Jumping Evaluation of Nested Regular Path Queries

Rustam Azimov, Joachim Niehren, Sylvain Salvati

Jumping Evaluation of Nested Regular Path Queries

Rustam Azimov
JetBrains Research, Saint Petersburg State University

Joachim Niehren, Sylvain Salvati
Inria Lille, Université de Lille

Abstract

The propositional dynamic logic is a fundamental language that provides nested regular path queries for datagraphs, as needed for querying graph databases and RDF triple stores. We propose a new algorithm for evaluating nested regular path queries. Not only does it evaluate path queries on datagraphs from a set of start nodes in combined linear time, but also this complexity bound depends only on the size of the query's top-down needed subgraph, a notion that we introduce formally. For many queries relevant in practice, the top-down neeeded subgraph is way smaller than the whole datagraph. Our algorithm is based on a new compilation schema from nested regular path queries to monadic datalog queries that we introduce. We prove that the top-down evaluation of the datalog program visits only the top-down needed subgraph for the path query. Thereby, the combined linear time complexity depending on the size of the top-down needed subgraph is implied by a general complexity result for top-down datalog evaluation (Tekle and Liu 2010).

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, our variant 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 specialized or dedicated techniques, by simply using any efficient datalog evaluator.

1 Introduction

Regular path queries (Martens and Trautner 2018) are regular expressions for navigating in edge labeled graphs. They belong to the core of various query languages for datagraphs, as part of query languages of graph databases and RDF triple stores. Nested regular path queries (NRPQs) (Libkin et al. 2013) extend on regular expressions by adding filters with logical operators, that in turn may contain regular path queries. They were first invented as the programs of propositional dynamic logic (PDL) (Fischer and Ladner 1979), constitute the navigational core of regular XPath where they are restricted to query datatrees, and are also part of *n*Sparql for querying knowledge stores in the semantic Web (Pérez et al. 2010).

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 $\mathcal{O}(|P||G|)$. This is folklore in the context of PDL, XPath, and *n*Sparql but was first shown for the richer alternation-free modal μ-calculus (Cleaveland and Steffen 1991). However, this complexity upper bound alone is far too high in practice: if the graph is a database then it may be too large for a complete traversal for each query. Furthermore, for many queries only a fraction of the graph may be relevant for answering the query, which which fraction may depend on the query answering algorithm. 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 propose 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, a canonical notion of the top-down needed subgraph seems quite intuitive. It contains all nodes and edges that are traversed when considering the path query as a description for navigation while starting in the given set of start nodes. Of course, the presence of the Kleene star makes memoization mandatory for otherwise the algorithm may loop infinitely. The part of the graph that is traversed this way is what we call the top-down needed subgraph. The notion of top-down needed nodes can then be lifted from regular path queries to NRPQs rather naturally. What becomes more tedious is to find an evaluation algorithm for NRPQs that satisfies our complexity requirement. The existing proposals in (Pérez et al. 2010; Arenas and Pérez 2011; Gottlob et al. 2003) 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. Evaluating the filters top-down seems more difficult, since one would have to jump back to the starting node, requiring to compute a binary relation. However, the bottom-up pre-computation of the filters over all the graph may visit nodes that are not needed for top-down evaluation of the NRPQ so these algorithms do not satisfy the envisaged complexity bound.

As an example, consider the NRPQ $P_0 = \mathrm{edge}_a[\mathrm{edge}_b/\mathrm{edge}_c]$, the graph G_0 in Fig. 2 with edge labels $\{a,b,c\}$, and set of start nodes $S_0 = \{0\}$. 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 an path over a b-edge followed by a c-edge. The top-down algorithm with pre-evaluation of filters will first compute the answer set of the filter filter $[\mathrm{edge}_b/\mathrm{edge}_c]$, 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.

We will show that this complexity problem can be avoided by enhancing the naive top-down evaluator with memoization – instead of precomputing the filter queries. The right kind of memoization can be obtained by compiling the path query into a monadic datalog program, and then evaluating this datalog program in a top-down manner. Even though monadic, the datalog program may still use extensional predicates of higher arities. In the case of P_0 we obtain the datalog program in Fig. 1, which for talking about graphs with edge labels in $\{a,b,c\}$ uses the binary extensional predicates $\mathrm{edge}_a, \mathrm{edge}_b, \mathrm{edge}_c$. Furthermore, there is the monadic extensional predicate start for representing the start set. We note that a filter query such as $[\mathrm{edge}_b/edge_c]$ is compiled quite differently (see the rules of the intensional predicates q_2 and q_3) to how one would compile to the path query $\mathrm{edge}_b/edge_c$. The reason is that a filter query returns the node where the path starts – under the condition that some node is reached at the end – while the path query selects all the nodes reached at the end.

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

Fig. 1. The Datalog program M_0 for the nested regular path query $P_0 = \text{edge}_a[\text{edge}_b/\text{edge}_c]$.

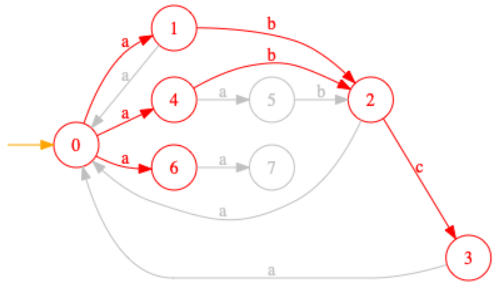

Fig. 2. Graph G_0, start set $S_0 = \{0\}$, and the top-down needed subgraph for P_0 in red.

Our first contribution is an algorithm that answers nested regular path queries in the combined linear time $O(|tdn_{G,S}(P)||P|)$ with respect to the size of top-down needed subgraph $tdn_{G,S}(P)$. For this, we present a linear time compilation scheme for mapping path queries to datalog queries. For the sake of presentation, we treat only negation-free NRPQs, so that stratified negation is not needed. We prove that the compiler is correct in that if it transforms a query P and a start set S into a datalog query M, then 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 as way that their top-down evaluation can be done in combined time depending on the size of the top-down visited subdatabase (Ullman's Theorem 3 on p9 of (Ullman 1989)) and (Tekle and Liu 2010) for an extensions with stratified negation). It follows that the answer set of the NRPQ on the graph with start set S can indeed be computed in time $O(|tdn_{G,S}(P)||P|)$.

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 $I = \text{edge}^*/a?$ 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 by integrated into the graph as new edges that are labeled by the index'es name, which is I in our example. Furthermore, the NRPQ is then rewritten by substituting all occurences 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 the evaluate the monadic datalog program produced by our compiler.

As an application our jumping algorithm permits to reformulate in simple terms a very efficient automata-based algorithm proposed by Maneth and Nguyen (Maneth and Nguyen 2010) that evaluates NRPQs on datatrees with indexes based on jumping. More precisely, their algorithm covers navigational forwards 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 approach overcomes the limitations of theirs: 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, and it can be implemented efficiently without any specialized or dedicated techniques.

Outline. In Section 2, we recall the definition of NRPQs. In Section 3, we formally define the notion of top-down needed subgraphs for NRPQs. In Section 4 recall preliminaries on datalog queries, while discussing the complexity of top-down evalatuion in Section 5. Our compiler from NRPQs to datalog queries with its complexity proof is given in Section 6. Section 7 presents the jumping evaluation algorithm for NRPQs on graphs with indexes.

2 Nested Regular Path Queries

Regular path queries on labeled graphs (Libkin et al. 2013) can be extended to NRPQs by adding filters with logical operators (Martens and Trautner 2018). CoreXPath (Gottlob et al. 2003) is a sublaguage NRPQs with limited recursion where the interpretation is restricted to unranked tree. What is still less well known is that NRPQs were known earlier as the propositional dynamic logic (PDL) (Fischer and Ladner 1979).

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. Note that nodes may have multiple labels or none, while each edge has unique label. Between to 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. 2. The set of nodes of the graph is $V = \{0,\dots,7\}$. The nodes are not labeled, so $V_a = V_b = V_c = \emptyset$. Each of the edge has a unique label. There are nine a-labeled edges in E_a, two b-labeled edges in E_b and one c-labeled edge in E_c.

The syntax of NRPQs with labels in Σ consists of a set of filters $\mathscr{F}_\Sigma$ select a set of graph nodes, and a set of paths $\mathscr{P}_\Sigma$, that select a set of pairs of graph nodes.

$$\begin{array}{llll} \text{filters} & F \in \mathscr{F}_\Sigma & ::= & [P] \mid \text{node} \mid \text{node}_a \mid F \wedge F' \mid F \vee F' \mid \neg F \qquad \text{where } a \in \Sigma \\ \text{paths} & P \in \mathscr{P}_\Sigma & ::= & F? \mid \mathsf{edge}_a \mid \mathsf{edge}_a^{-1} \mid P/P' \mid P \cup P' \mid P^+ \mid goto(F) \end{array}$$

Filter node selects all nodes, while filter node_a selects all a-labeled nodes. The set of nodes nodes that are both a-labeled and b-labeled but not c-labeled is queried by filter $\text{node}_a \wedge \text{node}_b \wedge \neg\text{node}_c$. Path edge_a selects all a-labeled edges and filter $\mathsf{edge} = \text{df} \cup_{a\in\Sigma} \mathsf{edge}_a$ 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 suspported as well repeated path composition P^+. The Kleene star on paths can be defined by $P^* =_{\text{df}} P^+ \cup \text{node}?$. 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 $goto(\text{node}_{root})/P$ 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 $P_2 = \text{node}_a?/(\mathsf{edge}^+/[\mathsf{edge}_b/\mathsf{edge}_c]?)^*$. 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 a binary relation $[\![P]\!]_G \subseteq V \times V$ that we define in Fig. 3 mutually recursively with the semantics of filters $[\![F]\!]_G \subseteq V$. Despite

$$
\begin{array}{ll}
[\![[P]]\!]_G = \{v \mid \exists v'.\ (v,v') \in [\![P]\!]_G\} & [\![F?]\!]_G = \{(v,v) \mid v \in [\![F]\!]_G\} \\
[\![\mathrm{node}]\!]_G = V & [\![\mathrm{edge}_a]\!]_G = E_a \\
[\![\mathrm{node}_a]\!]_G = V_a & [\![\mathrm{edge}_a^{-1}]\!]_G = E_a^{-1} \\
[\![\neg F]\!]_G = V \setminus [\![F]\!]_G & [\![P/P']\!]_G = [\![P]\!]_G \circ [\![P']\!]_G \\
[\![F \wedge F']\!]_G = [\![F]\!]_G \cap [\![F']\!]_G & [\![P^+]\!]_G = [\![P]\!]_G^+ \\
[\![F \vee F']\!]_G = [\![F]\!]_G \cup [\![F']\!]_G & [\![P \cup P']\!]_G = [\![P]\!]_G \cup [\![P']\!]_G \\
 & [\![\mathrm{goto}(F)]\!]_G = \{(v,v') \mid v' \in [\![F]\!]_G\}
\end{array}
$$

Fig. 3. Semantics of NRPQs on a Σ-labeled digraph $G = (V, (V_a)_{a\in\Sigma}, (E_a)_{a\in\Sigma})$.

of 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 $[\![P]\!]_G(S) = \{v \mid \exists v' \in S.\ (v',v) \in [\![P]\!]_G\}$ 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 $[\![F]\!]_G(S) = [\![F]\!]_G \cap S$ contains all nodes from S that satisfy the filter F.

3 Top-Down Needed Subgraphs

We are interested in the top-down evaluation of path queries, starting the navigation at the beginning of the path with a set of start nodes, and then moving along the path to other sets of node, reaching the end of the path.

We next define the subgraph that will be visited by such a traversal for a path query, and call it the top-down needed subgraph. For doing so we condider labeled graphs as extensional databases, i.e., as 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:

$$db(G) = \{\mathrm{node}(v) \mid v \in V\} \cup \{\mathrm{node}_a(v) \mid v \in V_a,\ a \in \Sigma\} \cup \{\mathrm{edge}_a(v,v') \mid (v,v') \in E_a,\ a \in \Sigma\}$$

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 $\mathrm{node}_a(v) \in D$ then $\mathrm{node}(v) \in D$ and 2. if $\mathrm{edge}_a(v,v') \in D$ then $\mathrm{node}(v) \in D$ and $\mathrm{node}(v') \in D$. 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. 4 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.

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 $\{\mathrm{node}(v) \mid v \in S\}$ and $\{\mathrm{node}_a(v) \mid v \in V_a \cap S\}$. 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, definition of $tdn_{G,S}(F \wedge F')$ in 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

$$
\begin{array}{ll}
tdn_{G,S}(\text{node}) = \{\text{node}(v) \mid v \in S\} & tdn_{G,S}([P]) = tdn_{G,S}(P) \\
tdn_{G,S}(\text{node}_a) = \{\text{node}(v) \mid v \in S\} & tdn_{G,S}(F \wedge F') = tdn_{G,S}(F) \cup tdn_{G,[\![F]\!]_G(S)}(F') \\
\quad \cup \{\text{node}_a(v) \mid v \in V_a \cap S\} & tdn_{G,S}(F \vee F') = tdn_{G,S}(F) \cup tdn_{G,S}(F') \\
tdn_{G,S}(F?) = tdn_{G,S}(F) & tdn_{G,S}(P/P') = tdn_{G,S}(P) \cup tdn_{G,[\![P]\!]_G(S)}(P') \\
tdn_{G,S}(\text{edge}_a) = \{\text{node}(v) \mid v \in S\} & tdn_{G,S}(P^+) = tdn_{G,[\![P^+]\!]_G(S)}(P) \\
\quad \cup \{\text{edge}_a(v,v'), \text{node}(v') \mid v \in S,\ (v,v') \in E_a\} & tdn_{G,S}(P \cup P') = tdn_{G,S}(P) \cup tdn_{G,S}(P') \\
tdn_{G,S}(\text{edge}_a^{-1}) = \{\text{node}(v) \mid v \in S\} & tdn_{G,S}(\text{goto}(F)) = tdn_G(F) \qquad \text{(see Fig. A 1)} \\
\quad \cup \{\text{edge}_a(v',v), \text{node}(v) \mid v' \in S,\ (v,v') \in E_a\} &
\end{array}
$$

Fig. 4. Facts of top-down needed subgraphs for negation-free paths and filters.

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 behaviour by compilation to datalog would require 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 $tdn_{G,S}(\text{goto}(F)) = tdn_G(F)$ 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 $F = \text{node}_a$. A better definition where only the nodes of V_a are top-down needed is given in Fig. A 1 of the appendix.

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. 2 with the start set $S_0 = \{0\}$. The set of top-down needed facts $tdn_{G_0,S_0}(P_0)$ is $\{\text{edge}_a(0,1)$, $\text{edge}_a(0,4)$, $\text{edge}_a(0,6)$, $\text{edge}_b(1,2)$, $\text{edge}_b(4,2)$, $\text{edge}_c(2,3)\}$. The top-down needed subgraph $graph(tdn_{G_0,\{0\}}(P_0))$ which is annotated in red in Fig. 2 is thus $(\{0,\ldots,6\}$, $(V_\ell)_{\ell\in\Sigma_0}, (E_\ell)_{\ell\in\Sigma_0})$ 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 $p,q,r \in \mathscr{P}$ and a disjoint finite set of constants $a,b,c \in \mathscr{C}$. The set of predicates is partitionned into a subset of extensional predicates $\mathscr{P}_{ext}$ and a disjoint subset of intensional predicates $\mathscr{P}_{int}$, so $\mathscr{P} = \mathscr{P}_{ext} \cup \mathscr{P}_{int}$. 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 $p \in \mathscr{P}_{ext}$ has arity $n \geq 0$ and $a_1,\ldots,a_n \in \mathscr{C}$.

We fix a set of variables $x,y,z, \in \mathscr{V}$ distinct from the constants and predicates. A term $u,s,t \in \mathscr{T}_\mathscr{C} = \mathscr{V} \uplus \mathscr{C}$ is either a variable or a constant. The set of (positive) literals $\mathscr{L}$ is a subset of terms of the form $q(u_1,\ldots,u_n)$ where $q \in \mathscr{P}$ has arity n and $u_1,\ldots,u_n \in \mathscr{T}_\mathscr{C}$. A vector of terms is denoted by $\vec{t} \in \mathscr{T}_\mathscr{C}^*$. The set of all literals with extensional predicates is denoted by $\mathscr{L}_{ext}$ and those with intensional predicates by $\mathscr{L}_{int}$. 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 thoughout 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 from $\mathcal{V}$ to $\mathcal{T}_{\mathcal{C}}$. 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 $x \notin dom(\sigma)$. We lift substitutions further to total functions $\sigma : \mathcal{T}_{\mathcal{C}}^* \to \mathcal{T}_{\mathcal{C}}^*$ such that $\sigma(t_1 \dots t_n) = \sigma(t_1) \dots \sigma(t_n)$ and $\sigma(a) = a$ for all $n \geq 0$, $t_1, \dots, t_n \in \mathcal{T}_{\mathcal{C}}$ and $a \in \mathcal{C}$: Similarly, substitutions are lifted to total functions $\sigma : \mathcal{L}^* \to \mathcal{L}^*$ such that $\sigma(q(\vec{t})) = q(\sigma(\vec{t}))$, $\sigma(\ell_1 \dots \ell_n) = \sigma(\ell_1) \dots \sigma(\ell_n)$ for all $\vec{t} \in \mathcal{T}_{\mathcal{C}}^*$, $n \geq 0$, and $\ell_1, \dots, \ell_n \in \mathcal{L}$. 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:

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

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 $V \subseteq \mathcal{V}$:

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

For any two literals ℓ, ℓ' we define $unif(\ell, \ell')$ 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 the least fixpoint that satisfies the following equations for all $n \geq 2$, $\ell, \ell_1, \dots \ell_n \in \mathcal{L}$:

$$\begin{aligned}
&[\![\varepsilon]\!]_{M,D} = \{[]\} \\
&[\![\ell]\!]_{M,D} = \{\Pi_{fv(\ell)}(\sigma \bowtie \sigma') \mid \sigma = unif(\ell, \ell'),\ \ell' \,{:}{-}\, \vec{\ell}.\text{ in } ren(M),\ \sigma' \in [\![\sigma(\vec{\ell})]\!]_{M,D}\} \text{ if } \ell \in \mathcal{L}_{int} \\
&[\![\ell]\!]_{M,D} = \{\Pi_{fv(\ell)}(\sigma) \mid \sigma = unif(\ell, \ell'),\ \ell' \in D\} \text{ if } \ell \in \mathcal{L}_{ext} \\
&[\![\ell_1 \dots \ell_n]\!]_{M,D} = \{\sigma' \bowtie \sigma \mid \sigma \in [\![\ell_1]\!]_{M,D},\ \sigma' \in [\![\sigma(\ell_2 \dots \ell_n)]\!]_{M,D}\}
\end{aligned}$$

Notice that whenever we use the operation $\sigma \bowtie \sigma'$ then we have $dom(\sigma) \cap dom(\sigma') = \emptyset$, 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 $\sigma : fv(\vec{\ell}) \to \mathcal{C}$. 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. B 1 of the appendix. In the case of infinite loops, the join tree is infinite.

5 Complexity of Top-Down Evaluation of Datalog Queries

We now recall results on the complexity of top-down datalog evaluation from (Tekle and Liu 2010). This gives 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.

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 $\text{node} \in \mathscr{P}_{ext}$ such that $\text{node}^D = \mathscr{C}$. 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 such that for all $\ell, \ell_1, \ldots, \ell_n \in \mathscr{L}$ where $n \geq 2$:

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

Definition 1. We call a datalog goal $\vec{\ell}$ simply combined linear (SCL) if $fv(\vec{\ell})$ is a singleton or empty, or if all its variables are guarded by a single extensional literal of $\vec{\ell}$. 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 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 both of its variables x and y are guared 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 guared by a single extensional literal. The goal $p(x), q(x)$ is SCQ 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 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 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 (Ullman, Theorem 3 on p9 of (Ullman 1989)). The answer set $[\![\vec{\ell}]\!]_{M,D}$ of an safe SCL datalog query $?{-}\vec{\ell}.\ M$ on an extensional database D can be computed in time $\mathscr{O}(|M||tdv_{M,D}(\vec{\ell})|)$.

For safe SCL datalog queries, the time needed for query answering by top-down evaluation with memoization is thus indeed combined linear with respect to size of the top-down visited sub-database. An extension of this theorem to stratified datalog can be found in (Tekle and Liu 2010).

6 Compiler to SCL Datalog Queries

We now contribute a compiler from negation-free path queries P and start set S to SCL datalog queries $?{-}\vec{\ell}.\ M$, such 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 (Tekle and Liu 2010).

For any set of start nodes S and monadic predicate $i \in \mathscr{P}_{int}$, we define a datalog program $\mathrm{Start}^i(S) = \{i(v) :- . \mid v \in S\}$. The compilation scheme for path queries follows the structure of paths and filters by mutual recursion. It is given by the datalog programs $\mathrm{Acc}^{i,f}(P)$ in Fig. 5, $\mathrm{Filt}^c(F)$ in Fig. 6 and $\mathrm{Ex}^{c,r}(P)$ in Fig. 7. Path queries outside filters need to compute all accessible nodes by $\mathrm{Acc}^{i,f}(P)$, while path queries within filters need to check the existence of accessible nodes by $\mathrm{Ex}^{c,r}(P)$. 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 contintuations $r, r', r'' \in \mathscr{P}_{int}$.

Given a graph G and with a start set $S \subseteq V$ of graph nodes, the answer set of the datalog query $?{-}f(x).\ \mathrm{Acc}^{i,f}(P) \cup \mathrm{Start}^i(S)$ 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 to communicate. For instance, we have $\mathrm{Acc}^{i,f}(P'/P'') = \mathrm{Acc}^{i,f'}(P') \cup \mathrm{Acc}^{f',f}(P'')$. 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 $\mathrm{Acc}^{i,f}(P^+) = \mathrm{Acc}^{i,f}(P) \cup \{i(x) :- f(x).\}$. 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 $\mathrm{Filt}^c(F)$ defined in Fig. 6. For any graph G the answer set of the datalog query $?{-}c(x).\ \mathrm{Filt}^c(F)$ on the extensional database $db(G)$ is $\{[x/v] \mid v \in [\![F]\!]_G\}$, so that the free variables x may be bound to any node seleced by the filter. Hence, for any start set S, the answer set of $?{-}i(x), c(x).\ \mathrm{Filt}^c(F) \cup \mathrm{Start}^i(S)$ is $\{[x/v] \mid v \in [\![F]\!]_G(S)\}$. The filter for all nodes is compiled to $\mathrm{Filt}^c(\mathrm{node}) = \{c(x) :- \mathrm{node}(x)\}$. 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 $\mathrm{Filt}^c(F' \wedge F'')$ is compiled by adding the clause $c(x) :- c'(x), c''(x)$ to the datalog programs $\mathrm{Filt}^{c'}(F')$ and $\mathrm{Filt}^{c''}(F'')$. 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 $\mathrm{Filt}^c(F' \vee F'')$ is compiled by adding the two clause $c(x) :- c'(x).$ and $c(x) :- c''(x).$ to the datalog programs $\mathrm{Filt}^{c'}(F')$ and $\mathrm{Filt}^{c''}(F'')$. The two added clauses check in parallel whether a node x is filtered by F' or whether x is filtered by F''. Our compiler could be extended to negated filters, but this would require to compile to stratified datalog. Startified negation would also allow to compile disjunctions sequentially, so that only a smaller subgraph would get visited.

The accessiblity of nodes over goto-paths could be expressed by $\widetilde{\mathrm{Acc}}^{i,f}(goto(F')) = \mathrm{Filt}^{f'}(F') \cup \{f(x) :- i(y), f'(x).\}$ where $\mathrm{Filt}^{f'}(F')$ is the datalog program that represents the computation of the filter query F' and where f' is the predicate which captures the

$\mathrm{Acc}^{i,f}(\mathrm{edge}_a) = \{f(x) :- i(y), \mathrm{edge}_a(y,x).\}$
$\mathrm{Acc}^{i,f}(\mathrm{edge}_a^{-1}) = \{f(x) :- i(y), \mathrm{edge}_a(x,y).\}$
$\mathrm{Acc}^{i,f}(P'/P'') = \mathrm{Acc}^{i,f'}(P') \cup \mathrm{Acc}^{f',f}(P'')$
$\mathrm{Acc}^{i,f}(P^+) = \mathrm{Acc}^{i,f}(P) \cup \{i(x) :- f(x).\}$
$\mathrm{Acc}^{i,f}(P' \cup P'') = \mathrm{Acc}^{i,f}(P') \cup \mathrm{Acc}^{i,f}(P'')$
$\mathrm{Acc}^{i,f}(goto(F')) = \mathrm{Filt}^{f'}(F') \cup \{f(x) :- j(), f'(x). \quad j() :- i(x).\}$
$\mathrm{Acc}^{i,f}(F'?) = \mathrm{Filt}^{f'}(F') \cup \{f(x) :- i(x), f'(x).\}$

Fig. 5. The datalog program $\mathrm{Acc}^{i,f}(P)$ for path P and monadic predicates $i, f \in \mathscr{P}_{int}$.

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

Fig. 6. The datalog program $\mathrm{Filt}^c(F)$ for filter F and monadic predicate $c \in \mathscr{P}_{int}$.

$\mathrm{Ex}^{c,r}(\mathrm{edge}_a) = \{c(x) :- \mathrm{edge}_a(x,y), r(y).\}$
$\mathrm{Ex}^{c,r}(\mathrm{edge}_a^{-1}) = \{c(x) :- \mathrm{edge}_a(y,x), r(y).\}$
$\mathrm{Ex}^{c,r}(P'/P'') = \mathrm{Ex}^{c,f}(P') \cup \mathrm{Ex}^{f,r}(P'')$
$\mathrm{Ex}^{c,r}(P^+) = \mathrm{Ex}^{c,r}(P) \cup \{r(x) :- c(x).\}$
$\mathrm{Ex}^{c,r}(P' \cup P'') = \mathrm{Ex}^{c,r}(P') \cup \mathrm{Ex}^{c,r}(P'')$
$\mathrm{Ex}^{c,r}(goto(F')) = \mathrm{Filt}^{c'}(F') \cup \{c(x) :- j(). \quad j() :- c'(y), r(y).\}$
$\mathrm{Ex}^{c,r}(F'?) = \mathrm{Filt}^{c'}(F') \cup \{c(x) :- c'(x), r(x).\}$

Fig. 7. The datalog program $\mathrm{Ex}^{c,r}(P)$ for path P with monadic predicates $c, r \in \mathscr{P}_{int}$.

answer set. The clause $f(x) :- i(y), f'(x).$ means that a node x satisfying the filter F' is in the answer set if there is some node y in the start set S. So if $S \neq \emptyset$ then the datalog query $?{-}f'(x).\ \mathrm{Filt}^c(F') \cup \mathrm{Start}^i(S)$ can be reduced to the datalog query $?{-}c(x).\ \mathrm{Filt}^c(F')$, which on the extensional database $db(G)$ has the answer set $\{[c/v] \mid v \in [\![F']\!]_G\}$. We note that the top-down evaluation of the latter datalog query avoids visiting the nodes of the graph that are not top-down needed for the path $goto(F')$. If for instance $F' = \mathrm{node}_a$ then only the a-labeled nodes of the graph are visited. The problem with the definition discussed so far, however, is that the clause $f(x) :- i(y), f'(x)$ is not SCL (see Definition 1). Even worse, it would lead a quadratic evaluation time. Therefore, we replace it by the equivalent datalog program with two clauses $\{f(x) :- j(), f'(x).\} \cup \{j() :- i(y).\}$ which is SCL, leading to the definition of $\mathrm{Acc}^{i,f}(goto(F'))$ in Fig. 5. Here $j \in \mathscr{P}_{int}$ is a fresh nullary predicate that is true for $\mathrm{Start}^i(S)$ if and only if $S \neq \emptyset$.

In Fig. 7 we define the datalog programs $\mathrm{Ex}^{c,r}(P)$ for evaluating paths P existentially as needed when paths are used in filters, that is $\mathrm{Filt}^c([P]) = \mathrm{Ex}^{c,r}(P) \cup \{r(x) :- node(x).\}$. 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 $?{-}c(x).\ \mathrm{Ex}^{c,r}(P) \cup \{r(x) :- node(x).\}$ 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 $\mathrm{Ex}^{c,r}(P'/P'') = \mathrm{Ex}^{c,f}(P') \cup \mathrm{Ex}^{f,r}(P'')$. Note that the interplay of the predicate c and r is similar to the one between i and f in $\mathrm{Acc}^{i,f}(P)$.

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 datalog programs $\mathrm{Start}^i(S)$, $\mathrm{Acc}^{i,f}(P)$, $\mathrm{Filt}^c(F)$, $\mathrm{Ex}^{c,r}(P)$ are safe and SLC.

Our correctness propositions relies on the function $reach_{M,r}(\vec{\ell}\,)$ which 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 $?{-}\vec{\ell}.\ M$:

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

Now, we provide two propositions for dividing the correctness proof into two parts. First — about subpaths and subfilters of some filter.

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

Proof sketch
By simultaneous induction on the structures of $F \in \mathscr{F}_\Sigma$ and $P \in \mathscr{P}_\Sigma$. We discuss few cases for the possible forms of filters and paths only. For warming up, we consider the proof of property 1. for filters $F = F' \wedge F''$. The definition of the compiler yields:

$$\mathrm{Filt}^c(F' \wedge F'') = \mathrm{Filt}^{c'}(F') \cup \mathrm{Filt}^{c''}(F'') \cup \{c(x) :- c'(x), c''(x).\}$$

Let $M = \mathrm{Filt}^c(F' \wedge F'') \cup \mathrm{Start}^i(S)$, $M' = \mathrm{Filt}^{c'}(F') \cup \mathrm{Start}^i(S)$, and $M'' = \mathrm{Filt}^{c''}(F'') \cup \mathrm{Start}^i([\![F']\!]_G(S))$. The top-down evaluator for $?{-}i(x), c(x).\ M$ will visit $tdn_{G,S}(F')$ in order to get all facts corresponding to the filter query F'. After that, the top-down evaluator will start from all nodes in S that satisfy filter F' to find those that also satisfy the filter F''. First, we show that the top-down evaluation $[\![i(x), c(x)]\!]_M$ yields $\{[x/v] \mid v \in [\![F' \wedge F'']\!]_G(S)\}$. By induction hypothesis, $[\![i(x), c'(x)]\!]_{M'} = \{[x/v] \mid v \in [\![F']\!]_G(S)\}$ and $[\![i(x), c''(x)]\!]_{M''} = \{[x/v] \mid v \in [\![F'']\!]_G([\![F']\!]_G(S))\}$. Therefore, $[\![i(x), c(x)]\!]_M = \{[x/v] \mid v \in [\![F'']\!]_G \cap ([\![F']\!]_G \cap S)\} = \{[x/v] \mid v \in [\![F' \wedge F'']\!]_G \cap S\} = \{[x/v] \mid v \in [\![F' \wedge F'']\!]_G(S)\}$. Second, we show that the top-down visited sub-database is equal to $tdn_{G,S}(F' \wedge F'')$. By induction hypothesis, $tdv_{M',db(G)}(i(x), c'(x)) = tdn_{G,S}(F')$ and $tdv_{M'',db(G)}(i(x), c''(x)) = tdn_{G,[\![F']\!]_G(S)}(F'')$. Therefore: $tdv_{M,db(G)}(i(x), c(x)) = tdn_{G,S}(F') \cup tdn_{G,[\![F']\!]_G(S)}(F'') = tdn_{G,S}(F' \wedge F'')$.

We now turn to the most complicated case, which is the proof of property 2. for paths $P = P'/P''$. The definition of the compiler yields:

$$\mathrm{Ex}^{c,r}(P'/P'') = \mathrm{Ex}^{c,f}(P') \cup \mathrm{Ex}^{f,r}(P'')$$

Let $M = \mathrm{Ex}^{c,r}(P'/P'') \cup \mathrm{Start}^i(S) \cup \{r(x) :- node(x).\}$, $M' = \mathrm{Ex}^{c,f}(P') \cup \mathrm{Start}^i(S) \cup \{f(x) :- node(x).\}$, and $M'' = \mathrm{Ex}^{f,r}(P'') \cup \mathrm{Start}^i(R) \cup \{r(x) :- node(x).\}$. We can divide the top-down evaluation of the datalog query $?{-}i(x), c(x).\ M$ into two steps. First — the top-down evaluation of the datalog query $?{-}i(x), c(x).\ M'$ reaches the set $R = reached_{M',f}(i(x), c(x))$ of fact of the form $f(v)$. The second step is equivalent to the evaluation of the datalog query $?{-}i(x), f(x).\ M''$ where the set R is used as the set of starting nodes. By induction hypothesis, $R = [\![P']\!]_G(S)$. Therefore, the result of top-down evaluation $[\![i(x), c(x)]\!]_{M,db(G)}$ is equal to the join of the result of evaluation $[\![i(x), c(x)]\!]_{M',db(G)}$ and result of evaluation $[\![i(x), f(x)]\!]_{M'',db(G)}$. By induction hypothesis, $[\![i(x), c(x)]\!]_{M',db(G)} = \{[x/v] \mid v \in S,\ [\![P']\!]_G(\{v\}) \neq$

$\emptyset\}$ and $[\![i(x), f(x)]\!]_{M'',db(G)} = \{[x/v] \mid v \in R,\ [\![P'']\!]_G(\{v\}) \neq \emptyset\}$. Therefore:

$$\begin{aligned}
[\![i(x), c(x)]\!]_{M,db(G)} &= \{\sigma \bowtie \Pi_\emptyset(\sigma') \mid \sigma \in \{[x/v] \mid v \in S,\ [\![P']\!]_G(\{v\}) \neq \emptyset\},\\
&\qquad \sigma' \in \{[x/v, y/v'] \mid v \in S, (v,v') \in [\![P']\!]_G, [\![P'']\!]_G(\{v'\}) \neq \emptyset\}\}\\
&= \{[x/v] \mid v \in S, \exists v', v''.(v,v') \in [\![P']\!]_G, (v',v'') \in [\![P'']\!]_G\}\\
&= \{[x/v] \mid v \in S,\ [\![P'/P'']\!]_G(\{v\}) \neq \emptyset\}
\end{aligned}$$

Next, we show that the top-down visited sub-database is equal to $tdn_{G,S}(P'/P'')$. By induction hypothesis, we obtain $tdv_{M',db(G)}(i(x), c(x)) = tdn_{G,S}(P')$ and $tdv_{M'',db(G)}(i(x), f(x)) = tdn_{G,[\![P']\!]_G(S)}(P'')$. Therefore, $tdv_{M,db(G)}(i(x), c(x)) = tdn_{G,S}(P') \cup tdn_{G,[\![P']\!]_G(S)}(P'') = tdn_{G,S}(P'/P'')$.

Finally, we show that the set of all nodes v, such that the $r(v)$ is queried during the top-down evaluation of the datalog query $?{-}i(x), c(x).\ M$, is equal to $[\![P'/P'']\!]_G(S)$. The $r(v)$ can be queried only in the second step of the top-down evaluation which is equivalent to the evaluation of the datalog query $?{-}i(x), f(x).\ M''$. By induction hypothesis, $reached_{M'',r}(i(x), f(x)) = [\![P'']\!]_G([\![P']\!]_G(S))$. Thus: $reached_{M,r}(i(x), c(x)) = reached_{M'',r}(i(x), f(x)) = [\![P'']\!]_G([\![P']\!]_G(S)) = [\![P'/P'']\!]_G(S)$. □

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 $M = \mathrm{Acc}^{i,f}(P) \cup \mathrm{Start}^i(S)$ then $[\![f(x)]\!]_{M,db(G)} = \{[x/v] \mid v \in [\![P]\!]_G(S)\}$ and $tdv_{M,db(G)}(f(x)) = tdn_{G,S}(P)$.

Proof sketch
Let G be a graph with a node set V. The proof is induction on the structure of paths $P \in \mathscr{P}_\Sigma$. We only consider the case $P = P'/P''$. The definition of the compiler yields:

$$\mathrm{Acc}^{i,f}(P'/P'') = \mathrm{Acc}^{i,f'}(P') \cup \mathrm{Acc}^{f',f}(P'')$$

Let $M = \mathrm{Acc}^{i,f}(P'/P'') \cup \mathrm{Start}^i(S)$, $M' = \mathrm{Acc}^{i,f'}(P') \cup \mathrm{Start}^i(S)$ and $M'' = \mathrm{Acc}^{f',f}(P'') \cup \mathrm{Start}^{f'}(R).\}$. We can divide the top-down evaluation of the datalog query $?{-}f(x).\ M$ into two steps. irst — the top-down evaluation of the datalog query $?{-}f'.\ M'$ provides the set of nodes R equal to the set of all nodes v, such that $f'(v)$ is inferred after the first step of evaluation. The second step is equivalent to the evaluation of the datalog query $?{-}f(x).\ M''$ where the set R is used as the set of starting nodes. By induction hypothesis, $R = [\![P']\!]_G(S)$. Therefore, $[\![f(x)]\!]_{M,db(G)} = [\![f(x)]\!]_{M'',db(G)} = \{[x/v] \mid v \in [\![P'']\!]_G([\![P']\!]_G(S))\} = \{[x/v] \mid v \in [\![P'/P'']\!]_G(S)\}$. Next, we show that the top-down visited sub-database is equal to $tdn_{G,S}(P'/P'')$. By induction hypothesis, we obtain $tdv_{M',db(G)}(f'(x)) = tdn_{G,S}(P')$ and $tdv_{M'',db(G)}(f(x)) = tdn_{G,[\![P']\!]_G(S)}(P'')$. Therefore: $tdv_{M,db(G)}(f'(x)) = tdn_{G,S}(P') \cup tdn_{G,[\![P']\!]_G(S)}(P'') = tdn_{G,S}(P'/P'')$. □

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 $\mathscr{O}(|P||tdn_{G,S}(P)|)$.

7 Jumping in Graphs

Preprocessing is mandatory for sharing efforts when evaluating multiple queries on the same large graph. Most typically, one can pre-computate 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 to from any 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 $acc_a = \mathsf{edge}^*/a?$. 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 $1 \le j \le n$, 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 a very efficient automata-based algorithm proposed by Maneth and Nguyen (Maneth and Nguyen 2010) that evaluates navigational path queries on datatrees. More precisely, their algorithm covers navigational forwards XPath queries on XML documents, and is based on alternating tree automata with selection states (which can be seen a binary datalog programs, while ours are monadic). XML documents can be seen as labeled graphs, with two edge labels firstchild and nextsibling. Their algorithm can be based on indexes for jumping to a-labeled children, that is $\mathsf{edge}/a?$, and for jumping to top-most a-labeled descendants, i.e., $top_a = (\mathsf{edge}/\neg a?)^*/\mathsf{edge}/a?$. An XPath query such as `descendant::a` can the be rewritten as the 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 approach overcomes the main 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 NRPQs also with backward steps, and it can be implemented efficiently without any specialized or dedicated techniques.

It should be noticed that avoiding indexes of quadratic size may be relevant in practice, but more difficult to reach without restrictions. The index top_a, for instance, may be of the quadratic size, but only for XML documents that do not occur in practice. The choice of appropriate indexes raise many interesting research questions that are out of the scope of the present paper. It should also be mentioned that one may want to represent binary indexes in a more concise manner, rather than by enumeration of node pairs. For instance, for being able to jump to a-labeled nodes it is sufficient to store all a-labeled nodes, rather than pair of nodes (x,y) such that y is a-labeled.

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. It thus sets the stage for query optimization. In particular, the goto instruction permits algorithms 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 filters when it has bee proven correct. In our implementation, this effect may only be obtained if we use a datalog top-down implementation that follows the early completion strategy, i.e. stops whenever a ground predicate (such as filter queries in our case) is proven true. But this strategy does not survive magic-set rewriting of Datalog programs, in order to mimic top-down evaluation in a bottom-up manner.. Moreover, early completion does not allow us to define clearly a notion of needed nodes in a graph for a given query. A way out of this problem is to implement directly in datalog what it means to traverse a set of nodes sequentially. For this, we need to assume that outgoing edges in graphs are ordered. This local order extends to a total order on paths starting at a given node by using a lexicographic order. Then we may implement a depth-first left-to-right traversal of the graph following this lexicographic order.

We implementing our evaluation algorithm for NRPQs on datagraphs and started applying it the XPath queries from the XPathMark benchmark. The jumping algorithm with indexes is not yet done, so we cannot provide experimental results at the time being.

Acknowledgments The research was supported by the Russian Science Foundation grant 18-11-00100.

References

Arenas, M. and Pérez, J. 2011. Querying semantic web data with sparql. In Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems. 305–316.

Cleaveland, R. and Steffen, B. 1991. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. In Proceedings of the 3rd International Workshop on Computer Aided Verification. CAV '91. Springer-Verlag, Berlin, Heidelberg, 48–58.

Fischer, M. J. and Ladner, R. E. 1979. Propositional dynamic logic of regular programs. J. Comput. Syst. Sci. 18, 2, 194–211.

Gottlob, G., Koch, C., and Pichler, R. 2003. The complexity of XPath query evaluation. In 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. 179–190.

Libkin, L., Martens, W., and Vrgovc, D. 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, 129–140.

Maneth, S. and Nguyen, K. 2010. Xpath whole query optimization. PVLDB 3, 1, 882–893.

Martens, W. and Trautner, T. 2018. Evaluation and Enumeration Problems for Regular Path Queries. In 21st International Conference on Database Theory (ICDT 2018), B. Kimelfeld and Y. Amsterdamer, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 98. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 19:1–19:21.

Pérez, J., Arenas, M., and Gutiérrez, C. 2010. nSPARQL: A navigational language for RDF. J. Web Semant. 8, 4, 255–270.

Tekle, K. and Liu, Y. 2010. Precise complexity analysis for efficient datalog queries. In PPDP'10 - Proceedings of the 2010 Symposium on Principles and Practice of Declarative Programming. 35–44.

Ullman, J. D. 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, 140–149.

$tdn_G([P]) = tdn_G(P)$
$tdn_G(a) = \{\text{node}_a(v) \mid v \in V_a\}$
$tdn_G(F?) = tdn_G(F)$
$tdn_G(\text{edge}_a) = \{\text{edge}_a(v,v') \mid (v,v') \in E_a\}$
$tdn_G(\text{edge}_a^{-1}) = \{\text{edge}_a(v',v) \mid (v,v') \in E_a\}$
$tdn_G(\text{goto}(F)) = tdn_G(F)$

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

Fig. A 1. Top-down needed subgraphs without start sets as neeeded for goto expressions.

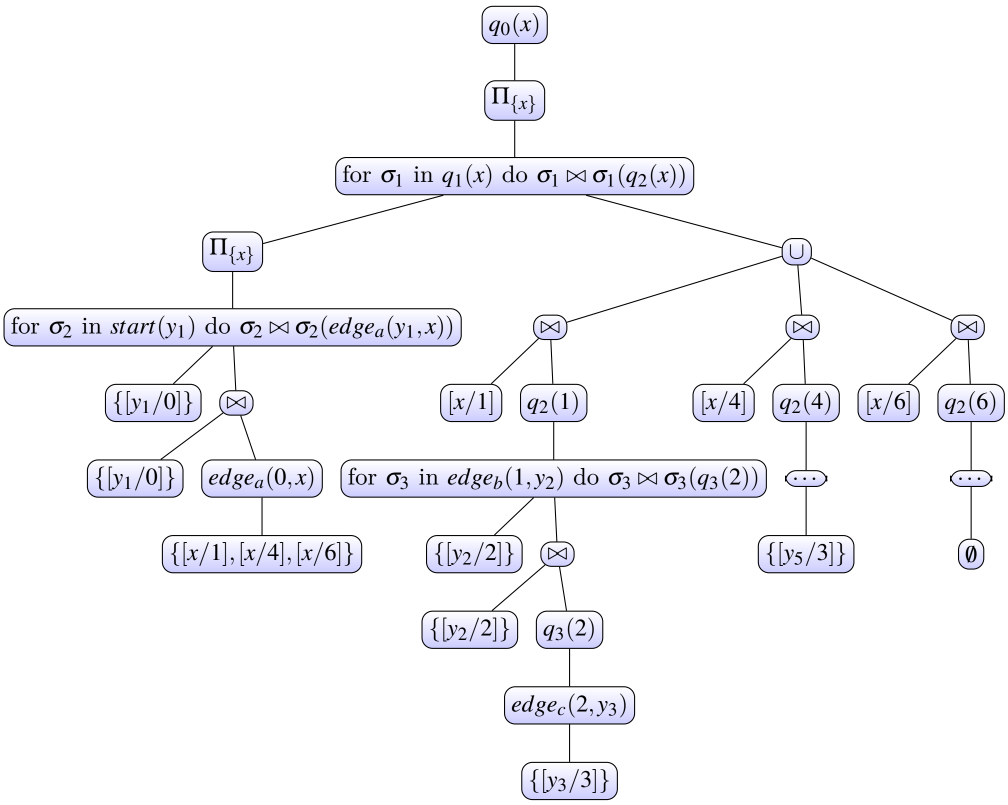

Fig. B 1. Top-down evaluation of $[\![q_0(x)]\!]_{M_0,db(G_0)\cup\{start(0)\}} = \{[x/1],[x/4]\}$ where M_0 is the datalog program from Fig. 1 corresponding to the path query $P_0 = \text{edge}_a[\text{edge}_b/\text{edge}_c]$, and G_0 the graph from Fig. 2.

Appendix A Proofs for Section 3 (Top-Down Needed Subgraphs)

In the case of goto expressions, we define in Fig. A 1 $tdn_{G,S}(\text{goto}(F)) = tdn_G(F)$ for restarting the computation with all nodes satisfying F.

Appendix B Proofs for Section 4 (Datalog Queries)

The whole top-down evaluation can always be represented as a join tree as we illustrate by example in Fig. B 1. In the case of infinite loops, the join tree is infinite.

Appendix C Proofs for Section 6 (Compiler to SCL Datalog Queries)

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 datalog programs $\mathrm{Start}^{i}(S)$, $\mathrm{Acc}^{i,f}(P)$, $\mathrm{Filt}^{c}(F)$, $\mathrm{Ex}^{c,r}(P)$ are safe and SLC.

Proof
Elementary by inspection of all cases of the definitions of these datalog programs. □

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 $\mathscr{O}(|P||tdn_{G,S}(P)|)$.

Proof
From Proposition 5, Proposition 4 and Theorem 2. □