQL query logs [18] also brought to our attention specific query language fragments adopted in practice by users and bots. A major future challenge is to let these studies influence the design of graph query benchmarks that take into account the requirements of users and

applications, as reflected by concrete usage of graph query languages.

As discussed in Section 2, the efforts to understand and hone in graph query expressivity have benefited from the purely-declarative, logic based formulation provided by Datalog. Various of its fragments have been tailored to user-specific applications, with Regular Datalog having recently emerged as an optimal compromise between usability and tractability. In this setting, we provide, as presented in Section 3, a mechanically certified specification of this query language, as well as a custom algorithm for its evaluation and fine-grained incremental maintenance in the dynamic setting. Additionally, we build on state-of-the-art theorem proving technology to verify, with the Coq theorem prover, the correct behavior of the engine. As shown in [16], its reasonable performance on realistic, synthetically-generated [8] graph instances ascertains the potential of employing formal methods to obtain correct-by-construction query engines.

Various works, such as [32, 9, 33], have tackled the complexity of evaluating graph queries, highlighting the challenges it poses. To mitigate these, in Section 4, we explore alternative processing approaches. A first such technique is based on approximate query evaluation, as introduced in [23]. Inspired by the internalization of transitive closure, which lies at the core of Regular Queries, we define a graph summarization that seeks to compress the nodes in the same label-connectivity closure. We combine the compactness of the obtained representation with the expressivity of the property graph model, used to store reachability preservation information, to maximize both evaluation efficiency and accuracy. With respect to existing related approaches, we base our work on the property graph model, leveraging aggregate pre-computation and query-driven graph summarization to provide scalable, high-accuracy, in-database query answers.

In Section 4.2, we have described a polynomial algorithm for learning graph queries of the basic RPQ class. More expressive fragments, for example those including conjunctions, can benefit from our previous work on learning relational queries [14]. A possible unification between the two lines of research would be desirable given the actual occurrences of C2RPQ in real-world query logs [18]. Another direction of future work would be to actually ameliorate the interactive paradigm presented in [14, 13, 12]. In these instances, the initial sample is empty and the user gradually fills the sample by providing positive and negative labels, until the inferred query and the user goal query coincide. For instance, more feedback from the learning system would be needed to more accurately model the user's intentions and to more efficiently reduce the search space given by the initial sample, through asking questions [5].

6. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Angles. The property graph database model. In *AMW*, volume 2100 of *CEUR Workshop Proceedings*, 2018.

[3] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. In *ACM Computing Surveys*, volume 50, pages 68:1–68:40, 2017.

[4] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[5] A. Arioua and A. Bonifati. User-guided repairing of inconsistent knowledge bases. In *EDBT*, pages 133–144, 2018.

[6] P. B. Baeza. Querying graph databases. In *PODS*, pages 175–188, 2013.

[7] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. Generating Flexible Workloads for Graph Databases. *PVLDB*, 9(13):1457–1460, 2016.

[8] G. Bagan, A. Bonifati, R. Ciucanu, G. H. L. Fletcher, A. Lemay, and N. Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering*, 29(4):856–869, 2017.

[9] G. Bagan, A. Bonifati, and B. Groz. A Trichotomy for Regular Simple Path Queries on Graphs. In *PODS*, pages 261–272. ACM, 2013.

[10] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *PODS*, pages 1–15, 1986.

[11] C. Bizer, J. Lehmann, G. Kobilarov, S. Auer, C. Becker, R. Cyganiak, and S. Hellmann. DBpedia - A crystallization point for the Web of Data. *Journal of Web Semantics*, 7(3):154–165, 2009.

[12] A. Bonifati, R. Ciucanu, and A. Lemay. Interactive Path Query Specification on Graph Databases. In *EDBT*, pages 505–508, 2015.

[13] A. Bonifati, R. Ciucanu, and A. Lemay. Learning Path Queries on Graph Databases. In *EDBT*, pages 109–120, 2015.

[14] A. Bonifati, R. Ciucanu, A. Lemay, and S. Staworko. A Paradigm for Learning Queries on Big Data. In *Data4U@VLDB*, page 7, 2014.

[15] A. Bonifati, R. Ciucanu, and S. Staworko. Learning Join Queries from User Examples. *ACM Transactions on Database Systems*, 40(4):24:1–24:38, 2016.

[16] A. Bonifati, S. Dumbrava, and E. J. G. Arias. Certified Graph View Maintenance with Regular Datalog. *Theory and Practice of Logic Programming*, 18(3-4):372–389, 2018.

[17] A. Bonifati, G. H. L. Fletcher, H. Voigts, and N. Yakovets. *Querying Graphs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2018.

[18] A. Bonifati, W. Martens, and T. Timm. An Analytical Study of Large SPARQL Query Logs. *PVLDB*, 11(2):149–161, 2017.

[19] A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *STOC*, pages 77–90, 1977.

[20] S. Cohen and Y. Weiss. Certain and possible XPath answers. In *ICDT*, pages 237–248, 2013.

[21] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[22] R. Delanaux, A. Bonifati, M. Rousset, and R. Thion. Query-based linked data anonymization. In *ISWC*, pages 530–546, 2018.

[23] S. Dumbrava, A. Bonifati, A. Ruiz-Diaz, and R. Vuillemot. Approximate Query Processing for Label-Constrained Reachability Queries. *CoRR*, abs/1811.11561, 2018.

[24] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat-Pérez, M. Pham, and P. A. Boncz. The LDBC social network benchmark: Interactive workload. In *SIGMOD*, pages 619–630, 2015.

[25] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[26] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A Machine-Checked Proof of the Odd Order Theorem. In *ITP*, pages 163–179, 2013.

[27] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. *SIGMOD Record*, 22(2):157–166, 1993.

[28] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[29] T. Imielinski and W. Lipski Jr. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

[30] A. P. Iyer, A. Panda, S. Venkataraman, M. Chowdhury, A. Akella, S. Shenker, and I. Stoica. Bridging the GAP: towards approximate graph analytics. In *GRADES*, pages 10:1–10:5, 2018.

[31] H. V. Jagadish, R. Agrawal, and L. Ness. A Study of Transitive Closure As a Recursion Mechanism. In *SIGMOD*, pages 331–344, 1987.

[32] K. Losemann and W. Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Transactions on Database Systems*, 38(4):24:1–24:39, 2013.

[33] W. Martens and T. Trautner. Evaluation and Enumeration Problems for Regular Path Queries. In *ICDT*, pages 1–21, 2018.

[34] Michel Dumontier and Alison Callahan and Jose Cruz-Toledo and Peter Ansell and Vincent Emonet and François Belleau and Arnaud Droit. Bio2RDF Release 3: A larger, more connected network of Linked Data for the Life Sciences. In *ISWC*, pages 401–404, 2014.

[35] J. Oncina and P. García. Inferring regular languages in polynomial update time. *Pattern Recognition and Image Analysis*, pages 49–61, 1992.

[36] J. L. Reutter, M. Romero, and M. Y. Vardi. Regular queries on graph databases. *Theory of Computing Systems*, 61(1):31–83, 2017.

[37] The Coq Development Team. The Coq proof assistant, version 8.8.0. `https://zenodo.org/record/1219885`, 2018.

[38] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[39] M. Y. Vardi. A theory of regular queries. In *PODS*, pages 1–9, 2016.

[40] Vrandečić, Denny and Krötzsch, Markus. Wikidata: A Free Collaborative Knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.