Jumping Evaluation of Nested Regular Path Queries

Joachim Niehren, Sylvain Salvati, Rustam Azimov

Jumping Evaluation of Nested Regular Path Queries

Joachim Niehren, Sylvain Salvati

Inria Lille, Université de Lille, France

Rustam Azimov

Saint Petersburg State University, Russia

Nested regular path queries are used for querying graph databases and RDF triple stores. We propose a new algorithm for evaluating nested regular path queries on a graph from a set of start nodes in combined linear time. We show that this complexity upper bound can be reduced by making it dependent on the size of the query's *top-down needed* subgraph, a notion that we introduce. For many queries in practice, the top-down needed subgraph is way smaller than the whole graph. Our algorithm is based on a novel compilation schema from nested regular path queries to monadic datalog queries. Its complexity upper bound follows from known properties of top-down datalog evaluation. As an application, we show that our algorithm permits to reformulate in simple terms a variant of a very efficient automata-based algorithm proposed by Maneth and Nguyen that evaluates navigational path queries in datatrees based on indexes and jumping. Moreover, it overcomes some limitations of Maneth and Nguyen's: it is not bound to trees and applies to graphs; it is not limited to forward navigational XPath but can treat any nested regular path query and it can be implemented efficiently without any dedicated techniques, by using any efficient top-down datalog evaluator. We confirm the efficiency of our algorithm experimentally based on an implementation with LogicBlox.

Keywords   Graph databases, path queries, propositional dynamic logic, XPath, Datalog

1   Introduction

Regular path queries [8] are regular expressions for navigating in edge labeled graphs. They belong to the core of various query languages for graph databases and RDF triple stores. Nested regular path queries (NRPQs) [6] extend on regular expressions by adding filters with logical operators, that may again contain regular path queries. NRPQs were first invented as the programs of propositional dynamic logic (PDL) [4]. NRPQs are also part of nSPARQL for querying knowledge stores in the Semantic Web [9]. The restriction of NRPQs to data trees constitutes the navigational core of regular XPATH. The NRPQ $\text{edge}_a{}^*/\neg[\text{edge}_b{}^*/\text{edge}_c]?$, for instance, selects all nodes of an edge-labeled graph, which can be reached from a given set of start nodes over a sequence of a-labeled edges, under the condition expressed by the filter $\neg[\text{edge}_b{}^*/\text{edge}_c]$: there must not exist any outgoing sequence of edges satisfying the regular expression $\text{edge}_b{}^*/edge_c$.

The set of nodes that can be reached by an NRPQ P on a graph G with a set of start nodes S can be computed in combined linear time, i.e. in linear time in the product of the inputs $\mathscr{O}(|P||G|)$. This complexity upper bound is folklore in the context of database theory, XPATH, and nSPARQL, and was known already for PDL, but was first shown for the richer alternation-free modal μ-calculus [3]. However, it is also satisfied by many inefficient algorithms: only the relevant fraction of the graph database should be visited for answering a database query. Any efficient query answer algorithm must avoid complete traversals of large graphs whenever possible.

Which part of a graph is relevant for a answering a query may depend on the query answering algorithm though. Therefore, we formalize a notion of needed subgraph coined as *top-down needed subgraph*, as the subgraph that is traversed with a top-down evaluation of the query. We then search
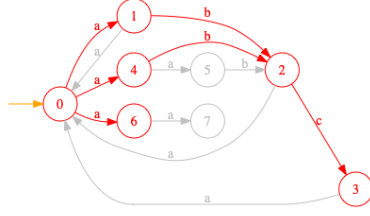
Figure 1: The graph G_0, the start set $S_0 = \{0\}$, and the top-down needed subgraph for $P_0 = \mathrm{edge}_a/[\mathrm{edge}_b/\mathrm{edge}_c]?$ in red.

$$q_0(x) :\!- q_1(x), q_2(x).$$
$$q_1(x) :\!- start(y), \mathrm{edge}_a(y,x).$$
$$q_2(x) :\!- \mathrm{edge}_b(x,y), q_3(y).$$
$$q_3(y) :\!- \mathrm{edge}_c(y,z).$$

Figure 2: The Datalog program M_0 for the nested regular path query P_0 from Fig. 1.

for a query answering algorithm with combined linear complexity *with respect to the top-down needed subgraph*, instead of the whole graph which we consider as too expensive.

For regular path queries (without filters) a canonical notion of the top-down needed subgraph seems quite intuitive, since a regular expression can be interpreted in a top-down manner as a navigation plan for traversing a graph. The top-down needed subgraph contains all the nodes and edges that are traversed when executing this navigation plan on the graph, while starting with the given set of start nodes. This notion of top-down needed nodes can then be lifted from to NRPQs, so that any filter of a NRPQ is tested only for those nodes where it is required when executing the query in a top-down manner.

For evaluating regular path queries (without filters) with the above complexity requirement, it is sufficient to execute it top-down navigation plan on the graph. We note that the presence of the Kleene star makes memoization mandatory, otherwise the algorithm may loop infinitely. What becomes more tedious is to find an evaluation algorithm for NRPQs that satisfies our complexity requirement. The existing proposals in [9, 2, 5] achieve combined linear time complexity by pre-evaluating the filters all over the graph in a bottom-up manner and then running an evaluation algorithm for regular path queries. However, the bottom-up pre-computation of the filters over all the graph may visit nodes that are *not* needed for the top-down evaluation of NRPQs, so these algorithms do not satisfy the envisaged complexity bound.

As an example, we consider in Fig. 1 the graph G_0 with edge labels $\{a,b,c\}$, the NRPQ $P_0 = \mathrm{edge}_a/[\mathrm{edge}_b/\mathrm{edge}_c]?$, and the set of start nodes $S_0 = \{0\}$. The query P_0 started at S_0 selects all those nodes of G_0 that are connected to the start node 0 by an a-edge, and have a path over a b-edge followed by a c-edge. The top-down algorithm with pre-evaluation of filters for P_0 will first compute the answer set of the filter $[\mathrm{edge}_b/\mathrm{edge}_c]$ on G_0 starting with S_0, which is $\{1,4,5\}$. It will then compute the set of nodes that are reached from the start node 0 over an a-edge which is $\{1,4,6\}$. The answer set is the intersection which is $\{1,4\}$. This algorithm, however, will inspect some nodes and edges for the pre-evaluation of the filters that are *not* top-down needed, namely the node 5 and the b-edge from 5 to 2. So the difficulty is to avoid the bottom-up pre-evaluation of filters.

We will show that NRPQs can be evaluated with the expected complexity by enhancing the naive top-down evaluator for NRPQs with memoization – instead of precomputing the filters. We obtain the right kind of memoization by compiling the path query into a monadic datalog program, and then evaluating this datalog program in a top-down manner with memoization. We note that a monadic datalog program may still use extensional predicates of higher arities, like the predicates edge_a, edge_b, and edge_c in our example. While the existence of compilers from NRPQs to monadic datalog is less surprising – even though none was published earlier to the best of our knowledge – the main difficulty is to find a datalog program that captures the top-down neededness with respect to the NRPQ. In the case of P_0 we obtain the datalog program in Fig. 2. We suppose that the table of the monadic extensional predicate *start* contains the set of start nodes in S_0. We note that paths in filters such as $[\mathrm{edge}_b/edge_c]$ are compiled

quite differently to paths outside filters. The reason is that paths outside filter have to return the end node that is reached, while paths inside filters have to jump back from the end node to the start node. The binary relation between end nodes and start nodes, however, cannot be stored in any predicate of the datalog program, since this would require a binary intensional predicate that are ruled out by monadic datalog. What we exploit instead is that a monadic datalog program can perform multiple tests on the same node, as with the rule $q_0(x) :- q_1(x), q_2(x)$. In our example, $q_1(x)$ will bind x to some node reached over the path $edge_a$ from the set of start nodes S_0, while $q_2(x)$ will test the filter $[edge_b/edge_c]$ there.

Our first contribution is an algorithm that answers NRPQs in the time $\mathcal{O}(|tdn_{G,S}(P)||P|)$ where $|tdn_{G,S}(P)|$ is the size of the top-down needed subgraph of G for query P. For this, we present a novel linear time compilation scheme mapping path queries to datalog queries. We restrict ourselves to negation-free NRPQs in order to avoid the usage of stratified negation for the sake of simplicity. We prove that if the compiler transforms a query P and a start set S into a datalog query M, then the top-down needed subgraph $tdn_{G,S}(P)$ is the part of the graph's database that is visited by top-down evaluation of the datalog query M on the database. Furthermore, the datalog queries produced are monadic and restricted in such a way, that their top-down evaluation can be done in combined linear time depending on the size of the top-down visited subdatabase. It follows that the answer set of an NRPQ P on a graph G with start set S can indeed be computed in time $\mathcal{O}(|tdn_{G,S}(P)||P|)$.

Our algorithm can be extended to a jumping algorithm for answering NRPQs on graphs with indexes. The indexes are binary relations defined by other NRPQs that allow the algorithm to jump in the graph. For instance, when given an index for the NRPQ $I = edge^*/a$? on the input graph, the evaluation algorithm can always jump to all a-labeled nodes accessible from the current node, without visiting the intermediates. We consider that the indexes are given with the input, since they are usually pre-computed elsewhere. Therefore, the indexes can simply be integrated into the graph as new edges that are labeled by the index's name, which is I in our example. Furthermore, the NRPQ is then rewritten by substituting all occurrences of I as a subquery in the NRPQ by $edge_I$, so that we can apply the previous machinery. An efficient implementation of our algorithm can be based on any efficient top-down datalog evaluator, since it is sufficient to evaluate the monadic datalog program produced by our compiler.

Our graph jumping algorithm permits to reformulate without specialized techniques a very efficient automata-based algorithm proposed by [7] that evaluates NRPQs on datatrees with indexes based on jumping. More precisely, their algorithm covers forward navigational XPATH queries on XML documents. It is based on alternating tree automata with selection states (which can be seen as binary datalog programs while ours are monadic). Our generic approach overcomes the limitations of their algorithm: it is not bound to trees but applies to graphs; it is not limited to forward navigational XPath but can treat any NRPQs also with backward steps. it can be implemented efficiently without any dedicated techniques, by using any efficient datalog evaluator supporting top-down evaluation such as LogicBlox [1].

Outline. In Section 2, we recall the definition of NRPQs. In Section 3, we formally define top-down needed subgraphs. In Section 4, we recall preliminaries on datalog queries, while discussing the complexity of top-down evaluation in Section 5. In Section 6, we give our compiler from NRPQs to datalog queries with its complexity theorem. Proofs can be found in the appendix. Section 7 presents the jumping evaluation algorithm for NRPQs on graphs with indexes, and Section 8 preliminary experimental results.

2 Nested Regular Path Queries

Regular path queries on labeled graphs [6] can be extended to NRPQs by adding filters with logical operators [8]. CoreXPath [5] is a sublanguage of NRPQs with limited recursion where the interpretation

filters $F \in \mathscr{F}_\Sigma$ ::= $[P] \mid node \mid node_a \mid F \wedge F' \mid F \vee F' \mid \neg F$
paths $P \in \mathscr{P}_\Sigma$ ::= $F? \mid edge_a \mid edge_a^{-1} \mid P/P' \mid P \cup P' \mid P^+ \mid goto(F)$

Figure 3: The syntax of NRPQs with labels $a \in \Sigma$.

$$\llbracket [P] \rrbracket_G = \{v \mid \exists v'. \ (v,v') \in \llbracket P \rrbracket_G\}$$
$$\llbracket node \rrbracket_G = V$$
$$\llbracket node_a \rrbracket_G = V_a$$
$$\llbracket \neg F \rrbracket_G = V \setminus \llbracket F \rrbracket_G$$
$$\llbracket F \wedge F' \rrbracket_G = \llbracket F \rrbracket_G \cap \llbracket F' \rrbracket_G$$
$$\llbracket F \vee F' \rrbracket_G = \llbracket F \rrbracket_G \cup \llbracket F' \rrbracket_G$$

$$\llbracket F? \rrbracket_G = \{(v,v) \mid v \in \llbracket F \rrbracket_G\}$$
$$\llbracket edge_a \rrbracket_G = E_a$$
$$\llbracket edge_a^{-1} \rrbracket_G = E_a^{-1}$$
$$\llbracket P/P' \rrbracket_G = \llbracket P \rrbracket_G \circ \llbracket P' \rrbracket_G$$
$$\llbracket P^+ \rrbracket_G = \llbracket P \rrbracket_G^+$$
$$\llbracket P \cup P' \rrbracket_G = \llbracket P \rrbracket_G \cup \llbracket P' \rrbracket_G$$
$$\llbracket goto(F) \rrbracket_G = \{(v,v') \mid v' \in \llbracket F \rrbracket_G\}$$

Figure 4: Semantics of NRPQs on a Σ-labeled digraph $G = (V, (V_a)_{a \in \Sigma}, (E_a)_{a \in \Sigma})$.

is restricted to an unranked tree. NRPQs were known even much earlier as the propositional dynamic logic (PDL) of [4].

We start from a finite set of labels Σ. A (finite) Σ-labeled digraph is a tuple $G = (V, (V_a)_{a \in \Sigma}, (E_a)_{a \in \Sigma})$ where V is a finite set of nodes, $V_a \subseteq V$ a finite subset of a-labeled nodes, and $E_a \subseteq V \times V$ a finite set of a-labeled edges where $a \in \Sigma$. Note that nodes may have multiple labels or none, while each edge has a unique label. Between two nodes there may be multiple edges with different labels though. An example for a labeled graph G_0 with labels in $\Sigma = \{a,b,c\}$ was given graphically in Fig. 1. The set of nodes of the graph is $V = \{0, \ldots, 7\}$. Here, the nodes are not labeled, so $V_a = V_b = V_c = \emptyset$. Each of the edge has a unique label. There are 8 a-labeled edges in E_a, 3 b-labeled edges in E_b and one c-labeled edge in E_c.

The syntax of NRPQs with labels in Σ is presented in Fig. 3. It consists of a set of filters $\mathscr{F}_\Sigma$ that select a set of graph nodes, and a set of paths $\mathscr{P}_\Sigma$ that select a set of pairs of graph nodes. The filter *node* selects all the nodes, while the filter *node$_a$* selects all a-labeled nodes. The set of nodes that are both a-labeled and b-labeled but not c-labeled is queried by filter $node_a \wedge node_b \wedge \neg node_c$. Path $edge_a$ selects all a-labeled edges and path edge $=_{df} \cup_{a \in \Sigma} edge_a$ the set of all edges. The path *node*? selects the identify on nodes $\{(v,v) \mid v \in V\}$. Path composition P/P', path union $P \cup P'$ are supported as well as repeated path composition P^+. The Kleene star on paths can be defined by $P^* =_{df} P^+ \cup node?$. Backwards edges can be queried by $edge_a^{-1}$, so that general backwards path P^{-1} can be defined, where $(P_1/P_2)^{-1} = P_2^{-1}/P_1^{-1}$ and $F?^{-1} = F?$. Finally, the path $goto(F)$ permits to jump to any node of the graph satisfying filter F. In particular, if there is a label $root \in \Sigma$ that distinguishes a set of roots, than path $goto(node_{root})/P$ first jumps to some root node before executing path P.

A little more complex example for an NRPQ with signature $\Sigma = \{a,b,c\}$ is the path query $P_2 = node_a?/(edge^+/[edge_b/edge_c]?)^*$. The evaluation of P_2 on a given graph from a start node tests whether the start node is a-labeled, and if so, it navigates from there repeatedly, over a sequence of edges to some node for which there exists an outgoing path over edges with labels b and then c. The set of all nodes reached this way is selected.

The semantics of paths P on labeled digraphs G is the binary relation $\llbracket P \rrbracket_G \subseteq V \times V$ defined in Fig. 4 in mutual recursivion with the semantics of filters $\llbracket F \rrbracket_G \subseteq V$. Despite its binary semantics, we will use paths for defining sets of nodes by fixing a start set S for the navigation. So let G be a labeled graph and S a subset of the nodes of G. For any $P \in \mathscr{P}_\Sigma$, the set $\llbracket P \rrbracket_G(S) = \{v \mid \exists v' \in S. \ (v',v) \in \llbracket P \rrbracket_G\}$ contains all nodes that can be reached when starting at some node of the start set S and navigating over the path P. Similarly, the set $\llbracket F \rrbracket_G(S) = \llbracket F \rrbracket_G \cap S$ contains all nodes from S that satisfy the filter F.

$$tdn_{G,S}(node) = \{node(v) \mid v \in S\}$$
$$tdn_{G,S}(node_a) = \{node(v) \mid v \in S\}$$
$$\cup \{node_a(v) \mid v \in V_a \cap S\}$$
$$tdn_{G,S}(F?) = tdn_{G,S}(F)$$
$$tdn_{G,S}(edge_a) = \{node(v) \mid v \in S\}$$
$$\cup \{edge_a(v,v'), node(v') \mid v \in S, (v,v') \in E_a\}$$
$$tdn_{G,S}(edge_a^{-1}) = \{node(v) \mid v \in S\}$$
$$\cup \{edge_a(v',v), node(v) \mid v' \in S, (v,v') \in E_a\}$$

$$tdn_{G,S}([P]) = tdn_{G,S}(P)$$
$$tdn_{G,S}(F \wedge F') = tdn_{G,S}(F) \cup tdn_{G,[\![F]\!]_G(S)}(F')$$
$$tdn_{G,S}(F \vee F') = tdn_{G,S}(F) \cup tdn_{G,S}(F')$$
$$tdn_{G,S}(P/P') = tdn_{G,S}(P) \cup tdn_{G,[\![P]\!]_G(S)}(P')$$
$$tdn_{G,S}(P^+) = tdn_{G,[\![P^+]\!]_G(S)}(P)$$
$$tdn_{G,S}(P \cup P') = tdn_{G,S}(P) \cup tdn_{G,S}(P')$$
$$tdn_{G,S}(goto(F)) = tdn_G(F) \qquad \text{(see Fig. 6)}$$

Figure 5: Facts of top-down needed subgraphs for negation-free paths and filters.

$$tdn_G(node) = \{node(v) \mid v \in V\}$$
$$tdn_G(node_a) = \{node_a(v) \mid v \in V_a\}$$
$$tdn_G(F?) = tdn_G(F)$$
$$tdn_G(edge_a) = \{edge_a(v,v') \mid (v,v') \in E_a\}$$
$$tdn_G(edge_a^{-1}) = \{edge_a(v',v) \mid (v,v') \in E_a\}$$

$$tdn_G([P]) = tdn_G(P)$$
$$tdn_G(F \wedge F') = tdn_G(F) \cup tdn_{G,[\![F]\!]_G}(F')$$
$$tdn_G(F \vee F') = tdn_G(F) \cup tdn_G(F')$$
$$tdn_G(P/P') = tdn_G(P) \cup tdn_{G,[\![P]\!]_G(V)}(P')$$
$$tdn_G(P^+) = tdn_{G,[\![P^+]\!]_G(V)}(P)$$
$$tdn_G(P \cup P') = tdn_G(P) \cup tdn_G(P')$$
$$tdn_G(goto(F)) = tdn_G(F)$$

Figure 6: Top-down needed subgraphs without start sets as neeeded for goto expressions.

3 Top-Down Needed Subgraphs

We are interested in the top-down evaluation of path queries, starting with a set of start nodes, and navigating along the path to other sets of nodes. The *top-down needed subgraph* of a path query will be the subgraph visited by such a traversal.

For the formal definition, we consider labeled graphs as extensional databases, i.e., as the sets of relational facts constructed from a relational signature and a set of constants. More concretely, we map any Σ-labeled graph $G = (V, (V_a)_{a \in \Sigma}, (E_a)_{a \in \Sigma})$ to the following set of database facts:

$$db(G) = \{node(v) \mid v \in V\} \cup \{node_a(v) \mid v \in V_a, \ a \in \Sigma\}$$
$$\cup \{edge_a(v,v') \mid (v,v') \in E_a, \ a \in \Sigma\}.$$

The facts are build from the monadic predicates $node$ and $node_a$ and the binary predicates $edge_a$ for all $a \in \Sigma$, and the graph nodes $v \in V$ as constants. Conversely, consider a set of facts D with the following properties: 1. if $node_a(v) \in D$ then $node(v) \in D$ and 2. if $edge_a(v,v') \in D$ then $node(v) \in D$ and $node(v') \in D$. For any such set D there exists a unique graph G such that $db(G) = D$. We can therefore identify any graph G with the sets of facts $D = db(G)$.

For any Σ-labeled digraph G and set of start nodes S we define in Fig. 5 the set of *facts of top-down needed subgraph $tdn_{G,S}(P)$* and $tdn_{G,S}(F)$ for negation-free paths P and filters F in mutual recursion. In the case of goto expressions, Fig. 6 defines $tdn_{G,S}(goto(F)) = tdn_G(F)$ for restarting the computation with all nodes satisfying F. The natural algorithm for computing the answer set of filter $node_a$ at start set S will filter for all nodes $v \in S$ such that $v \in V_a$. Therefore all nodes in S need to be visited, as well as the a-label of all nodes in $V_a \cap S$. The extensional database of the top-down needed subgraph $tdn_{G,S}(a)$ therefore contains the facts in $\{node(v) \mid v \in S\}$ and $\{node_a(v) \mid v \in V_a \cap S\}$. The definition

of $tdn_{G,S}(F \wedge F')$ is sequential from the left to the right. When the filter query F is failing for a node v then there is no need to check the filter query F' so as to know that the filter query $F \wedge F'$ is not verified by v. In contrast, the definition of $tdn_{G,S}(F \vee F')$ is done a parallel manner, so that both subfilters need to be evaluated from the start nodes. The sequential alternative would lead to smaller top-down needed subgraphs, which might seem advantageous:

$$tdn^{seq}_{G,S}(F \vee F', S) = tdn_{G,S}(F) \cup tdn_{G,[\![\neg F]\!]_G(S)}(F').$$

However, obtaining an evaluator with this sequential behavior by compilation to datalog would require us to use stratified negation, that we prefer to avoid for the sake of presentation. For the same reason, we restrict the definition of top-down needed subgraphs to negation-free path queries.

The definition of $tdn_{G,S}(P^+)$ is made of every attempt to construct a path of P starting from the nodes of S or the nodes that can be reached from S with a path of P^+. In the case of goto expressions, we have defined $tdn_{G,S}(goto(F)) = tdn_G(F)$ for restarting the computation with all nodes satisfying F. We could set $tdn_G(F)$ to $tdn_{G,V}(F)$, but this would not be optimal since all nodes of V would be top-down needed even for most simple filter $F = node_a$. A better definition where only the nodes of V_a are top-down needed is given in Fig. 6.

Example 1. *Consider the query $P_0 = \text{edge}_a[\text{edge}_b/\text{edge}_c]$ on the graph G_0 with signatue $\Sigma_0 = \{a, b, c\}$ in Fig. 1 with the start set $S_0 = \{0\}$. The set of top-down needed facts $tdn_{G_0, S_0}(P_0)$ is then $\{edge_a(0, 1), edge_a(0, 4), edge_a(0, 6), edge_b(1, 2), edge_b(4, 2), edge_c(2, 3)\}$. The top-down needed subgraph which is annotated in red in Fig. 1 is thus $graph(tdn_{G_0, \{0\}}(P_0)) = (\{0, \ldots, 6\}, (V_\ell)_{\ell \in \Sigma_0}, (E_\ell)_{\ell \in \Sigma_0})$ where $V_a = V_b = V_c = \emptyset$, $E_a = \{(0, 1), (0, 4), (0, 6)\}$, $E_b = \{(1, 2), (4, 2)\}$, and $E_c = \{(2, 3)\}$.*

4   Datalog Queries

We recall preliminaries on the syntax and semantics of datalog programs without negation and how to use them to define datalog queries on extensional databases.

The syntax of datalog is parametrized by a finite set of predicates $p, q, r \in \mathcal{P}$ and a disjoint finite set of *constants* $a, b, c \in \mathcal{C}$. The set of predicates is partitionned into a subset of *extensional predicates* $\mathcal{P}_{ext}$ and a disjoint subset of *intensional predicates* $\mathcal{P}_{int}$, so $\mathcal{P} = \mathcal{P}_{ext} \cup \mathcal{P}_{int}$. Constants will serve as database elements and extensional predicates for naming database relations. An *(extensional) database* is a subsets of ground literals of the form $p(a_1, \ldots, a_n)$ where $p \in \mathcal{P}_{ext}$ has arity $n \geq 0$ and $a_1, \ldots, a_n \in \mathcal{C}$.

We fix a set of variables $\mathcal{V} = \{x, y, z, \ldots\}$ distinct from the constants and predicates. A *term* $u, s, t \in \mathcal{T}_\mathcal{C} = \mathcal{V} \uplus \mathcal{C}$ is either a variable or a constant. The set of *(positive) literals* $\mathcal{L}$ is a subset of terms of the form $q(u_1, \ldots, u_n)$ where $q \in \mathcal{P}$ has arity n and $u_1, \ldots, u_n \in \mathcal{T}_\mathcal{C}$. A vector of terms is denoted by $\vec{t} \in \mathcal{T}_\mathcal{C}^*$. The set of all literals with extensional predicates is denoted by $\mathcal{L}_{ext}$ and those with intensional predicates by $\mathcal{L}_{int}$. A *goal* is a vector of literals $\vec{\ell} \in \mathcal{L}^*$ that is to be understood as a conjunction. The set of free variables $fv(\vec{t}), fv(\vec{\ell}) \subseteq \mathcal{V}$ are defined as usual. Similarly for the sets of occuring constants $cst(\vec{t}), cst(\vec{\ell}) \subseteq \mathcal{C}$. A *clause* is a pair of the form $q(\vec{t}) :- \vec{\ell}$. where $q(\vec{t}) \in \mathcal{L}_{int}$ and $\vec{\ell} \in \mathcal{L}^*$. We call $q(\vec{t})$ the *head* and $\vec{\ell}$ the *body* of the clause. The clause $q(\vec{t}) :- \vec{\ell}$. is *safe* if $fv(\vec{t}) \subseteq fv(\vec{\ell})$. We only work with safe clauses throughout this paper.

A *(safe) datalog program* is a finite subset M of safe clauses. A *(safe) datalog query* has the form $? - \vec{\ell}$. M, where $\vec{\ell} \in \mathcal{L}^*$ is a datalog goal and M a safe datalog program M. We now turn our attention to the semantics of datalog queries. Given a datalog query $? - \vec{\ell}$. M and an extensional database D, we need to define the set of substitutions that answer the query. A *substitution* is a finite partial function σ

$$[\![\varepsilon]\!]_{M,D} = \{[]\}$$

$$[\![\ell]\!]_{M,D} = \begin{cases} \{\Pi_{fv(\ell)}(\sigma \bowtie \sigma') \mid \sigma = \mathit{unif}(\ell,\ell'), \ \ell' :\!-\ \vec{\ell}. \text{ in } \mathit{ren}(M), \ \sigma' \in [\![\sigma(\vec{\ell})]\!]_{M,D}\} & \text{if } \ell \in \mathscr{L}_{int} \\ \{\Pi_{fv(\ell)}(\sigma) \mid \sigma = \mathit{unif}(\ell,\ell'), \ \ell' \in D\} & \text{if } \ell \in \mathscr{L}_{ext} \end{cases}$$

$$[\![\ell_1 \ldots \ell_n]\!]_{M,D} = \{\sigma' \bowtie \sigma \mid \sigma \in [\![\ell_1]\!]_{M,D}, \ \sigma' \in [\![\sigma(\ell_2 \ldots \ell_n)]\!]_{M,D}\} \qquad \text{where } n \geq 2$$

Figure 7: Least fixed-point semantics of a datalog query $?\!-\!\vec{\ell}$. M on a database D for $\ell, \ell_1, \ldots \ell_n \in \mathscr{L}$.

from $\mathscr{V}$ to $\mathscr{T}_{\mathscr{C}}$. We write $[]$ for the empty substitution. Any substitution can be lifted to a total function on all variables by defining $\sigma(x) = x$ for all $x \notin \mathit{dom}(\sigma)$. We lift substitutions further to total functions $\sigma : \mathscr{T}_{\mathscr{C}}^* \to \mathscr{T}_{\mathscr{C}}^*$ such that for all $n \geq 0$, $t_1, \ldots, t_n \in \mathscr{T}_{\mathscr{C}}$ and $a \in \mathscr{C}$:

$$\sigma(t_1 \ldots t_n) = \sigma(t_1) \ldots \sigma(t_n) \quad \text{and} \quad \sigma(a) = a$$

Similarly, substitutions are lifted to functions $\sigma : \mathscr{L}^* \to \mathscr{L}^*$ such that for all $\vec{t} \in \mathscr{T}_{\mathscr{C}}^*$ and $\ell_1, \ldots, \ell_n \in \mathscr{L}$:

$$\sigma(q(\vec{t})) = q(\sigma(\vec{t})), \quad \text{and} \quad \sigma(\ell_1 \ldots \ell_n) = \sigma(\ell_1) \ldots \sigma(\ell_n)$$

The *renaming closure* of a program is the set of all clauses that can be obtained from the clauses of the program by renaming variables bijectively:

$$\mathit{ren}(M) = \{\sigma(\ell) :\!-\ \sigma(\vec{\ell}) \mid \ell :\!-\ \vec{\ell}. \text{ in } M, \sigma \text{ is one-to-one substitution}, \mathit{ran}(\sigma) \subseteq \mathscr{V}\}$$

We define joins and projections on substitutions as for the relational algebra: for any two substitutions σ and σ' and any finite subset of variables $V \subseteq \mathscr{V}$:

$$\sigma \bowtie \sigma' = \begin{cases} \sigma \cup \sigma' & \text{if } \sigma \cup \sigma' \text{ is functional} \\ \text{undefined} & \text{otherwise} \end{cases} \qquad \Pi_V(\sigma) = \sigma_{|V}$$

For any two literals ℓ, ℓ' we define $\mathit{unif}(\ell, \ell')$ as the most general unifier σ such that $\sigma(\ell) = \sigma(\ell')$ if it exists, and leave it undefined otherwise.

We define the semantics $[\![\vec{\ell}]\!]_{M,D}$ of a datalog query $?\!-\!\vec{\ell}$. M on an extensional database D as the least fixpoint that satisfies the equations in Fig. 7. Notice that whenever we use the operation $\sigma \bowtie \sigma'$ then we have $\mathit{dom}(\sigma) \cap \mathit{dom}(\sigma') = \emptyset$, so that $\sigma \bowtie \sigma' = \sigma \cup \sigma'$ is a well-defined substitution. Each query answer $\sigma \in [\![\vec{\ell}]\!]_{M,D}$ has domain $fv(\vec{\ell})$ and always maps to constants since we work with safe datalog programs, so $\sigma : fv(\vec{\ell}) \to \mathscr{C}$. The semantics that we have given mimics the top-down datalog evaluation, which starts with the goal in the query and generates subgoals by unfolding the clauses of the datalog program, while instantiating the variables, until it reaches some ground facts from the extensional database. In general, this process may enter into infinite loops if not controlled by memoization. The whole top-down evaluation can always be represented as a join tree as we illustrate by example in Fig. 8. In the case of infinite loops, the join tree is infinite.

5 Complexity of Top-Down Evaluation of Datalog Queries

Known results on the complexity of top-down datalog evaluation give us the formal tools to prove for particular datalog queries, that the complexity of the top-down evaluation is in combined linear time but with respect to the top-down visited sub-database, rather than with respect to the full database.
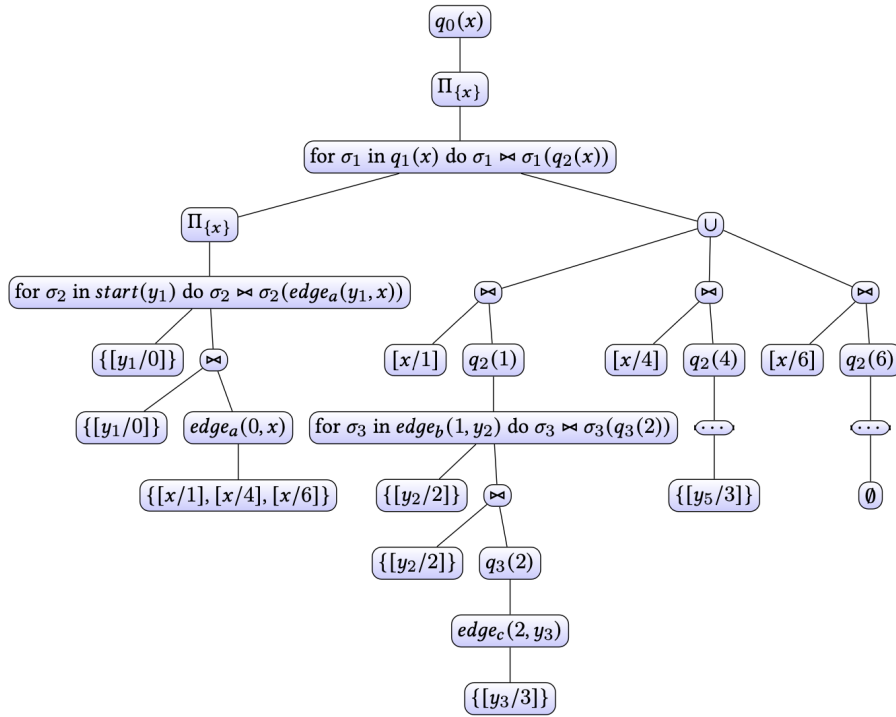
Figure 8: Top-down evaluation of $[\![q_0(x)]\!]_{M_0,db(G_0)\cup\{start(0)\}} = \{[x/1],[x/4]\}$ where M_0 is the datalog program from Fig. 2 for $P_0 = edge_a[edge_b/edge_c]$, and G_0 the graph from Fig. 1.

$$tdv_{M,D}(\varepsilon) = \emptyset$$
$$tdv_{M,D}(\ell) = \begin{cases} \{node(a) \mid a \in cst(\ell)\} \cup \bigcup \{\ell' \mid unif(\ell,\ell') \text{ defined}, \ell' \text{ in } D\} & \text{if } \ell \in \mathscr{L}_{ext} \\ \{node(a) \mid a \in cst(\ell)\} \cup \bigcup \{tdv_{M,D}(\sigma(\vec{\ell})) \mid \sigma = unif(\ell,\ell'), \ell' :- \vec{\ell}. \text{ in } ren(M)\} & \text{if } \ell \in \mathscr{L}_{int} \end{cases}$$
$$tdv_{M,D}(\ell_1\ldots\ell_n) = tdv_{M,D}(\ell_1) \cup \bigcup \{tdv_{M,D}(\sigma(\ell_2\ldots\ell_n)) \mid \sigma \in [\![\ell_1]\!]_{M,D}\}$$

Figure 9: The top-down visited sub-database $tdv_{M,D}(\vec{\ell})$ where $\ell,\ell_1,\ldots,\ell_n \in \mathscr{L}$ and $n \geq 2$.

For any datalog query $?{-}\vec{\ell}.\ M$ and extensional database D we next define the part of D that is visited by the top-down evaluation of the datalog query. For this we assume that the set of extensional predicates of D contains a monadic predicate $node \in \mathscr{P}_{ext}$ such that $node^D = \mathscr{C}$. We define the *top-down visited sub-database $tdv_{M,D}(\vec{\ell})$* as the extensional database over $\mathscr{P}_{ext}$ – following the semantics of datalog queries – as the least fixed point of equations in Fig. 9.

Definition 1. *We call a datalog goal $\vec{\ell}$ simply combined linear (SCL) if any proper prefix of $\vec{\ell}$ is SCL and $fv(\vec{\ell})$ is either guarded by a single extensional literal of $\vec{\ell}$ or contains no more than one variable. We call a datalog query $?{-}\vec{\ell}.\ M$ SCL if the datalog goal $\vec{\ell}$ is SCL and for each of the clauses $\ell :{-}\ \vec{\ell}.$ in the datalog program M, the datalog goal $\ell\vec{\ell}$ is SCL.*

For example, let $p,\ q \in \mathscr{P}_{int}$ be monadic and $r \in \mathscr{P}_{ext}$ be binary. The goal $p(x),r(x,y),q(y)$ is then SCL, since its prefix $p(x)$ contains no more than one variable, and both of its variables x and y are guarded by the extensional literal $r(x,y)$. The goal $p(x),r(x,x),q(y)$ on the contrary is not SCL, as it contains two variables of which y is not guarded by any extensional literal. The goal $p(x),q(x)$ is SCL since it contains no more than a single free variable.

Given an extensional database D, any SCL goal $\vec{\ell}$ has a number of ground instances that is linear in the size of D. Even better the number of ground instances inspected by top-down evaluation of the datalog query $?-\vec{\ell}$. M is linear in the size of the top-down visited database $tdv_{M,D}(\vec{\ell})$. In the case where $fv(\vec{\ell})$ contains at most one variable, this variable must be instantiated by some node of the top-down visited sub-database. Otherwise, the set of free variables $fv(\vec{\ell})$ is guarded by a single extensional literal of $\vec{\ell}$, say $p(\vec{t}) \in \mathscr{L}_{ext}$. In this case, any ground instance of $\vec{\ell}$ visited by the top-down evaluation of M is determined by $unif(p(\vec{t}), p(\vec{v}))$ for some fact $p(\vec{v}) \in tdv_{M,D}(\vec{\ell})$.

Theorem 2. *The answer set $[\![\vec{\ell}]\!]_{M,D}$ of a safe SCL query $?-\vec{\ell}$. M on an extensional database D can be computed in time $\mathcal{O}(|M||tdv_{M,D}(\vec{\ell})|)$.*

For proof we can show for any safe SCL datalog queries, that its top-down evaluation with memoization can be done in combined linear time with respect to the size of the top-down visited sub-database. For this, we can rely on the top-down evaluator in Figure 1 of [11]. The needed arguments on safe SCL datalog programs were given above. We also note that the magic set transformation on datalog programs without negation can be used to reduce top-down evaluation with memoization to semi-naive bottom-up evaluation. As stated by Theorem 3 of Ullman [12], the bottom-up evaluator obtained is at least as time efficient as the top-down evaluator. The magic set transformation, however, may need exponential space. Tekle and Liu [11] show that this problem can be solved by perfoming it on demand. They also proposed an on demand magic set transformation for stratified datalog programs [10].

6 Compiler to SCL Datalog Queries

We now contribute the compiler from negation-free path queries P and start set S to SCL datalog queries $?-\vec{\ell}$. M, such that for any graph G with nodes subsuming S, the extensional database of the top-down needed subgraph $tdn_{G,S}(P)$ is equal to the top-down visited sub-database $tdv_{M,db(G)}(\vec{\ell})$. The top-down evaluation of the datalog query $?-\vec{\ell}$. M on the graph's database $db(G)$ thus yields the expected upper complexity bound for the evaluation of path queries by Theorem 2.

For any set of start nodes S and monadic predicate $i \in \mathscr{P}_{int}$, we define a datalog program $Start^i(S) = \{i(v) :- . \mid v \in S\}$. The compilation scheme for path queries follows the structure of paths and filters by mutual recursion. It is given by the datalog programs $Acc^{i,f}(P)$ in Fig. 10, $Filt^c(F)$ in Fig. 11 and $Ex^{c,r}(P)$ in Fig. 12. Path queries outside filters need to compute all accessible nodes by $Acc^{i,f}(P)$, while path queries within filters need to check the existence of accessible nodes by $Ex^{c,r}(P)$. The compiler introduces fresh monadic predicates for all subexpressions: *initial predicates* $i, i', i'' \in \mathscr{P}_{int}$, *final predicates* $f, f', f'' \in \mathscr{P}_{int}$ *final*, *checks*. $c, c', c'' \in \mathscr{P}_{int}$, and *continuations* $r, r', r'' \in \mathscr{P}_{int}$.

Given a graph G and with a start set $S \subseteq V$ of graph nodes, the answer set of the datalog query $?-f(x)$. $Acc^{i,f}(P) \cup Start^i(S)$ on the extensional database $db(G)$ is $\{[x/v] \mid v \in [\![P]\!]_G(S)\}$, assigning the free variable x to some node v reachable from S over P in G. The initial predicate i captures the set of start nodes, and the *final* predicate f the answer set of the path query P started from there. The fresh monadic predicates make the datalog programs for the subexpressions able to communicate. For instance, we have $Acc^{i,f}(P'/P'') = Acc^{i,f'}(P') \cup Acc^{f',f}(P'')$. Here the final predicate $f' \in \mathscr{P}_{int}$ represents the answer set of path P' started at node set i, but also the start set for the path P''. This is since the start nodes of P'' in the query P'/P'' are the nodes that are reached with the query P'. For the recursive path queries P^+ we have $Acc^{i,f}(P^+) = Acc^{i,f}(P) \cup \{i(x) :- f(x).\}$. Here the rule $i(x) :- f(x)$. represents the fact that once a node is reached by the query P^+ it becomes a possible start node for the same query.

We next consider the datalog programs $Filt^c(F)$ defined in Fig. 11. For any graph G the answer set of the datalog query $?-c(x)$. $Filt^c(F)$ on the extensional database $db(G)$ is $\{[x/v] \mid v \in [\![F]\!]_G\}$, so that

$$Acc^{i,f}(\text{edge}_a) = \{f(x) :- i(y), \text{edge}_a(y,x).\}$$
$$Acc^{i,f}(\text{edge}_a^{-1}) = \{f(x) :- i(y), \text{edge}_a(x,y).\}$$
$$Acc^{i,f}(P'/P'') = Acc^{i,f'}(P') \cup Acc^{f',f}(P'')$$
$$Acc^{i,f}(P^+) = Acc^{i,f}(P) \cup \{i(x) :- f(x).\}$$

$$Acc^{i,f}(P' \cup P'') = Acc^{i,f}(P') \cup Acc^{i,f}(P'')$$
$$Acc^{i,f}(goto(F')) = Filt^{f'}(F') \cup$$
$$\{f(x) :- j(), f'(x). \quad j() :- i(x).\}$$
$$Acc^{i,f}(F'?) = Filt^{f'}(F') \cup \{f(x) :- i(x), f'(x).\}$$

Figure 10: The datalog program $Acc^{i,f}(P)$ for path P and monadic predicates $i, f \in \mathscr{P}_{int}$.

$$Filt^c(a) = \{c(x) :- node_a(x).\}$$
$$Filt^c(node) = \{c(x) :- node(x).\}$$
$$Filt^c(F' \vee F'') = Filt^{c'}(F') \cup Filt^{c''}(F'') \cup$$
$$\{c(x) :- c'(x). \quad c(x) :- c''(x).\}$$

$$Filt^c(F' \wedge F'') = Filt^{c'}(F') \cup Filt^{c''}(F'') \cup$$
$$\{c(x) :- c'(x), c''(x).\}$$
$$Filt^c([P]) = Ex^{c,r}(P) \cup \{r(x) :- node(x).\}$$

Figure 11: The datalog program $Filt^c(F)$ for filter F and monadic predicate $c \in \mathscr{P}_{int}$.

$$Ex^{c,r}(\text{edge}_a) = \{c(x) :- \text{edge}_a(x,y), r(y).\}$$
$$Ex^{c,r}(\text{edge}_a^{-1}) = \{c(x) :- \text{edge}_a(y,x), r(y).\}$$
$$Ex^{c,r}(P'/P'') = Ex^{c,f}(P') \cup Ex^{f,r}(P'')$$
$$Ex^{c,r}(P^+) = Ex^{c,r}(P) \cup \{r(x) :- c(x).\}$$

$$Ex^{c,r}(P' \cup P'') = Ex^{c,r}(P') \cup Ex^{c,r}(P'')$$
$$Ex^{c,r}(goto(F')) = Filt^{c'}(F') \cup$$
$$\{c(x) :- j(). \quad j() :- c'(y), r(y).\}$$
$$Ex^{c,r}(F'?) = Filt^{c'}(F') \cup \{c(x) :- c'(x), r(x).\}$$

Figure 12: The datalog program $Ex^{c,r}(P)$ for path P with monadic predicates $c, r \in \mathscr{P}_{int}$.

the free variables x may be bound to any node seleced by the filter. Hence, for any start set S, the answer set of $?-i(x), c(x).\ Filt^c(F) \cup Start^i(S)$ is $\{[x/v] \mid v \in [\![F]\!]_G(S)\}$. The filter for all nodes is compiled to $Filt^c(node) = \{c(x) :- node(x)\}$. Thereby, the check c is called for all nodes of the graph. Note that *node* is an extensional predicate, so this clause is safe. A conjunction of filters $Filt^c(F' \wedge F'')$ is compiled by adding the clause $c(x) :- c'(x), c''(x)$ to the datalog programs $Filt^{c'}(F')$ and $Filt^{c''}(F'')$. The added clause checks sequentially, whether a node x is filtered by F' and if so whether it is also filtered by F''. A disjunction of filters $Filt^c(F' \vee F'')$ is compiled by adding the two clause $c(x) :- c'(x)$. and $c(x) :- c''(x)$. to the datalog programs $Filt^{c'}(F')$ and $Filt^{c''}(F'')$. The two added clauses check in parallel whether a node x is filtered by F' or whether x is filtered by F''.

In Fig. 12 we define the datalog programs $Ex^{c,r}(P)$ for evaluating paths P existentially as needed when paths are used in filters, that is $Filt^c([P]) = Ex^{c,r}(P) \cup \{r(x) :- node(x).\}$. The check predicate c denotes the set of source nodes, from which some target node can be reached over P, while r is the continuation to which the target node must belong. Given a graph G and a start set S, the answer set of the datalog query $?-c(x).\ Ex^{c,r}(P) \cup \{r(x) :- node(x).\}$ on the extensional database $db(G)$ is $\{[c/v] \mid (v, v') \in [\![P]\!]_G\}$. The continuation predicate r is required to allow us to compile path concatenations in filters, i.e., in $Ex^{c,r}(P'/P'') = Ex^{c,f}(P') \cup Ex^{f,r}(P'')$. Note that the interplay of the predicate c and r is similar to the one between i and f in $Acc^{i,f}(P)$.

Lemma 3. *For any path P, filter F, graph G, start set S, and monadic predicates $i, f, c, r \in \mathscr{P}_{int}$, the programs $Start^i(S)$, $Acc^{i,f}(P)$, $Filt^c(F)$, $Ex^{c,r}(P)$ are safe and SLC.*

The function $reach_{M,r}(\vec{\ell})$ defined in Fig. 13 returns the set of all nodes v, such that $r(v)$ is queried in the proccess of the top-down evaluation of the datalog query $?-\vec{\ell}.\ M$. Now, we provide the Propositions 4 and 5 for dividing the correctness proof into two parts. First — about subpaths and subfilters of some filter. Concommittantly with Theorem 2 they will imply the main efficiency Theorem 6.

$reach_{M,r}(\varepsilon) = \emptyset$

$reach_{M,r}(r(v), \vec{\ell}_1) = \{v\} \cup reach_{M,r}(\sigma(\vec{\ell}_2, \vec{\ell}_1)) \mid \sigma = unif(r(v), \ell'), \ell' :- \vec{\ell}_2.\ in\ ren(M)\}$

$reach_{M,r}(\ell, \vec{\ell}_1) = reach_{M,r}(\sigma(\vec{\ell}_2, \vec{\ell}_1)) \mid \sigma = unif(\ell, \ell'),\ \ell' :- \vec{\ell}_2.\ in\ ren(M)\}$ if $\ell \neq r(v)$

Figure 13: The $reach_{M,r}$ function for the datalog query $?-\vec{\ell}.\ M$.

Proposition 4. *For any filter query* $F \in \mathscr{F}_\Sigma$, *path query* $P \in \mathscr{P}_\Sigma$, *label* $a \in \Sigma$, *labeled graph G, subset* $S \subseteq V$ *of nodes of G, distinct monadic predicates* $i, c, r \in \mathscr{P}_{int}$ *and* $x \in \mathscr{V}$.

1. *if* $M = Filt^c(F) \cup Start^i(S)$ *and* $\vec{\ell} = i(x), c(x)$ *then:*
   - $[\![\vec{\ell}]\!]_{M,db(G)} = \{[x/v] \mid v \in [\![F]\!]_G(S)\}$
   - $tdv_{M,db(G)}(\vec{\ell}) = tdn_{G,S}(F)$

2. *if* $M = Ex^{c,r}(P) \cup Start^i(S) \cup \{r(x) :- node(x).\}$ *and* $\vec{\ell} = i(x), c(x)$ *then:*
   - $[\![\vec{\ell}]\!]_{M,db(G)} = \{[x/v] \mid v \in S,\ [\![P]\!]_G(\{v\}) \neq \emptyset\}$
   - $tdv_{M,db(G)}(\vec{\ell}) = tdn_{G,S}(P)$
   - $reached_M(\vec{\ell}) = [\![P]\!]_G(S)$

Proposition 5. *For any path query* $P \in \mathscr{P}_\Sigma$, *labeled graph G, subset S of nodes of G, distinct intensional predicates* $i, f \in \mathscr{P}_{int}$ *and* $x \in \mathscr{V}$, *if* $M = Acc^{i,f}(P) \cup Start^i(S)$ *then:*
   - $[\![f(x)]\!]_{M,db(G)} = \{[x/v] \mid v \in [\![P]\!]_G(S)\}$
   - $tdv_{M,db(G)}(f(x)) = tdn_{G,S}(P)$

Theorem 6. *For any graph G with subset of nodes S and any path query* $P \in \mathscr{P}_\Sigma$ *the answer set* $[\![P]\!]_G(S)$ *can be computed in time* $\mathcal{O}(|P||tdn_{G,S}(P)|)$.

7   Jumping in Graphs

Preprocessing is mandatory for sharing efforts when evaluating multiple queries on the same large graph. Most typically, one can pre-compute indexes that give efficient access to some particular relations of the graph. Here we consider indexes, which are binary relations defined by NRPQs themselves.

For instance, we might want to jump from a node of the graph to the next a-labeled node in some fixed total order. In this case, one would like to have a jumping algorithm that visits only the top-down needed subgraph, but taken with respect to the graph, that is enriched with extra edges labeled by the names of the indexes.

Let us next consider a little more complex example. For this we suppose that we have an index for the NRPQ $acc_a = edge^*/a?$. We can then extend the signature Σ with a new label acc_a, the graph G with acc_a-labeled edges for all pairs in $[\![acc_a]\!]_G$, and rewrite the target path query by substituting all its subqueries acc_a by $edge_{acc_a}$. This has the advantage that fewer nodes are top-down needed after the rewriting on the enriched graph. For instance, a top-down evaluator for the path query acc_a without jumping needed to inspect *all* nodes of the graph accessible from S, since all of them needed to be tested for whether they satisfied the filter query a. After the rewriting to $edge_{acc_a}$, a top-down algorithm can jump directly from the start nodes in S to the accessible a-labeled nodes by using the index, so only accessible a-labeled nodes will be visited.

The general jumping algorithm starts with a set of indexes for NRPQs say for $P_1, \ldots, P_n$. For answering a query P on a graph G with these indexes the jumping algorithm enriches the signature Σ by new labels $P_1, \ldots, P_n$, the original graph G with new labeled edges $E_{P_j} = [\![P_j]\!]_G$ where $1 \leq j \leq n$, and then substitutes in the target query P all occurrences of the subqueries P_j by $edge_{P_j}$. The order of the substitution can be chosen arbitrarily, depending on the intended jumping strategy. In this way, the top-down needed subgraph of the enriched graph for the rewritten query is intuitively exactly the subgraph of the original graph that a top-down evaluation algorithm with jumping needs to visit.

This jumping algorithm can be used to reformulate in simple terms a variant of the efficient automata-based algorithm from [7] that evaluates navigational path queries. More precisely, their algorithm covers navigational forward XPATH queries on XML documents. It is based on alternating tree automata with selection states, which can be seen a binary datalog programs, while ours are monadic. XML documents are seen as labeled graphs, with two edge labels *firstchild* and *nextsibling*. Their algorithm can be based on the indexes for jumping to the *a*-labeled children, that is edge$/a$?, and for jumping to the top-most *a*-labeled descendants, i.e., $top_a = (\text{edge}/\neg a?)^*/\text{edge}/a?$. An XPATH query such as `descendant::a` can the be rewritten as the NRPQ $(top_a)^+$. The evaluation of the query $(top_a)^+$ can then take advantage of the index edge$_{top_a}$. The main difference between both approaches is that ours doesn't try to produce the answer set in document order, while theirs does so. Therefore, binary indexes are sufficient for our purpose, while they need to use a ternary index (for relating following *a*-labeled nodes *x* of *y* below *z*.). Moreover, our algorithm traverses the same part of the XML document as theirs and will thus be as efficient while being much simpler in terms of presentation. Our general graph approach overcomes the main limitations of Maneth and Nguyen's: it is not bound to trees and is not limited to forward navigational XPath but can treat any NRPQs also with backward steps.

8   Preliminary Experiments

We implemented in OCaml our compiler from NRPQs to Datalog and also a compiler from navigational XPath queries to NRPQs on the graphs of XML documents. The edges of these gstring, and the edges by *first*, *next*, *name*, and attribute names.

We selected in in Fig. 14 two typical benchmark XPATH queries from [7] that can be applied to the scalable XML-documents from the XPathMark benchmark: query $Q01$ composes two child axis, and query $Q05$ two descendant axis $//listitem//keyword$. The translations for these XPath queries to the NRPQs *NRP.Q*01 and *NRP.Q*05 can also be found there.

Query *NRP.Q*01 is easier for top-down evaluation, since it does not contain vertically recursive axis, so that its top-down needed subgraph remains small on the benchmark documents. $Q05$ is more difficult since using descendant axis, so that the top-down needed subgraph *NRP.Q*05 is the whole graph if not using indexes. So we also computed the indexes $top_{keyword}$ and $top_{listitem}$ for the descendant axis of $Q05$ and added them as extra edges to the graphs. Furthermore, the optimized query *NRP.Q*05.*index* obtained from *NRP.Q*05 by using the index edges is given in Fig. 14 too.

The gold standard for the evaluation of XPath queries is obtained by using the Saxon XSLT evaluator. In order to measure the time we run the same XPath query 100 times in the same XSLT program with Saxon 10.5, substract the time needed to load and index the XML document and divide by 100. It turns out, that Saxon has the best performance in all our tests, confirming our conjecture that it performs jumping evaluation with indexing for descendant axis.

We then implemented and tested our jumping algorithm based on existing top-down Datalog evaluators. We started with OCaml's Datalog 0.6, but had to notice that the top-down evaluator did not always produce the correct results. We then experimented with the Prolog engines XSB 4.0 and SWI 8.4.1.1. On a small XML-document of 27KB, both engines perform decently, even though not as quick as Saxon. On Q01, they are one order of magnitude slower. The same holds for Q05 but only when using indexing.

We then considered a much bigger XML-document of 100MB. With this size we had to give up with SWI. XSB in contrast could read the graph of the XML document, but needed more than 30 minutes. Once the graph was read, it could answer the query *pdl.Q*05 without indexing in 35 seconds. With indexing the time for answering *pdl.Q*05.*index* went down to 5 seconds. Saxon, in contrast, can load the

Q01	/site/regions
NRP. Q01	$node_{document}?/\text{edge}_{first}/(\text{edge}_{next})^*/$ $node_{element}?/[\text{edge}_{name}/node_{site}?]?/$ $\text{edge}_{first}/(\text{edge}_{next})^*/$ $node_{element}?/[\text{edge}_{name}/node_{regions}?]?$

Q05	//listitem//keyword
NRP. Q05	$node_{document}?/(\text{edge}_{first}/(\text{edge}_{next})^*)^+/$ $node_{element}?/[\text{edge}_{name}/node_{listitem}?]?/$ $(\text{edge}_{first}/(\text{edge}_{next})^*)^+/$ $node_{element}?/[\text{edge}_{name}/node_{keywords}?]?$
NRP. Q05. index	$node_{document}?/(\text{edge}_{top_{listitem}})^+/$ $(\text{edge}_{top_{keyword}})^+$

Figure 14: Two benchmark XPATH queries from [7], their translation to NRPQs, and the indexed NRPQ queries.

27KB	Q01	Q05	Q05.index
Saxon	0.000206	indexing	0.000315
XSB	0.001	0.006	0.001
SWI	0.004	0.189	0.004
LogicBlox	0.0045	0.0054	0.0045

Figure 15: Time in seconds for querying the 27KB XML-document with indexes.

100MB	Q01	Q05	Q05.index
Saxon		indexing	0.0016
XSB		35.857	5.029
SWI	-	-	-
LogicBlox	0.0124	-	0.0974

Figure 16: Time in seconds for querying the 100MB XML-document with indexes.

graph in 15 seconds and answer query *Q*05 in 1.6 milliseconds. So for answering the query *Q*05, Saxon showed 4 orders of magnitude more efficient than XSB.

We finally investigated the LogicBlox system [1], a more recent deductive database system which implements the language LogiQL extending on Datalog. With version 4.38 of LogicBlox we could read the graph of 100 MB in 19 seconds (rather then in more than 30 minutes as with XSB). LogicQL is a typed language implying some minor syntactic differences to standard datalog. Finally, LogicBlox has a transaction level, that permits to interact with graphs dynamically, so that it can be queried many times without being reloaded. The earlier versions of LogicBlox supported bottom-up evaluation only. But since recently, top-down evaluation can be chosen by adding On-Demand annotations for all extensional predicates. When doing so, we could answer the query *NRP.Q*05.*index* in 97.4 milliseconds on the 100MB document. This is 2 orders of magnitudes better than with XSB! Nevertheless it is still by a factor of 75 slower than with Saxon. Figure 4 of [7] reports 65 milliseconds for Q05 with optimal jumping, but on a slightly larger 116MB document. So the question is how the efficiency of our implementation could be increased further: with better indexes, early completion during Datalog evaluation, or by using special features of XPATH queries?

9   Conclusion and Future Work

The definition of the top-down needed subgraph allows us to prove that our algorithm for answering negation-free NRPQs visits only the interesting part of the graph. We believe that the restriction to negation-freeness can be relieved by compiling to stratified datalog. The new notion of top-down needed subgraphs may also allow the design of algorithms that transform NRPQs into equivalent ones that have a smaller top-down needed subgraph, for instance by inverting the path, or starting with some filter. In particular, the *goto* instructions permit the algorithm to jump directly to nodes with *rare* properties in the graph first and then compute the queries more efficiently. Another line of improvement would be to stop the evaluation of a filter when it has been proven correct. This effect may only be obtained if we

use a datalog top-down evaluator that follows the *early completion* strategy, i.e. stops whenever a ground predicate (such as filter queries in our case) is proven true.

References

[1] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen & Geoffrey Washburn (2015): *Design and Implementation of the LogicBlox System.* In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ACM, pp. 1371–1382, doi:10.1145/2723372.2742796. Available at `https://doi.org/10.1145/2723372.2742796`.

[2] Marcelo Arenas & Jorge Pérez (2011): *Querying Semantic Web Data with SPARQL.* In: *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '11, Association for Computing Machinery, New York, NY, USA, p. 305–316, doi:10.1145/1989284.1989312. Available at `https://doi.org/10.1145/1989284.1989312`.

[3] Rance Cleaveland & Bernhard Steffen (1993): *A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-Calculus.* Form. Methods Syst. Des. 2(2), p. 121–147, doi:10.1007/BF01383878. Available at `https://doi.org/10.1007/BF01383878`.

[4] Michael J. Fischer & Richard E. Ladner (1979): *Propositional Dynamic Logic of Regular Programs.* J. Comput. Syst. Sci. 18(2), pp. 194–211, doi:10.1016/0022-0000(79)90046-1. Available at `https://doi.org/10.1016/0022-0000(79)90046-1`.

[5] Georg Gottlob, Christoph Koch & Reinhard Pichler (2003): *The Complexity of XPath Query Evaluation.* In: *Proceedings of the Twenty-Second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '03, Association for Computing Machinery, New York, NY, USA, p. 179–190, doi:10.1145/773153.773171. Available at `https://doi.org/10.1145/773153.773171`.

[6] Leonid Libkin, Wim Martens & Domagoj Vrgovc (2013): *Querying Graph Databases with XPath.* In: *Proceedings of the 16th International Conference on Database Theory*, ICDT '13, Association for Computing Machinery, New York, NY, USA, p. 129–140, doi:10.1145/2448496.2448513. Available at `https://doi.org/10.1145/2448496.2448513`.

[7] Sebastian Maneth & Kim Nguyen (2010): *XPath Whole Query Optimization.* Proc. VLDB Endow. 3(1–2), p. 882–893, doi:10.14778/1920841.1920954. Available at `https://doi.org/10.14778/1920841.1920954`.

[8] Wim Martens & Tina Trautner (2018): *Evaluation and Enumeration Problems for Regular Path Queries.* In Benny Kimelfeld & Yael Amsterdamer, editors: *21st International Conference on Database Theory (ICDT 2018)*, LIPIcs 98, Dagstuhl, Germany, pp. 19:1–19:21, doi:10.4230/LIPIcs.ICDT.2018.19. Available at `http://drops.dagstuhl.de/opus/volltexte/2018/8594`.

[9] Jorge Pérez, Marcelo Arenas & Claudio Gutiérrez (2010): *nSPARQL: A navigational language for RDF.* J. Web Semant. 8(4), pp. 255–270, doi:10.1016/j.websem.2010.01.002. Available at `https://doi.org/10.1016/j.websem.2010.01.002`.

[10] K. Tuncay Tekle & Yanhong A. Liu (2019): *Extended Magic for Negation: Efficient Demand-Driven Evaluation of Stratified Datalog with Precise Complexity Guarantees.* Electronic Proceedings in Theoretical Computer Science 306, pp. 241–254, doi:10.4204/eptcs.306.28. Available at `https://doi.org/10.4204%2Feptcs.306.28`.

[11] K. Tuncay Tekle & Yanhong A. Liu (2010): *Precise complexity analysis for efficient Datalog queries.* In: *PPDP'10 - Proceedings of the 2010 Symposium on Principles and Practice of Declarative Programming*, pp. 35–44, doi:10.1145/1836089.1836094.

[12] J. D. Ullman (1989): *Bottom-up Beats Top-down for Datalog.* In: *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, Association for Computing Machinery, New York, NY, USA, p. 140–149, doi:10.1145/73721.73736. Available at `https://doi.org/10.1145/73721.73736`.