HAL open science

Jumping Evaluation of Nested Regular Path Queries

Rustam Azimov, Joachim Niehren, Sylvain Salvati

To cite this version:

Rustam Azimov, Joachim Niehren, Sylvain Salvati. Jumping Evaluation of Nested Regular Path Queries. 2020. hal-02492780v1

HAL Id: hal-02492780

https://inria.hal.science/hal-02492780v1

Preprint submitted on 27 Feb 2020 (v1), last revised 4 Aug 2022 (v6)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Jumping Evaluation of Nested Regular Path Queries

Rustam Azimov[1,2], Joachim Niehren[3,4], and Sylvain Salvati[4,3]

[1] Saint Petersburg State University
[2] JetBrains Research
[3] Inria Lille
[4] Université de Lille

Abstract

Nested Regular Path Queries (NRPQs) are at the core of query languages for datagraphs, and part of query languages of graph databases and RDF triple stores. We propose a new jumping algorithm based on indexes that can evaluate NRPQs in graphs. Not only does it evaluate the query in combined linear time worst-case complexity, but it also does it by provably visiting only the *top-down needed* part of the graph for a given set of start nodes. We formally define this needed part with respect to the query and to the available indexes. Our algorithm is based on a new compilation schema from NRPQs to monadic datalog, such that the top-down evaluation of the datalog program visits only the needed subgraph. Thereby, it allows us 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. 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 NRPQ, and it can be implemented efficiently without any specialized or dedicated techniques, by simply using any efficient datalog evaluator.

1 Introduction

Regular path queries (RPQs) [7] 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) [5] extend on regular expressions by adding filters with logical operators, that may, in turn, contain regular path queries. They were first invented as the programs of propositional dynamic logic (PDL) [3], they constitute the navigational core of regular XPATH where they are restricted to query datatrees, and they are also part of nSPARQL for querying knowledge stores in the semantic Web [8].

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 but was first shown for the richer alternation-free modal μ-calculus [2]. However, this complexity upper bound alone is far too high in practice: the graph may be too large for a complete traversal. In general, only a fraction of the graph needs to be visited for computing the answer of a query. This fraction should be even smaller in the presence of indexes allowing the query to jump in the graph. 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 while using indexes for jumping. Now that combined linear complexity may be considered too expensive, we propose a jumping algorithm with combined linear complexity with respect to the top-down needed subgraph instead of the whole graph.

In a first step, we consider the same question without indexes for jumping. If we further restrict our attention to RPQs, that is to NRPQs without filter queries, a canonical notion of top-down needed nodes seems quite intuitive. These are all nodes reached when considering the

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

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

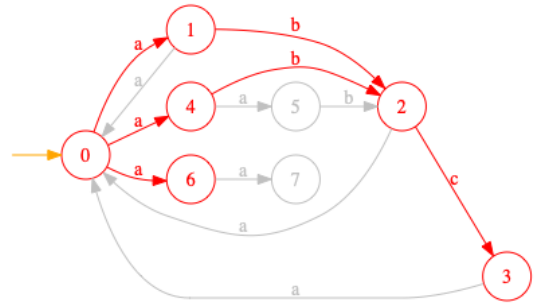

Figure 2: Graph G_0, start set $S_0 = \{0\}$, and the top-down needed subgraph for P_0 in red.

RPQ 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 coincides with what we call the top-down needed subgraph. Let us now consider the filter queries again. The notion of top-down needed nodes can then be lifted from RPQs 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 [8, 1, 4] achieve combined linear time complexity by pre-evaluating the filter queries all over the graph in a bottom-up manner and then running an evaluation algorithm for RPQs. Evaluating the filter queries 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 filter queries 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 Figure 2 with edge labels $\{a, b, c\}$, and set of starting 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 filter queries will first compute the set of nodes that satisfy the filter query $[\mathrm{edge}_b/\mathrm{edge}_c]$, 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 answers are in the intersection of both sets, which is $\{1, 4\}$. This algorithm, however, will inspect for the pre-evaluation of the filter queries some nodes and edges 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 Figure 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.

Our first contribution is an algorithm that answers path queries in the combined linear time $O(|G^{tdn}||P|)$ with respect to the top-down needed subgraph. For this, we present a linear time compilation scheme for mapping nested regular path queries P to datalog programs M. This compilation scheme is correct in that when it transforms a query P into a program M, then the set of nodes of a graph G that can be reached starting from nodes S with P is the set of nodes computed by the fixpoint of M with an extensional database that represents G and S. Furthermore, we can prove that the top-down needed subgraph G^{tdn} of the query P started with S is exactly the subgraph G^{vis} that the datalog program M visits when evaluated in the top-down manner with facts from G and S. Since the top-down evaluation of the datalog program can be done in time $O(|G^{vis}||M|)$ (see [9] or Theorem 1 below), it follows that the answer set of the query on the graph can indeed be computed in time $O(|G^{tdn}||P|)$.

In a second step, we reduce the full problem of answersing NRPQs with indexes to the case without. Here, the indexes are binary relation that are defined by NRPQs themselves. For us they come with the input, but usallay the were pre-computed elsewhere. 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 present such a jumping evaluation algorithm that will visit only the top-down needed nodes for an NRPQ on a graph with indexes. The notion of top-down neededness, however needs to be adapted so that it also depend on the indexes. The idea of the reduction is to add the indexes I to edge labels and the node pairs in the indexes to the graph as new edges labeled by I. Furthermore, the NRPQ is rewritten by substituting all occurences of I as a subqueries in the NRPQ by edge_I. We can the apply the previous machinery.

This approach can be used in particular to reformulate in simple terms a variant of a very efficient automata-based algorithm proposed by Maneth and Nguyen [6] that evaluates NRPQs on datatrees. 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 a 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.

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. Furthermore, any efficient semi-naive bottom-up evaluator will do the job, since we can rely on the magic set transformation to reduce top-down evaluation to bottom-up evaluation. The point here is that the magic set transformation of the datalog programs produced by our compilation schema can be done in linear time since these programs are monadic. For positive path queries, where the resulting datalog program is negation free, we can apply Ullman's Theorem 3 on p9 of [11]), showing that bottom-up evaluation after the magic set transform is more efficient than top-down evaluation. In the case with negation, we can use a recent generalization of this theorem to stratified datalog programs [10].

Outline. In Section 2 with present preliminaries on datalog while focussing on top-down evaluation. In Section 3, we recall the definition of NRPQs. In Section 4, we formally define the notion of top-down needed subgraphs for NRPQs. Our compiler from NRPQs to datalog is given in Section 5. Section 6 presents the jumping evaluation algorithm for NRPQs on graphs with indexes. The correctness proof is given in Appendix A.

2 Preliminaries on Datalog

This section gives a quick presentation of datalog's while focusing on complexity aspects of top-down evaluation.

Syntax. The vocabulary consists of a set $\mathcal{V}$ of variables ranging over x, y, z, etc, a set of $\mathcal{P}_{ext}$ of *extensional predicates* and a set $\mathcal{P}_{int}$ of *intensional predicates*. We assume that these three sets are pairwise disjoint. The set of all predicates is $\mathcal{P} = \mathcal{P}_{ext} \cup \mathcal{P}_{int}$. They will be ranged over by p, q, r, etc.

Extensional predicates are names for database relations, while intensional predicates are names of database queries, which define new relations from the given database relations. In our setting, extensional predicates will name the relations of the input graph, while intensional predicates will be used to define path queries in datalog.

We will use constants for the elements of a database. So let C be a set of constants that will be ranged over by a, b, c, etc. A *term* designs an element of a database. A term in $\mathcal{T}_C = \mathcal{V} \uplus C$ is either a variable or a constant and will be denoted by u, s, t, etc. A vector of terms is a word $\vec{t} \in \mathcal{T}_C^*$. The set of *positive literals* $\mathcal{L}^+$ with constants in C is the expressions $q(\vec{t})$ where $q \in \mathcal{P}$ and $\vec{t}$ in $\mathcal{T}_C^*$. Th set of *negative literals* $\mathcal{L}^-$ with constants in C is the set of expressions $\neg\ell$ with ℓ in $\mathcal{L}^+$. The set of all *literals* is then $\mathcal{L} = \mathcal{L}^+ \cup \mathcal{L}^-$. 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* or a *query* is a vector of literals $\vec{\ell} \in \mathcal{L}^*$, that is to be understood as a conjunction. We write $?{-}\vec{\ell}$ instead of $\vec{\ell}$ when we use a goal $\vec{\ell}$ as a query. The set of free variables of $fv(\vec{t})$, $fv(\vec{\ell})$ terms and goal vectors are defined as usual. Similarly for the sets of occuring constants $cst(\vec{t})$ and $cst(\vec{\ell})$. A datalog *clause* is a pair of the form $q(\vec{t}) :- \vec{\ell}.$ where q is an intensional predicate and $\vec{\ell} \in \mathcal{L}^*$. We call ℓ the *head* and $\vec{\ell}$ the *body* of the clause. The clause $q(\vec{t}) :- \vec{\ell}.$ is *safe* when all the variables occurring in $\vec{t}$ also occur in $\vec{\ell}$.

An *extensional database* (EDB) with constants in C is a subset of facts $E \subseteq \mathcal{L}_{ext}$ without variables. A *(safe) intensional database* with constants in C is a finite subset $I \subseteq \mathcal{L}_{int}$ of safe datalog clauses with constants in C. We only work with safe intensional databases and so all intensional database are assumed to be safe. A datalog program $M = E \cup I$ with constants in C is the union of a EDB E and an IDB I both with constants in C. The set $cst(M)$ is the set of constants that are used in the literals of M.

We use negation in our definition of datalog while adoption the usual restriction of *stratification*. Stratification of IDBs is a technical restriction that guarantees the existence of a fixpoint semantics for any EDB. An IDB I is called *stratified* if there is function stratum from $\mathcal{P}_{int}$ into $\mathbb{N}$ so that whenever there is a clause $q(\vec{t}) :- \vec{\ell}.$ in I so that $\neg p(\vec{t'})$ occurs in $\vec{\ell}$, then $\text{stratum}(q) > \text{stratum}(p)$. A datalog program $M = E \cup I$ is called stratified if I is stratified.

Semantics. We now turn our attention to the semantics of datalog programs. Given a query $?{-}\vec{\ell}$, an intensional database I and an extensional database E that form a datalog program M, we construct a set of substitutions that answer the query.

A *substitution* is a finite partial function σ from $\mathcal{V}$ to $\mathcal{T}_C$. 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 $\sigma : \mathcal{T}_C^* \to \mathcal{T}_C^*$ such that for all $n \geq 0$, $t_1, \ldots, t_n \in \mathcal{T}_C$ and $a \in C$:

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

Similarly, substitutions are lifted to total functions $\sigma : \mathcal{L}^* \to \mathcal{L}^*$ such that for all $\vec{t} \in \mathcal{T}_C^*$, $n \geq 0$, and $\ell_1, \ldots, \ell_n \in \mathcal{L}$:

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

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:

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

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 $V \subseteq \mathcal{V}$:

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

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 $[\![\vec{\ell}\,]\!]_M$ of a datalog goal $?{-}\vec{\ell}$ with respect to a datalog program M made with an IDB I and an EDB E, so that it is defined as the usual fixpoint semantics in presence of stratified negation that satisfy the following equations for all $n \geq 2$, $\ell \in \mathcal{L}^+$, and $\ell_1, \ldots \ell_n \in \mathcal{L}$:

$$[\![\epsilon]\!]_M = \{[]\}$$
$$[\![\ell]\!]_M = \{\Pi_{fv(\ell)}(\sigma \bowtie \sigma') \mid \sigma = unif(\ell, \ell'),\ \ell' :- \vec{\ell}. \text{ in } ren(I),\ \sigma' \in [\![\sigma(\vec{\ell})]\!]_M\} \text{ if } \ell \in \mathcal{L}_{int}$$
$$[\![\ell]\!]_M = \{\Pi_{fv(\ell)}(\sigma) \mid \sigma = unif(\ell, \ell'),\ \ell' \in E\} \text{ if } \ell \in \mathcal{L}_{ext}$$
$$[\![\neg\ell]\!]_M = \{\sigma \mid dom(\sigma) = fv(\ell) \wedge \sigma \notin [\![\ell]\!]_M\}$$
$$[\![\ell_1 \ldots \ell_n]\!]_M = \{\sigma' \bowtie \sigma \mid \sigma \in [\![\ell_1]\!]_M,\ \sigma' \in [\![\sigma(\ell_2 \ldots \ell_n)]\!]_M\}$$

Notice that we use the $\bowtie$ operation only in trivial circumstances: each time the semantics computes $\sigma \bowtie \sigma'$, we have $dom(\sigma) \cap dom(\sigma') = \emptyset$. Moreover, as we only work with safe datalog programs, every substitution in $[\![\vec{\ell}]\!]_M$ maps every variable free in $\vec{\ell}$ to a constant.

The definition of semantics that we have given mimics top-down evaluation algorithms for datalog. In particular, we can represent the top-down evaluation tree as a join tree induced by the definition (see Figure 3). Top-down evaluation starts with a given query and generates subqueries using the clauses of the datalog program until it reaches ground facts from the extensional database. In general, this process may enter into infinite loops. This problem is avoided in datalog by using memoization: each time a (variant up to renaming of variables of a) subquery is evaluated in the process it is tabled allowing to reuse computation and detect loops. Making the assumption that this memoization allows us to store and access each data in constant time (for example, by using hash tables, or arrays if constants can be aliased to integers), then the complexity of executing top-down evaluation is linear in the number of solutions of subqueries it finds (see [9]). Here a solution of a subquery is understood as the instantiation of a prefix of the body of a clause or of the query during the evaluation. In top-down evaluation, negated goals are then matched with failure to obtain the fact. In that case, this evaluation mechanism computes the stratified semantics mentioned above.

Complexity. As we use datalog as a way of describing our algorithm, the complexity analysis of our algorithm requires the evaluation of the number of subqueries generated by top-down

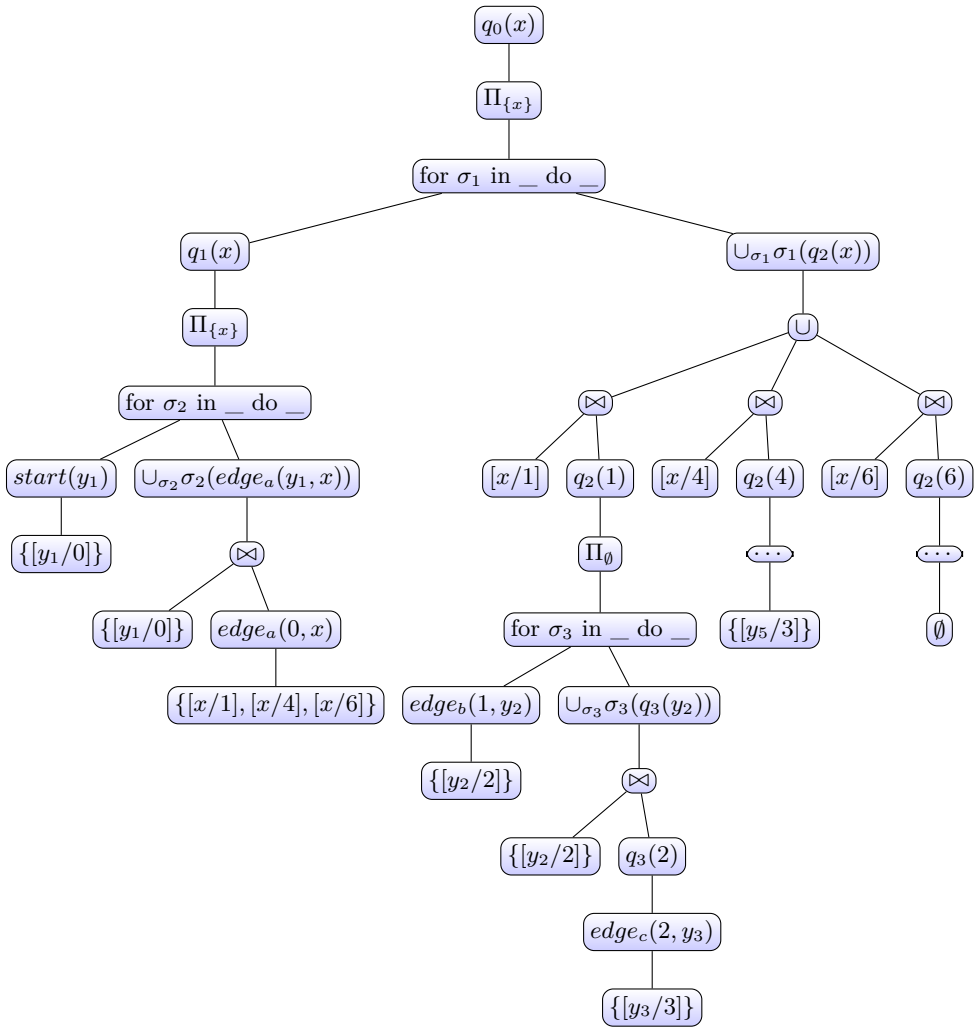

Figure 3: Top-down evaluation of $[\![q_0(x)]\!]_M = \{[x/1],[x/4]\}$ where M is the datalog program from Figure 3 corresponding to the path query $P_0 = \text{edge}_a[\text{edge}_b/\text{edge}_c]$ with EDB obtained by graph G_0 from Figure 2 and the starting datalog program $\{start(0).\}$.

evaluation of its datalog implementation. A particular property of the datalog program we generate for our query and graph is that every intensional predicate is monadic. However, even though there is always a way of evaluating the monadic datalog program in combined linear time, it is not the case that the top-down evaluation of a monadic datalog program is performed in combined linear time. Moreover, for us, combined linear time is not enough, we wish for an algorithm that runs in combined linear time not with respect to the complete EDB but rather with a notion of needed EDB. The work of [9] gives us the formal tools so as to prove the complexity properties of the top-down evaluation of our program is in combined linear time with respect to the needed EDB.

A *full extensional database* (FEDB) is a pair $F = (V, E)$ such that E is an EDB with constants in V. The size $|F|$ of F is the sum of the cardinalities of V and E. Given two FEDBs

$F_1 = (V_1, E_1)$ and $F_2 = (V_2, E_2)$ we write $F_1 \oplus F_2$ for the *union of* F_1 *and* F_2 defined by $(V_1 \cup V_2, E_1 \cup E_2)$. Given a set $\mathcal{F}$ of FEDBs, we write $\bigoplus \mathcal{F}$ for the union of the FEDBs in $\mathcal{F}$.

We next define the FEDB vis_M visited by the top-down evaluation of a query $\vec{\ell}$ for datalog program M. Given a stratified program $M = E \cup I$ and a query $\vec{\ell}$, we define $vis_M(\vec{\ell})$ as the FEDB visited by the top-down evaluation of $\vec{\ell}$. It computed as a fixpoint based on the stratified semantics (here we assume that $\ell \in \mathcal{L}^+$, $\vec{\ell} \in \mathcal{L}^*$ and ℓ_1,...,$\ell_n \in \mathcal{L}$):

- $vis_M(\epsilon) = (\emptyset, \emptyset)$
- $vis_M(\ell) = (cst(\ell), \emptyset) \oplus \bigoplus \left\{ vis_M(\sigma(\vec{\ell})) \mid \sigma = unif(\ell, \ell'),\ \ell' :- \vec{\ell}. \text{ in } ren(I) \right\}$ if $\ell \in \mathcal{L}_{int}$,
- $vis_M(\ell) = (cst(\ell), \emptyset) \oplus \bigoplus \{(cst(\ell'), \{\ell'\}) \mid unif(\ell, \ell') \text{ defined}, \ell'. \text{ in } E\}$ if $\ell \in \mathcal{L}_{ext}$,
- $vis_M(\neg\ell) = vis_M(\ell) \oplus (cst(M), \emptyset)$,
- $vis_M(\ell_1 \dots \ell_n) = vis_M(\ell_1) \oplus \bigoplus \{vis_M(\sigma(\ell_2 \dots \ell_n)) \mid \sigma \in [\![\ell_1]\!]_M\}$

Now [9] gives us sufficient conditions for a datalog program M with IDB I and EDB E to have a combined linear complexity in I and in $vis_M(\vec{\ell})$ when solving a query $?{-}\vec{\ell}$. The main remark in [9] that we use is that provided the EDB can be queried in constant time, the complexity of the top-down evaluation depends on the number of possible instantiations of left prefixes of bodies of clauses or of the initial query. We say that a sequence of literals $\vec{\ell}$ is *simply combined linear* (SCL) when it either uses only one variable or when it only uses one extensional predicate and every variable occurring in $\vec{\ell}$ are the argument of that extensional predicate. For example, if p, q are intensional monadic predicates, and a is a binary extensional predicate, then $p(x), a(x, y), q(y)$ is SCL as both x and y are arguments of the extensional predicate a. The sequence $p(x), a(x, x), q(y)$ is on the contrary not SCL as it contains 2 variables and one of them, namely y, does not occur as an argument an extensional predicate. An SCL sequence has a number of instantiations that is linear in the size of the EDB, and even in the size of the visited hypergraph as in case it contains only one variable, it can only be instantiated by one of the visited nodes and when every variable are arguments of an extensional predicate, then the instantiations of the sequence need to make the extensional predicate equal to one of the facts in the visited hypergraph. Now we call an IDB (or a datalog program built with this IDB) *simply combined linear* (SCL) when for each clause $\ell :- \vec{\ell}$ the sequence $\ell\vec{\ell}$ is SCL.

Theorem 1 ([9])**.** *Given a safe stratified SCL datalog program M with IDB I and EDB E and an SCL sequence $\vec{\ell}$, the top-down evaluation of the query $?{-}\vec{\ell}$ is in time $\mathcal{O}(|I||vis_M(\vec{\ell})|)$ where $|I|$ is the size of I.*

It is easy to check that every clause we use in our compilation scheme is SCL and that the queries we evaluate are also SCL. (see Section 5).

3 Nested Regular Path Queries

We now recall the syntax and semantics of NRPQs for digraphs with labeled edges and nodes.

We start from a finite set of labels Σ. A (finite) Σ-labeled digraph is a pair $G = ((V_a)_{a\in\Sigma}, (E_a)_{a\in\Sigma})$ where $V = \uplus_{a\in\Sigma} V_a$ is a finite set of nodes and $E = \cup_{a\in\Sigma} E_a \subseteq V \times V$ a finite set of edges. The nodes in V_a and the edges in E_a are called labeled by a.

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

Figure 4: Semantics of NRPQs on a labeled digraph $G = (V, E)$ where v, v' are nodes from V.

The syntax of NRPQs for Σ-labeled graphs provides of a set of filter queries $\mathcal{F}_\Sigma$, that will denote a subset of nodes of the graph, and a set of nested regular path queries $\mathcal{P}_\Sigma$ that will denote a set of pairs of the graph nodes.

$$
\begin{array}{llll}
\text{filter queries} & F \in \mathcal{F}_\Sigma & ::= & [P] \mid a \mid \mathit{true} \mid F \wedge F' \mid F \vee F' \mid \neg F \\
\text{path queries} & P \in \mathcal{P}_\Sigma & ::= & F? \mid \mathrm{edge}_a \mid \mathrm{edge}_a^{-1} \mid P/P' \mid P \cup P' \mid P^+ \mid \mathit{goto}(F)
\end{array}
$$

Note that we can express path queries with Kleene stars by defining $P^* =_{df} P^+ \cup \mathit{true}?$ but we prefer to keep its non-reflexive version as a primitive. We will also define the path query edge as a shortcut for $\cup_{a \in \Sigma} \mathrm{edge}_a$? An example for an NRPQ with signature $\Sigma = \{a, b, c\}$ is the path query:

$$P_2 = a?/(\mathrm{edge}^+/[\mathrm{edge}_b/\mathrm{edge}_c]?)^*$$

The evaluation of this query on a given graph will start on a given start node of the graph, test whether the start node is labeled by a, and if so, it will navigate 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 in this way is returned by the query.

The semantics of NRPQs P on a labeled digraph G is defined in Figure 4. It is a binary relation $[\![P]\!]_G \subseteq V \times V$, that is defined by mutual recursion and the semantics of filter queries F by a monadic relation $[\![F]\!]_G \subseteq V$. We will use path queries for defining monadic queries, by fixing some set of start nodes and then querying for the set of target nodes reachable from over the path. Similarly, filter queries may be restricted to nodes of a given start set.

Definition 1. *Let G be a labeled graph with node set V and $S \subseteq V$. For any* NRPQ *P and filter query F we define a node set as follows:*

- $[\![P]\!]_G(S) = \{v \mid \exists v' \in S.\ (v', v) \in [\![P]\!]_G\}$.
- $[\![F]\!]_G(S) = [\![F]\!]_G \cap S$

It is worth mentioning, that our language of NRPQs can express backward path queries $\mathit{inv}(P)$ for all path queries P, up to a linear time transformation defined in Figure 5. For this, it is sufficient to push the inversion operator down to edges. Thereby, we clearly have $[\![\mathit{inv}(P)]\!]_G = [\![P]\!]_G^{-1}$ is the inverse of the binary relation $[\![P]\!]_G$.

4 Top-Down Needed Subgraphs

Intuitively NRPQs define graph traversals: path queries give possible directions to follow and filter queries check the existence of node labels and of certain paths that can be followed starting

$$\begin{array}{ll} inv(P^+) = inv(P)^+ & inv(P'/P'') = inv(P'')/inv(P') \\ inv(\text{edge}_a) = \text{edge}_a^{-1} & inv(P' \cup P'') = inv(P') \cup inv(P'') \\ inv(\text{edge}_a^{-1}) = \text{edge}_a & inv(F?) = F? \\ & inv(goto(F)) = F?/goto(true) \end{array}$$

Figure 5: Backwards path queries $inv(P)$.

at a node. Following this intuition, we associate to any NRPQ P and labeled digraph $G = ((V_a)_{a\in\Sigma}, (E_a)_{a\in\Sigma})$ a notion of *top-down visited subgraph* (we let V be $\bigcup_{a\in\Sigma} V_a$). Interestingly, this notion is similar to the notion of visited FEDB for the top-down evaluation of a datalog program. For the sake of simplicity, we present the top-down needed subgraph as a hypergraph. It makes it easier for us to connect the notions when it comes to technical work. For a path query P and a set of start nodes this FEDB is denoted by $G^{tdn}(P, S)$. In database algorithms, operations often have an asymmetric behavior. In our case, when we use the $goto(F)$ operator, we know that we can jump anywhere in the graph. However, when F is equal to $[edge_a/P]$ for some P, computing for each node x in the graph the nodes y so that (x, y) is in E_a has cost in $\Theta(|V|+|E_a|)$ while just scanning E_a has a lower cost: $\Theta(|E_a|)$. So as to have a better complexity we introduce a symbolic notation $\top$ that represents symbolically V. The only moment when we know for sure that the set of starting nodes is V and that we may use such an algorithm is when we use the *goto* command. In this case, we use $\top$ as the set of starting nodes so as to pass the information that we can use the more efficient algorithm. Note that it may be the case that the set of starting nodes is equal to V by chance. In that case we would need to check this fact so as to use the optimization. This checking would result in a $|V|$ overhead in the computation so that we will not do it.

We take the convention that in set operations, $\top$ behaves like V. For example, we assume that $S \cap \top = \top \cap S = S$ and that if R is a binary relation then $R(\top)$ is simply $R(V)$. The only place where $\top$ differs from V is when it comes to define the top-down needed FEDB. The place where it makes a difference is when it comes to extensional predicates.

For this, given a set of starting nodes S, we introduce the notation $\overline{S}$ which maps $\top$ to $\emptyset$ and S to S otherwise. Furthermore, given a set of facts E we write $cst(E)$ to denote the set of constants in E. Now given a set of starting nodes S, we let $\mathcal{E}_{\text{edge}_a}(S)$, $\mathcal{E}_{\text{edge}_a^{-1}}(S)$ and $\mathcal{E}_{node_a}(S)$ be the following EDBs:

- $\mathcal{E}_{\text{edge}_a^{\pm 1}}(S) = \begin{cases} \{\text{edge}_a(v_1, v_2) \mid (v_1, v_2) \in E_a\} \text{ when } S = \top \\ \{\text{edge}_a((s, v)^{\pm 1}) \mid s \in S \wedge (s, v)^{\pm 1} \in E_a\} \text{ otherwise} \end{cases}$,
- $\mathcal{E}_{node_a}(S) = \begin{cases} \{node_a(v) \mid v \in V_a\} \text{ when } S = \top \\ \{node_a(s) \mid s \in S\} \text{ otherwise.} \end{cases}$.

We now define $\mathcal{H}_{\text{edge}_a}(S)$, $\mathcal{H}_{\text{edge}_a^{-1}}(S)$ and $\mathcal{H}_{node_a}(S)$ be the following FEDB:

- $\mathcal{H}_{\text{edge}_a^{\pm 1}}(S) = (\overline{S} \cup cst(\mathcal{E}_{\text{edge}_a^{\pm 1}}(S)), \mathcal{E}_{\text{edge}_a^{\pm 1}}(S))$,
- $\mathcal{H}_{node_a}(S) = (\overline{S} \cup cst(\mathcal{E}_{node_a}(S)), \mathcal{E}_{node_a}(S))$.

This definition is so that $\mathcal{H}_{\text{edge}_a}(\top)$ the nodes which that have an ingoing or outgoing edge labeled a and the corresponding edges; while the definition of $\mathcal{H}_{\text{edge}_a}(S)$ (with $S \neq \top$) contains all the nodes in S and those that are related to some nodes in S by an edge labeled a and only these edges.

$$
\begin{array}{ll}
G^{tdn}(\neg F, S) = G^{tdn}(F, S) \oplus (S, \emptyset) & \\
G^{tdn}([P], S) = G^{tdn}(P, S) & G^{tdn}(F' \wedge F'', S) = G^{tdn}(F', S) \oplus G^{tdn}(F'', [\![F']\!]_G \cap S)) \\
G^{tdn}(a, S) = \mathcal{H}_{node_a}(S) & G^{tdn}(F' \vee F'', S) = G^{tdn}(F', S) \oplus G^{tdn}(F'', [\![\neg F']\!]_G \cap S)) \\
G^{tdn}(true, S) = (S, \emptyset) & G^{tdn}(P'/P'', S) = G^{tdn}(P', S) \oplus G^{tdn}(P'', [\![P']\!]_G(S)) \\
G^{tdn}(F?, S) = G^{tdn}(F, S) & G^{tdn}(P^+, S) = G^{tdn}(P, [\![P^+]\!]_G(S) \cup S) \\
G^{tdn}(\mathrm{edge}_a, S) = \mathcal{H}_{\mathrm{edge}_a}(S) & G^{tdn}(P' \cup P'', S) = G^{tdn}(P', S) \oplus G^{tdn}(P'', S) \\
G^{tdn}(\mathrm{edge}_a^{-1}, S) = \mathcal{H}_{\mathrm{edge}_a^{-1}}(S) & G^{tdn}(goto(F), S) = G^{tdn}(F, \top)
\end{array}
$$

Figure 6: Top-down needed subgraphs for path and filter queries.

Definition 2. *Let G be a labeled digraph with node set V, and $S \subseteq V$. The* top-down needed subgraph *that must be visited for evaluating a path query P (resp. filter query F) on graph G with set of starting nodes S is the hypergraph $G^{tdn}(P, S)$ (resp. $G^{tdn}(F, S)$) defined Figure 6.*

Note that the definitions of $G^{tdn}(P, S)$ and $G^{tdn}(F, S)$ in Figure 6 follow the intuitive traversal induced by P or F in G. We would like to comment a bit on certain parts of the definition. The definitions of the $\wedge$ and $\vee$ for filter queries make these operators non-commutative. This is because we consider them *sequential*. When the filter query F' is failing for a node s 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 s. Similarly when the filter query F' succeeds for s, we know that the filter query $F' \vee F''$ also succeeds for s without having to evaluation F'' at s. Of course, a parallel semantics for these operators and visit a larger subgraph with the possibility of being more time-efficient thanks to parallelism. We wish however to show that we can visit an as small part of the graph as possible and so opt for sequential logical operators. The definition of $G^{tdn}(P^+, S)$ 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^+. Finally, the definition of $G^{tdn}(goto(F), S)$ simply is jumping at any node of the graph and then checking whether F holds at that node. Finally, $\top$ is only introduced when the *goto* operator is used and is then propagated through the query by the definition.

Example 1. *For the query $P_0 = \mathrm{edge}_a[\mathrm{edge}_b/\mathrm{edge}_c]$, the graph G_0 in Figure 2 and the set of starting nodes $\{0\}$ we have $G_0^{tdn}(P_0, \{0\}) = (\{0, 1, 2, 3, 4, 6\}, (E_l)_{l \in \{a,b,c\}})$ where $E_a = \{\mathrm{edge}_a(0, 1), \mathrm{edge}_a(0, 4), \mathrm{edge}_a(0, 6)\}$, $E_b = \{\mathrm{edge}_b(1, 2), \mathrm{edge}_b(4, 2)\}$, and $E_c = \{\mathrm{edge}_c(2, 3)\}$). In Figure 2 the top-down needed subgraph is highlighted in red. Note that we did not talk about node labels in this example, but we could chose that all nodes of G_0 belong to V_a while $V_b = V_c = \emptyset$.*

5 Compiler to Monadic Datalog

In this section we describe how NRPQs are compiled to IDBs and then NRPQs, graphs and sets of starting nodes to datalog programs and queries that allow us to evaluate efficiently NRPQs. Let's first describe how we represent graphs as EDBs. Given a labeled digraph $G = ((V_a)_{a \in \Sigma}, (E_a)_{a \in \Sigma})$ we transform it into the EDB $M(G)$ that fixes the semantics of the extensional predicates. The EDB $M(G)$ has the set of constants $C_G = \bigcup_{a \in \Sigma} V_a$ and the set of predicates $\mathcal{P}_G = \{node^{(1)}\} \cup \{node_a^{(1)}, \mathrm{edge}_a^{(2)} \mid a \in \Sigma\}$ where all predicates *node* and $node_a$ are monadic, while all predicates edge_a are binary. The predicate *node* is meant to capture the set of all nodes in the graph. We need this predicate for technical reasons so as to make our datalog programs safe (see Section 2). The predicate $node_a$ is meant to capture the set of nodes labeled

with a and the predicate edge_a is meant to capture the edges labeled a as a binary relation. We thus let:

$$\begin{array}{rcl} M(G) & = & \{node(v). \mid a \in \Sigma,\ v \in C_G\} \\ & \cup & \{node_a(v). \mid a \in \Sigma,\ v \in V_a\} \\ & \cup & \{\text{edge}_a(v, v'). \mid a \in \Sigma,\ (v, v') \in E_a\} \end{array}$$

Now we see how we represent the set of start nodes. Recall that the set of start nodes can either be as subset of the nodes of G or the symbol $\top$. However we only give a representation of the start nodes when they are different from $\top$. So given such a set S and a monadic predicate i we let $M^i(S)$ be the EDB that is equal to $\{i(s) \mid s \in S\}$.

The compilation scheme for queries follows the structure of the queries by combining rules generated by immediate subformulae and possibly adding some of new rules. The compilation scheme is based on three different mutually recursive definitions given by Figures 7, 8 and 9.

The first definition is that of $Acc^{i,f}(P)$ (Figure 7). This computed IDB is used in programs combining a set S of start nodes and a graph G. With a suitable query this program computes the following set of nodes $\{v \mid \exists s \in S. (s, v) \in [\![P]\!]_G\}$, i.e. the set of nodes reachable from S with a path described by P. The superscripts i and f in the definition are particular monadic predicates of the IDB. The predicate i is meant to capture the set of starting nodes while the predicate f is meant to represent the answer set of the path query P. Then the datalog program $M^{i,f}_{Acc}(P, G, S)$ that evaluates the path query P on a graph G with starting nodes S, is defined as:

$$M^{i,f}_{Acc}(P, G, S) = Acc^{i,f}(P) \cup M(G) \cup M^i(S)\ .$$

The program then computes the answer set with when evaluating the query $?{-}f(x)$. In this case, we consider that S cannot be $\top$ as path queries that are inside filters, and thus may only appear in a *goto*, are treated by the $Ex^{c,r}(P)$ transformation.

The superscripts i and f play a central role in the compilation scheme as they make the various parts of the generated IDB communicate with each other. Consider for example the path query P'/P'', in that case we have $Acc^{i,f}(P'/P'') = Acc^{i,f'}(P') \cup Acc^{f',f}(P'')$. Here the predicate f' represents the answer set of P' and is representing the start nodes of the query P''. This is simply because the start nodes of P'' in the query P'/P'' are the nodes that are reached with the query P'. Something similar can be observed for the query P^+ for which $Acc^{i,f}(P^+) = Acc^{i,f}(P) \cup \{i(x) :- f(x).\}$. 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. In the other cases, the definition of $Acc^{i,f}(P)$ is as expected except maybe for the case of *goto*. Here the definition is $Acc^{i,f}(goto(F')) = Filt^{f'}(F') \cup \{f(x) :- j(), f'(x).\quad j() :- i(x).\}$ where $Filt^{f'}(F')$ is the IDB that represents the computation of the filter query F' and where f' is the predicate which captures the answer set. One might have expected to have the simpler definition of the form $Filt^{f'}(F') \cup \{f(x) :- i(y), f'(x).\}$ where the rule $f(x) :- i(y), f'(x).$ would mean that a node x is in the answer set when there is some node in the starting set thanks to the existence of a node y for which $i(y)$ holds and so that $f'(x)$ holds meaning that x is in the extension of F'. However, such a rule is not SCL (see Section 2) and could have quadratic behavior. We therefore make the projection earlier by using a fresh nullary predicate j which is true only when there is a y which is a starting node. Now the two rules are SCL and will thus satisfy our requirement for complexity.

As mentioned above, $Filt^c(F)$ (Figure 8) transforms a filter query F into an IDB where the monadic predicate c represent the subset of the start nodes for which the filter query F holds. The IDB associated to a filter query F may be executed in different contexts either simply as a filter query originating for example from a path query F? or when it occurs inside

a *goto*. In the first case the set of start nodes is fixed and is that of the path query $F?$ and the evaluation of the query has to check whether the $c(v)$ is true for every start node v. In the second case the set of start nodes is represented by $\top$ and in the top-down evaluation of the datalog program there will be only one subquery for $c(x)$ asking which are the nodes for which F holds. These two different behaviors account for the particular definition of the top-down needed FEDB given Section 4 for *goto* path queries. When seeking those nodes that satisfy F, the top-down evaluation will not necessarily inspect each nodes of the graph. When a filter query is compiled to a program $M^{i,c,g}_{Filt}(F, G, S)$ with a graph G and starting nodes S, there are two cases depending on whether $S = \top$:

- $M^{i,c,g}_{Filt}(F, G, \top) = Filt^c(F) \cup M(G) \cup \{g(x) :- c(x).\}$,
- $M^{i,c,g}_{Filt}(F, G, S) = Filt^c(F) \cup M(G) \cup M^i(S) \cup \{g(x) :- i(x), c(x).\}$ when $S \neq \top$.

Here g is the predicate that capture the answer of the query, and the query associated with the program is $?{-}g(x)$. The monadic predicate i is that of starting nodes, while c is the predicate that captures the semantics of F. In case the starting nodes are represented by $\top$, the predicate i is not used and g is equivalent to c. Otherwise, when $S \neq \top$, we add the EDB $M^i(S)$ that represents the starting nodes and g is guarded by the predicate i so that the program only tries to find solutions for c that are among the starting nodes.

We are now going to have a closer look at certain definitions. First, the filter query *true* is meant to hold at every node, so it is compiled as follows $Filt^c(true) = \{c(x) :- node(x)\}$. Remark that we use the extensional predicate *node* here so as to make the clause safe. Second, the definition of the disjunction of filters looks a bit complicated as it needs to accommodate the sequential semantics we chose. This sequential treatment is captured by the two rules $c(x) :- c'(x).$ and $c(x) :- \neg c'(x), c''(x).$ As only $c'(x)$ or $\neg c'(x)$ can be true, only one of the two rules can lead to a conclusion. Moreover, memoization in datalog makes it so that the computation concerning $c'(x)$ is shared between $c'(x)$ and $\neg c'(x)$. Finally as clauses are evaluated from left to right the goal $c''(x)$ in the second clause is evaluated only when $\neg c'(x)$ is solved. As you can see, this left to right order of evaluation of clause makes the definition of the sequential evaluation for the conjunction easier to obtain.

We now turn to the last part of the compilation scheme $Ex^{c,r}(P)$ (Figure 9) which deals with path queries. Given a set S of start nodes and a graph G, the generated IDB is used to compute the following set of nodes $\{s \in S \mid \exists v. (s, v) \in [\![P]\!]_G\}$, i.e. the set of nodes of S for which there is some path described by P that starts from them. The set of those nodes is captured by the predicate c while the predicate r captures the nodes that are reached with a path described by P during the search process. The predicate r is required to allow us to compile path concatenation. The compilation scheme $Ex^{c,r}(P)$ is used for path queries which occur within some filter queries. It is typically called when compiling the filter query $P?$. As expected, in that case, the set of start nodes may be $\top$. So given a set of start nodes S, a graph G and a path query P, we compile build the following program:

- $M^{i,c,r,g}_{Ex}(P, G, \top) = Ex^{c,r}(P) \cup M(G) \cup \{g(x) :- c(x).\} \cup \{r(x) :- node(x).\}$
- $M^{i,c,r,g}_{Ex}(P, G, S) = Ex^{c,r}(P) \cup M(G) \cup M^i(S) \cup \{g(x) :- i(x), c(x).\} \cup \{r(x) :- node(x).\}$ when $S \neq \top$

Here the predicates c and r are the monadic predicates of the IDB $Ex^{c,r}(P)$ while i and g, as for filter queries, respectively stand for the set of start nodes and the set of answers of the query. The case where the set of start nodes is $\top$ is treated similarly to when we dealt with filters. The main particularity come from the treatment of the predicate r for which we add a clause

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

Figure 7: The IDB $Acc^{i,f}(P)$ for path query P with initial and final predicates i resp. f. The intermediate predicates f', f'', i', i'', j must be fresh.

$$
\begin{aligned}
&Filt^c(a) = \{c(x) :- node_a(x).\}\\
&Filt^c(true) = \{c(x) :- node(x).\}\\
&Filt^c(F' \vee F'') = Filt^{c'}(F') \cup Filt^{c''}(F'') \cup \{c(x) :- c'(x). \quad c(x) :- \neg c'(x), c''(x).\}\\
&Filt^c(F' \wedge F'') = Filt^{c'}(F') \cup Filt^{c''}(F'') \cup \{c(x) :- c'(x), c''(x).\}\\
&Filt^c(\neg F') = Filt^{c'}(F') \cup \{c(x) :- \neg c'(x).\}\\
&Filt^c([P]) = Ex^{c,r}(P) \cup \{r(x) :- node(x).\}
\end{aligned}
$$

Figure 8: The IDB $Filt^c(F)$ for filter query F with predicate c. The intermediate predicates c', c'', r must be fresh.

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

Figure 9: The IDB $Ex^{c,r}(P)$ for path queries P with predicates c, r. The intermediate predicates f, c', j must be fresh.

$r(x) :- node(x)$. whose meaning is to capture that when a node r is reached after reading a path described by P from a starting node, then we have completed our search. Here again we use the monadic predicate *node* so as to make the clause safe. In the definition of $Ex^{c,r}(P)$ the interplay between the predicates c and r is similar to the one between i and f in $Acc^{i,f}(P)$.

First of all, we need to remark that the datalog program we compute all satisfy certain properties: safety, stratification and SLC.

Lemma 1. *For every path query P, filter query F, graph G and set of start nodes S, the datalog programs, $M^{i,f}_{Acc}(P,G,S)$, $M^{i,c,g}_{Filt}(F,G,S)$ and $M^{i,c,r,g}_{Ex}(P,G,S)$ are safe, stratified and SLC.*

When compiling a query P or F, the generated program is stratified since in every clause involving negation, the negated predicate is associated to a subformula of P or F that is lower than the one to which is associated the head of the clause. Thus stratification is induced by

the structure of formulae.

A next property we need is that the program computes the right set of nodes and in the right way, i.e. by only visiting the needed FEDB. Concerning this last property we meet a small technical problem. So as to ensure safety we use the extensional predicate *node* which is note used in the top-down visited FEDB. Thus we cannot have the equality between the top-down visited FEDB by the path or filter query and the top-down visited FEDB by the top-down evaluation of the program. We also have a similar problem with respect to starting nodes when the predicate i is considered extensional with the EDB $M^i(S)$. So as to obtain an equality we choose to remove the occurrences of the *node* and i predicates the FEDB of the top-down evaluation of the program. As the set of nodes that are concerned by the predicate *node* and i are in the domain of the FEDB, it does not change significantly its size. So given an FEDB E we write $node_i^-(E)$ for the FEDB obtained by removing the *node* and i facts. We now can formulate our correctness theorem. The proof of the Theorem 2 can be found in Appendix A.

Theorem 2. *For path query P, filter query F, labeled graph G, set S of start nodes, we have:*

- $[\![f(x)]\!]_{M^{i,f}_{Acc}(P,G,S)} = \{[x/v] \mid v \in [\![P]\!]_G(S)\}$ *and* $node_i^-(vis_{M^{i,f}_{Acc}(P,G,S)}(f(x))) = G^{tdn}(P,S)$,
- $[\![g(x)]\!]_{M^{i,c,g}_{Filt}(F,G,S)} = \{[x/v] \mid v \in [\![F]\!]_G \cap S\}$ *and* $node_i^-(vis_{M^{i,c,g}_{Filt}(F,G,S)}(g(x))) = G^{tdn}(F,S)$,
- $[\![g(x)]\!]_{M^{i,c,r,g}_{Ex}(P,G,S)} = \{[x/v] \mid v \in S \wedge [\![P]\!]_G(\{v\}) \neq \emptyset\}$ *and* $node_i^-(vis_{M^{i,c,r,g}_{Ex}(P,G,S)}(g(x))) = G^{tdn}(P,S)$.

Together with Theorem 1 this theorem has for consequence that the programs we generate evaluate with complexity $\mathcal{O}(|P||G^{tdn}(P,S)|)$ or $\mathcal{O}(|F||G^{tdn}(F,S)|)$ as announced.

6 Jumping in Graphs

Preprocessing large graphs is mandatory if one wants to share some of the efforts when evaluating multiple queries on the same graphs. Most typically, one can pre-computate indexes that give efficient access to some particular relations of the graph. For could allow us to jump 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 would visit only the top-down needed nodes but not not only with respect to the graph but also with respect to the given indexes.

For instance, suppose that we have an index for the NRPQ $acc_a = \text{edge}^*/a?$. 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.

Our general jumping algorithm can be used to reformulate in simple terms a variant of a very efficient automata-based algorithm proposed by Maneth and Nguyen [6] 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 edge/a?, and for jumping to top-most a-labeled descendants, i.e., top_a = (edge/$\neg a$?)*/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 for NRPQs allows us to prove that our algorithm only visits the interesting part of the graph. It may also allow the design of algorithms that transform NRPQs into equivalent ones that have a smaller needed subgraph. 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. 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 so as to follow more closely the implementation of Maneth and Nguyen. We believe that this is possible with datalog, but we leave the compilation schema for later work.

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

References

[1] Marcelo Arenas and Jorge Pérez. Querying semantic web data with sparql. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 305–316, 2011.

[2] Rance Cleaveland and Bernhard Steffen. 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, page 48–58, Berlin, Heidelberg, 1991. Springer-Verlag.

[3] Michael J. Fischer and Richard E. Ladner. Propositional dynamic logic of regular programs. *J. Comput. Syst. Sci.*, 18(2):194–211, 1979.

[4] Georg Gottlob, Christoph Koch, and Reinhard Pichler. The complexity of XPath query evaluation. In *22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 179–190, 2003.

[5] Leonid Libkin, Wim Martens, and Domagoj Vrgovc. Querying graph databases with XPath. In *Proceedings of the 16th International Conference on Database Theory*, ICDT '13, page 129–140, New York, NY, USA, 2013. Association for Computing Machinery.

[6] Sebastian Maneth and Kim Nguyen. Xpath whole query optimization. *PVLDB*, 3(1):882–893, 2010.

[7] Wim Martens and Tina Trautner. Evaluation and Enumeration Problems for Regular Path Queries. In Benny Kimelfeld and Yael Amsterdamer, editors, *21st International Conference on Database Theory (ICDT 2018)*, volume 98 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:21, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. nSPARQL: A navigational language for RDF. *J. Web Semant.*, 8(4):255–270, 2010.

[9] K. Tekle and Yanhong Liu. Precise complexity analysis for efficient datalog queries. In *PPDP'10 - Proceedings of the 2010 Symposium on Principles and Practice of Declarative Programming*, pages 35–44, 01 2010.

[10] K. Tekle and Yanhong Liu. Extended magic for negation: Efficient demand-driven evaluation of stratified datalog with precise complexity guarantees. *Electronic Proceedings in Theoretical Computer Science*, 306:241–254, 09 2019.

[11] J. D. Ullman. Bottom-up beats top-down for datalog. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '89, page 140–149, New York, NY, USA, 1989. Association for Computing Machinery.

A Correctness Proof

In the appendix we provide a sketch of the proof for our main Theorem 2.

For the proof, we define the function $reached_{M,r}(\vec{\ell}\,)$ which returns a set of all nodes v, such that $r(v)$ is queried in the proccess of the top-down evaluation of datalog program M with the goal $?{-}\vec{\ell}$:

$$\begin{aligned}
reached_{M,r}(\epsilon) &= \emptyset \\
reached_{M,r}(r(v),\vec{\ell}_1) &= \bigcup\Big\{\{v\}\cup reached_{M,r}(\sigma(\vec{\ell}_2,\vec{\ell}_1)) \mid \quad \sigma = unif(r(v),\ell'), \\
&\qquad\qquad\qquad\qquad\qquad\qquad \ell' :- \vec{\ell}_2.\text{ in } ren(M)\Big\} \\
reached_{M,r}(\ell,\vec{\ell}_1) &= \bigcup\Big\{reached_{M,r}(\sigma(\vec{\ell}_2,\vec{\ell}_1)) \mid \quad \sigma = unif(\ell,\ell'), \\
&\qquad\qquad\qquad\qquad\qquad\qquad \ell' :- \vec{\ell}_2.\text{ in } ren(M), \ell \neq r(v)\Big\}
\end{aligned}$$

Now, we provide two lemmas for dividing the proof into two parts. First — about filter queries and path queries that are subqueries of some query $[P]$. Second lemma about all other path queries.

Lemma 2. *For any filter query F, path query P, label a, labeled graph G, subset S of nodes of G, distinct extensional predicates $i, c, r, g \in \mathcal{P}$ and $x \in \mathcal{V}$.*

1. $\llbracket g(x)\rrbracket_{M^{i,c,g}_{Filt}(F,G,S)} = \{[x/v] \mid v \in \llbracket F\rrbracket_G \cap S\}$
   $node^-_i(vis_{M^{i,c,g}_{Filt}(F,G,S)}(g(x))) = G^{tdn}(F,S)$

2. $\llbracket g(x)\rrbracket_{M^{i,c,r,g}_{Ex}(P,G,S)} = \{[x/v] \mid v \in S \wedge \llbracket P\rrbracket_G(\{v\}) \neq \emptyset\}$
   $node^-_i(vis_{M^{i,c,r,g}_{Ex}(P,G,S)}(g(x))) = G^{tdn}(P,S)$
   $reached_{M^{i,c,r,g}_{Ex}(P,G,S),r}(g(x)) = \llbracket P\rrbracket_G(S)$

Proof (sketch). Let G be a graph with a node set V. The proof is simultaneous induction on $F \in \mathcal{F}_\Sigma$ and $P \in \mathcal{P}_\Sigma$. In this sketch of proof we consider some possible forms of filter queries $F \in \mathcal{F}_\Sigma$ and path queries $P \in \mathcal{P}_\Sigma$:

Proof of 1 for the case $F = F' \wedge F''$ Following the compiler to monadic datalog:

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

Let M be the datolog program $M^{i,c,g}_{Filt}(F' \wedge F'', G, S)$, $M' = M^{i,c',g}_{Filt}(F', G, S)$, $M'' = M^{i,c'',g}_{Filt}(F'', G, S)$. The top-down evaluator will visit $G^{tdn}(F', S)$ in order to get all facts corresponding to the filter query F'. After that, the top-down evaluator will start from all nodes that satisfy the filter query F' to find those that also satisfy the filter query F''.

First, we show that the result of top-down evaluation $\llbracket g(x)\rrbracket_M$ will be substitutions which maps x to the set of nodes $\llbracket F' \wedge F''\rrbracket_G \cap S$. By induction hypothesis, $\llbracket g(x)\rrbracket_{M'} = \{[x/v] \mid v \in \llbracket F'\rrbracket_G \cap S\}$ and $\llbracket g(x)\rrbracket_{M''} = \{[x/v] \mid v \in \llbracket F''\rrbracket_G \cap S\}$. Therefore, $\llbracket g(x)\rrbracket_M = \{[x/v] \mid v \in \cap\llbracket F''\rrbracket_G \cap (\llbracket F'\rrbracket_G \cap S)\} = \{[x/v] \mid v \in \llbracket F' \wedge F''\rrbracket_G \cap S\}$.

Next, we show that the top-down evaluator will visit the FEDB equal to $G^{tdn}(F' \wedge F'', S)$. By induction hypothesis, $node^-_i(vis_{M'}(g(x))) = G^{tdn}(F', S)$ and $node^-_i(vis_{M''}(g(x))) = G^{tdn}(F'', S)$. Therefore,

$$node^-_i(vis_M(g(x))) = G^{tdn}(F', S) \cup G^{tdn}(F'', \llbracket F'\rrbracket_G(S)) = G^{tdn}(F' \wedge F'', S).$$

Proof of 2 for the case $P = P'/P''$ The compiler to monadic datalog yields:

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

Let M be the datolog program $M^{i,c,r,g}_{Ex}(P, G, S)$. We can divide the top-down evaluation of the program M into two steps. First — the top-down evaluation $[\![g(x)]\!]_{M^{i,c,f,g}_{Ex}(P',G,S)}$ which provides the set of facts R for predicate f where $R = reached_{M^{i,c,f,g}_{Ex}(P',G,S),f}(g(x))$ is a set of all nodes v, such that $f(v)$ is queried during the first step of evaluation. The second step is equivalent to the evaluation $[\![g(x)]\!]_{M^{i,f,r,g}_{Ex}(P'',G,R)}$ where the set of facts R is used as the set of starting nodes. By induction hypothesis, $R = [\![P']\!]_G(S)$. Therefore, the result of top-down evaluation $[\![g(x)]\!]_M$ is equal to the join of the result of evaluation $[\![g(x)]\!]_{M^{i,c,f,g}_{Ex}(P',G,S)}$ and evaluation $[\![g(x)]\!]_{M^{i,f,r,g}_{Ex}(P'',G,[\![P']\!]_G(S))}$. By induction hypothesis, $[\![g(x)]\!]_{M^{i,c,f,g}_{Ex}(P',G,S)} = \{[x/v] \mid v \in [\![[P']\!]_G(S)\}$ and $[\![g(x)]\!]_{M^{i,f,r,g}_{Ex}(P'',G,[\![P']\!]_G(S))} = \{[x/v] \mid v \in [\![[P'']\!]_G([\![P']\!]_G(S))\}$. Therefore,

$$\begin{aligned}
[\![g(x)]\!]_M &= \{\sigma \bowtie \Pi_\emptyset(\sigma') \mid \sigma \in \{[x/v] \mid v \in S, v \in [\![[P']\!]_G\}, \\
&\qquad \sigma' \in \{[x/v, y/v'] \mid (v, v') \in [\![P']\!]_G, v' \in [\![[P'']\!]_G\}\} \\
&= \{[x/v] \mid v \in S, \exists v', v''.(v, v') \in [\![P']\!]_G, (v', v'') \in [\![P'']\!]_G\} \\
&= \{[x/v] \mid v \in [\![[P'/P'']\!]_G(S)\}
\end{aligned}$$

Next, we show that the top-down evaluator of the datalog program M will visit the FEDB equal to $G^{tdn}([P'/P''], S)$. By induction hypothesis, $node^-_i(vis_{M^{i,c,f,g}_{Ex}(P',G,S)}(g(x))) = G^{tdn}([P'], S)$ and $node^-_i(vis_{M^{i,f,r,g}_{Ex}(P'',G,[\![P']\!]_G(S))}(g(x))) = G^{tdn}([P''], [\![P']\!]_G(S))$. The FEDB visited during the top-down evaluation of the datalog program M is equal to $G^{tdn}([P'], S) \oplus G^{tdn}([P''], [\![P']\!]_G(S)) = G^{tdn}([P'/P''], S)$.

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 program 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 $[\![g(x)]\!]_{M^{i,f,r,g}_{Ex}(P'',G,[\![P']\!]_G(S))}$. By induction hypothesis,

$$reached_{M^{i,f,r,g}_{Ex}(P'',G,[\![P']\!]_G(S)),r}(g(x)) = [\![P'']\!]_G([\![P']\!]_G(S)).$$

Thus,

$$\begin{aligned}
reached_{M,r}(g(x)) &= reached_{M^{i,f,r,g}_{Ex}(P'',G,[\![P']\!]_G(S)),r}(g(x)) \\
&= [\![P'']\!]_G([\![P']\!]_G(S)) \\
&= [\![P'/P'']\!]_G(S)
\end{aligned}$$

□

Lemma 3. *For any path query P, labeled graph G, subset S of nodes of G, distinct predicates $i, f \in \mathcal{P}$ and $x \in \mathcal{V}$*

$$[\![f(x)]\!]_{M^{i,f}_{Acc}(P,G,S)} = \{[x/v] \mid v \in [\![P]\!]_G(S)\}$$
$$node^-_i(vis_{M^{i,f}_{Acc}(P,G,S)}(f(x))) = G^{tdn}(P, S)$$

Proof (sketch). Let G be a graph with a node set V. The proof is simultaneous induction on $P \in \mathcal{P}_\Sigma$. In this sketch of proof we consider one possible form of path queries $P \in \mathcal{P}_\Sigma$:

Case $P = P'/P''$ The compiler to monadic datalog yields:

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

Let M be the datolog program $M^{i,f}_{Acc}(P'/P'', G, S)$. We can divide the top-down evaluation of the program M into two steps. First — the top-down evaluation $[\![f'(x)]\!]_{M^{i,f'}_{Acc}(P',G,S)}$ which 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 $[\![f(x)]\!]_{M^{f',f}_{Acc}(P'',G,R)}$ where the set of nodes R is used as the set of starting nodes. By induction hypothesis, $R = [\![P']\!]_G(S)$. Therefore, $[\![f(x)]\!]_M = [\![f(x)]\!]_{M^{f',f}_{Acc}(P'',G,[\![P']\!]_G(S))} = \{[x/v''] \mid v'' \in [\![P'/P'']\!]_G(S)\}$.

Next, we show that the top-down evaluator of the datalog program M will visit the FEDB equal to $G^{tdn}(P'/P'', S)$. By induction hypothesis,

$$node_i^-(vis_{M^{i,f'}_{Acc}(P',G,S)}(f'(x))) = G^{tdn}(P', S),$$

$$node_i^-(vis_{M^{f',f}_{Acc}(P'',G,[\![P']\!]_G(S))}(f(x))) = G^{tdn}(P'', [\![P']\!]_G(S)).$$

Therefore,

$$node_i^-(vis_M(f(x))) = G^{tdn}(P', S) \oplus G^{tdn}(P'', [\![P']\!]_G(S)) = G^{tdn}(P'/P'', S)$$

□

Finally, combining the two lemmas (Lemma 2 and Lemma 3) gives us a proof for our main Theorem 2.