Computing and Informatics, Vol. 38, 2019, 19–56, doi: 10.31577/cai_2019_1_19

INCORPORATING STRATIFIED NEGATION INTO QUERY-SUBQUERY NETS FOR EVALUATING QUERIES TO STRATIFIED DEDUCTIVE DATABASES

Son Thanh Cao

School of Engineering and Technology, Vinh University
182 Le Duan street, Vinh, Nghe An, Vietnam
&
Nguyen Tat Thanh University, Ho Chi Minh City, Vietnam
e-mail: sonct@vinhuni.edu.vn

Linh Anh Nguyen*

Division of Knowledge and System Engineering for ICT
Faculty of Information Technology
Ton Duc Thang University, Ho Chi Minh City, Vietnam
&
Institute of Informatics, University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
e-mail: nguyenanhlinh@tdtu.edu.vn, nguyen@mimuw.edu.pl

Abstract. Most of the previously known evaluation methods for deductive databases are either breadth-first or depth-first (and recursive). There are cases when these strategies are not the best ones. It is desirable to have an evaluation framework for stratified Datalog$^\neg$ that is goal-driven, set-at-a-time (as opposed to tuple-at-a-time) and adjustable w.r.t. flow-of-control strategies. These properties are important for efficient query evaluation on large and complex deductive databases. In this paper, by incorporating stratified negation into so-called *query-subquery nets*, we develop an evaluation framework, called QSQN-STR, with such properties for evaluating queries to stratified Datalog$^\neg$ databases. A variety of flow-of-control strategies can be used for QSQN-STR. The generic evaluation method QSQN-STR for stratified Datalog$^\neg$ is sound, complete and has a PTIME data complexity.

Keywords: Deductive databases, datalog with negation, query processing

* corresponding author

Mathematics Subject Classification 2010: 68N17

1 INTRODUCTION

Datalog is a well-known rule-based query language for deductive databases. In [24], Huang et al. wrote *"we are witnessing an exciting revival of interest in recursive Datalog queries in a variety of emerging application domains such as data integration, information extraction, networking, program analysis, security, and cloud computing"* (see also, e.g., [23, 7]). Datalog expresses the Horn fragment with the safety condition[1] and without function symbols of first-order logic and uses the traditional monotonic semantics. The extension Datalog$^\neg$ of Datalog allows expressing non-monotonic queries by using negation in the bodies of program clauses. It uses a non-monotonic semantics like the standard semantics for stratified Datalog$^\neg$ programs and the well-founded semantics for the general case. A Datalog$^\neg$ program is stratifiable if it can be divided into strata such that, if a negative literal of a predicate p occurs in the body of a program clause in a stratum, then the clauses defining p must belong to an earlier stratum. A deductive database consists of a Datalog/Datalog$^\neg$ program (for defining intensional predicates) and an instance of extensional predicates.

This work studies query processing for stratified Datalog$^\neg$ databases. The topic is worthy of consideration due to practical applications of deductive databases.

1.1 Related Work

Researchers have developed a number of evaluation methods for Datalog databases, such as QSQ [43, 1], QSQR [43, 31], QoSaQ [44] and Magic-Sets [5, 6] (by Magic-Sets we mean the evaluation method that combines the magic-set transformation with the improved semi-naive evaluation method).

QSQ (Query-Subquery) [43, 1] is a framework for evaluating queries to Datalog databases. Its approach is top-down (i.e., query processing is closely related to the main goal) and set-at-a-time (i.e., operations are set-oriented but not tuple-oriented). It implements a *tabulation* (tabling/memoing) technique by using so-called *input*, *answer* and *supplement* relations to guarantee termination. *Adornments* for intensional predicates (and their corresponding *input* and *answer* relations) are used to enable exploiting relational operations like join and projection. In general, QSQ uses adornments to simulate SLD-resolution in pushing constant symbols from goals to subgoals. An enhanced version of QSQ, called *annotated QSQ*, also uses *annotations* to simulate SLD-resolution in pushing repeats of variables from goals to subgoals. A variety of flow-of-control strategies (which are similar to search strategies and called *control strategies* for short) can be used for QSQ.

[1] For a definition of the safety condition, see the paragraph after Definition 1.

QSQR (QSQ Recursive), introduced by Vieille in [43], is a query evaluation method for Datalog databases that follows the QSQ approach and uses a recursive strategy. Roughly speaking, the strategy is *depth-first*, but due to tabulation, as observed by Vieille [44], the QSQR approach is like iterative deepening search. The versions of QSQR presented in [43, 1] are incomplete [31, 44, 29]. This is corrected in [29] by using an outer loop that clears global *input* relations for each iteration.

In [44], Vieille introduced another method, called QoSaQ, for evaluating queries to Datalog databases. It is an adaptation of SLD-AL resolution. The method can be implemented as a set-oriented procedure, but as stated by Vieille himself, the practical interest of the method lies in its one-inference-at-a-time basis, as opposed to the set-at-a-time approach. The intention is to permit an advanced analysis of the duplicate elimination issue.

The magic-sets technique [5, 6] simulates the top-down QSQ approach by rewriting the Datalog program together with the given query to another equivalent one that when evaluated using a bottom-up technique (e.g., the improved semi-naive evaluation) produces only facts produced by the QSQ evaluation. Adornments are used as in the QSQ approach. To simulate annotations, the magic-sets transformation is augmented with subgoal rectification (see, e.g., [1]).

In [11, 9], we provided a framework called QSQN (Query-Subquery Nets) for evaluating queries to Horn knowledge bases. It uses a parameter for the limit on the nesting depths of terms occurring in the computation. When this limit is set to 0, the framework can be used for evaluating queries to Datalog databases. QSQN is an adaptation and a generalization of the QSQ approach for Horn knowledge bases. One of the key differences is that it does not use adornments and annotations, but uses substitutions instead. This is natural for the case with function symbols and without the safety condition. Like QSQ, every control strategy can be used for QSQN. The notion of query-subquery net makes a linkage to flow networks and is intuitive for developing efficient evaluation algorithms.

A top-down approach with tabulation for dealing with stratified Datalog$^\neg$ was proposed in [26, 41, 37]. The evaluation procedures given in [26, 41, 37] are similar to each other, with some differences as discussed in [37]. They are called "QSQR/SLS-procedure" in [26, 37] and we will refer to them as the QSQR/SLS method. This method relies on using a derivation forest (of global SLS-resolution) with tabulation and is implemented using the recursive approach like QSQR.

In [37], apart from QSQR/SLS, Ross also proposed a bottom-up evaluation method for stratified Datalog$^\neg$ by presenting a magic-sets transformation, which simulates the top-down QSQR/SLS method, but the program obtained from the transformation can be evaluated using a bottom-up technique. Programs obtained from the transformation are not stratified Datalog$^\neg$ programs, as they use special "literals" for checking whether the computation of the corresponding negative goals has been completed.

In [4], Balbin et al. proposed another bottom-up evaluation method for stratified Datalog$^\neg$. Their method applies a magic-sets transformation and a bottom-up computation with recursive calls for evaluating negative goals.

The well-founded semantics is a commonly accepted choice for (general) Datalog$^\neg$, as it coincides with the standard semantics for stratified Datalog$^\neg$, and using it Datalog$^\neg$ has a PTIME data complexity. This semantics was first introduced by Van Gelder et al. for normal logic programs [19] and can be characterized by the alternating fixpoint [18]. Several calculi for normal logic programs that are sound and complete w.r.t. the well-founded semantics have been developed. One of them is SLG-resolution. In [15], Chen et al. presented efficient techniques for implementing SLG-resolution. Their method maintains positive and negative dependencies among subgoals in a top-down evaluation, detects positive and negative loops, delays subgoals when possible loops occur, checks completion of subgoals and resumes their activeness when possible. It is tuple-oriented and its implementation XSB [40] can be used as an engine for in-memory Datalog$^\neg$ databases.

Kemp et al. [25] and Morishita [30] proposed bottom-up evaluation methods for Datalog$^\neg$ under the well-founded semantics. Their methods are based on Van Gelder's alternating fixpoint characterization and use a magic-sets transformation with adornments but without annotations.

In [14], together with a colleague we extended QSQN to obtain a method called QSQN-WF for evaluating queries to Datalog$^\neg$ databases under the well-founded semantics. It follows Przymusinski's SLS-resolution [34], with Van Gelder's alternating fixpoint semantics [18] on the background, but uses a query-subquery net to implement tabulation and the set-at-a-time technique.

1.2 Motivations

We first discuss some important aspects of query evaluation (in Sections 1.2.1–1.2.3), and then state motivations of our work (in Section 1.2.4).

1.2.1 Adjustability w.r.t. Control Strategies

The techniques used for query evaluation are usually separated into two classes depending on whether they focus on top-down or bottom-up evaluation [1]. Here, "top-down" is understood as "*goal-driven*" (i.e., query processing is relevant to the subgoals and therefore closely related to the main goal). As the bottom-up evaluation methods based on the magic-sets technique simulate the top-down evaluation, they are also goal-driven. Since the terms "top-down" and "bottom-up" are antonyms, it is better to classify top-down evaluation as goal-driven and characterize bottom-up evaluation methods by an additional property. Being goal-driven can be treated as a requirement for efficient evaluation methods.

The aforementioned bottom-up evaluation methods for Datalog [5, 6], stratified Datalog$^\neg$ [37, 4] and Datalog$^\neg$ [25, 30] use a magic-sets transformation and a bottom-up computation like the improved semi-naive evaluation. So, they can be characterized as goal-driven and *breadth-first* (i.e., based on using a breadth-first control strategy).[2] On the other hand, the top-down evaluation methods QSQR [43, 31]

[2] The naive evaluation can be described as follows: repeat applying all of the rules

and QoSaQ [44] (for Datalog), QSQR/SLS [26, 41, 37] (for stratified Datalog$^\neg$) and SLG-resolution [15, 40] (for normal logic programs and Datalog$^\neg$ databases) can be characterized as goal-driven and *depth-first*.[3] The frameworks QSQ [43, 1] (for Datalog), QSQN [11, 9] (for Horn knowledge bases and Datalog databases) and QSQN-WF [14] (for Datalog$^\neg$) follow the goal-driven approach but allow *every* control strategy.

The breadth-first and depth-first approaches are just two among possible approaches. There are cases when they are not the best ones [11]. When developing a framework for query evaluation one should make it general to a certain extent so that a variety of control strategies can be used. In particular, it is desirable to be able to control the computation flow dynamically.

1.2.2 Set-at-a-Time vs. Tuple-at-a-Time

The evaluation methods QoSaQ [44] (for Datalog) and SLG-resolution [15, 40] (for normal logic programs and Datalog$^\neg$ databases) are tuple-at-a-time (tuple-oriented). They use complex data structures for handling individual subgoals (tuples), and when the extensional relations and the search space are too large, in-memory computation may be impossible. XSB [40] is an efficient engine for in-memory deductive databases due to the suspension-resumption mechanism, advantages of WAM (Warren Abstract Machine) and other optimizations. Such techniques are highly tuple-oriented. When the extensional relations are too large and the program defining intensional predicates is sophisticated, accesses to the secondary storage may be unavoidable, and the set-at-a-time approach is preferable.

Regarding the evaluation methods QSQR [43, 31] (for Datalog) and QSQR/SLS [26, 41, 37] (for stratified Datalog$^\neg$), they can be implemented using either the tuple-at-a-time approach or the set-at-a-time approach. But, using the latter one the recursive strategy is unavoidable. As observed in [29, Remark 3.2], using the recursive approach, *input* relations should be cleared occasionally (e.g., at the beginning of each iteration of the main loop) in order to allow recomputations using updated *answer* relations. This causes redundant computations.

1.2.3 Why Are Evaluation Methods for Stratified Datalog$^\neg$ Needed?

The question is rather "are the known evaluation methods for (general) Datalog$^\neg$ efficient for evaluating queries to stratified Datalog$^\neg$ databases?". The general answer is "they are not as efficient as expected for that kind of tasks". The reason is that they were developed to cope with unstratified negation and are thus superfluous. For example, the methods proposed in [25, 30, 14] are based on Van Gelder's

sequentially, one after the other, until no new facts were derived during the last iteration. Its approach is like breadth-first search. The improved semi-naive evaluation (see, e.g., [1]) shares this property.

[3] The mentioned methods use a recursive control strategy, which is like the depth-first search strategy implemented using recursive calls.

alternating fixpoint characterization and use an (additional) outer loop to guarantee that an alternating fixpoint can be reached. When applied to stratified Datalog$^\neg$, that causes certain redundant (re)computations.

Apart from the well-founded semantics, the stable model semantics [20] is also a well-known semantics for normal logic programs (see, e.g., the survey [3]). These two semantics coincide for certain classes of logic programs [35, 16, 21], including stratified logic programs and stratified Datalog$^\neg$ programs. The stable model semantics is used for answer set programming (ASP), and systems like DLV [27], NP Datalog [22] and *clasp* [17], which deal among others with ASP, can be used for answering queries to stratified Datalog$^\neg$ databases. However, as the main aim of ASP engines is to find an answer set (i.e., a stable model) for a given logic program, they are not goal-driven and, in general, not as efficient as expected for answering queries to stratified Datalog$^\neg$ databases.

1.2.4 The Need for a New Evaluation Framework for Stratified Datalog$^\neg$

As discussed in Sections 1.1–1.2.3, the previously known methods that can be used for evaluating queries to stratified Datalog$^\neg$ databases are:

- breadth-first [4, 37, 25, 30] or depth-first [26, 41, 37, 15, 40]; or/and
- tuple-at-a-time [15, 40]; or/and
- designed for (general) Datalog$^\neg$ [25, 30, 14] or normal logic programs [15, 40], and not as efficient as expected for stratified Datalog$^\neg$.

That is, none of the previously known evaluation methods is goal-driven, set-at-a-time, adjustable w.r.t. control strategies, and designed specially for stratified Datalog$^\neg$ but not (general) Datalog$^\neg$. As these properties are important for efficient query evaluation on large and complex stratified Datalog$^\neg$ databases, it is desirable to develop an evaluation framework for stratified Datalog$^\neg$ with such properties.

1.3 Our Contributions

In this paper, we provide a novel framework, called QSQN-STR, for evaluating queries to stratified Datalog$^\neg$ databases. It extends the QSQN framework [11, 9] with the ability to handle stratified negation (but is formulated for stratified Datalog$^\neg$ databases instead of stratified knowledge bases in first-order logic). QSQN-STR is goal-driven, set-at-a-time and allows a variety of control strategies. In particular, every control strategy "*admissible w.r.t. strata's stability*" can be used for QSQN-STR. Roughly speaking, the admissibility w.r.t. strata's stability only requires that the computation can check whether a (ground) negative goal $\sim B$ of an intensional predicate p succeeds by searching the *answer* relation of p only after the (goal-driven) processing for the lower strata up to the stratum containing clauses defining p has been completed. QSQN-STR uses a net of nodes that correspond to *input*, *answer* and *supplement* relations like the ones used for QSQ [43, 1] but without adornments. The net is constructed from the given stratified Datalog$^\neg$ program.

It contains simple data structures that are needed for performing query evaluation. At the abstract level, the skeleton of QSQN-STR is as follows:

> while there are edges (u, v) such that u contains data to be processed for the edge (u, v), do:
>
> - select such an edge so that the selection is admissible w.r.t. strata's stability;
> - process the data at u to produce and transfer data through the edge (u, v).

As QSQN-STR allows every control strategy that is admissible w.r.t. strata's stability, it is really a framework. We also refer to it as a generic evaluation method for stratified Datalog$^\neg$. This method is sound, complete and has a PTIME data complexity.

What control strategies should be used for QSQN-STR is left for the implementation and experimentation phases. Besides, operations specified for QSQN-STR can be optimized at the implementation phase. We have implemented a prototype of QSQN-STR in Java, using a control strategy named IDFS2, which is specified in [9]. The prototype has not yet been optimized. So, in general, it cannot compete with highly optimized engines like XSB [40]. Nevertheless, we have performed experiments and made a comparison between our prototype of QSQN-STR, DES-DBMS[4] (version 5.0.1) and SWI-Prolog[5] (version 6.4) w.r.t. the execution time by using a number of tests. The experimental results show that our prototype of QSQN-STR outperforms DES-DBMS by a few orders of magnitude for all of the tests. It is competitive with SWI-Prolog for the tests for which SWI-Prolog can terminate properly.

This paper is a revised and extended/modified version of the conference paper [8] and a chapter of the first author's PhD dissertation [9]. The QSQN-STR framework presented in this paper is formulated for stratified Datalog$^\neg$ databases but not stratified knowledge bases in first-order logic. It has been improved by allowing a larger class of control strategies and adopting an essential optimization[6]. Consequently, the proof of soundness and completeness has been updated. Furthermore, the presentation has been significantly improved.

1.4 The Structure of This Paper

The rest of this paper is structured as follows. Section 2 recalls the most important concepts and definitions. In Section 3, we give a new presentation of the QSQN framework, which is thorough and more understandable than the one in [11, 9]. In Section 4, we incorporate stratified negation into query-subquery nets and extend QSQN to QSQN-STR. (To get the gist of QSQN-STR, the reader may watch the demonstration [12] in the PowerPoint-like mode first.) Conclusions are given

[4] The Datalog Education System (DES) with a DBMS via ODBC, available at `http://des.sourceforge.net` (see also, e.g., [39]).

[5] Available at `http://www.swi-prolog.org/`

[6] See the step 2a of $\texttt{fire}'(u, v)$ in Definition 23.

in Section 5. Due to the lack of space, our proofs and experimental results are presented only in the online appendix [13].

2 PRELIMINARIES

We assume that the reader is familiar with basic notions of first-order logic. In this section, we recall only the most important definitions and notions that are needed for our work, which are based on [1, 2, 14, 23, 28, 33]. We refer the reader to [1, 28] for further reading.

A *signature* for Datalog$^\neg$ consists of constants, variables and predicates. Each predicate is classified either as *intensional* or as *extensional*. Due to the absence of function symbols, a *term* is defined to be either a constant or a variable. An *atom* is an expression of the form $p(t_1, \ldots, t_n)$, where $n \geq 0$, p is an n-ary predicate and each t_i is a term. A *literal* is either an atom (called a *positive literal*) or the negation of an atom (called a *negative literal*). *Formulas* are defined in the usual way. An *expression* is a term, a tuple of terms, a formula without quantifiers or a list of formulas without quantifiers. A *simple expression* is either a term or an atom. An expression is *ground* if it does not contain variables.

2.1 Substitution and Unification

A *substitution* is a finite set $\theta = \{x_1/t_1, \ldots, x_k/t_k\}$, where $x_1, \ldots, x_k$ are pairwise distinct variables, $t_1, \ldots, t_k$ are terms, and $t_i \neq x_i$ for all $1 \leq i \leq k$. The set $dom(\theta) = \{x_1, \ldots, x_k\}$ is called the *domain* of θ, and $range(\theta) = \{t_1, \ldots, t_k\}$ the *range* of θ. The *restriction* of a substitution θ to a set X of variables is the substitution $\theta_{|X} = \{(x/t) \in \theta \mid x \in X\}$. By ε we denote the *empty substitution*.

Given an expression E and a substitution $\theta = \{x_1/t_1, \ldots, x_k/t_k\}$, the *instance* of E by θ, denoted by $E\theta$, is defined to be the expression obtained from E by simultaneously replacing every occurrence of x_i in E by t_i, for all $1 \leq i \leq k$.

Given substitutions $\theta = \{x_1/t_1, \ldots, x_k/t_k\}$ and $\delta = \{y_1/s_1, \ldots, y_h/s_h\}$, the *composition* $\theta\delta$ of θ and δ is defined to be the substitution obtained from the sequence $\{x_1/(t_1\delta), \ldots, x_k/(t_k\delta), y_1/s_1, \ldots, y_h/s_h\}$ by deleting any binding $x_i/(t_i\delta)$ with $x_i = (t_i\delta)$ and deleting any binding y_j/s_j with $y_j \in \{x_1, \ldots, x_k\}$.

A substitution θ is *idempotent* if $\theta\theta = \theta$. It is known that $\theta = \{x_1/t_1, \ldots, x_k/t_k\}$ is idempotent if and only if none of $x_1, \ldots, x_k$ occurs in any $t_1, \ldots, t_k$. If θ and δ are substitutions such that $\theta\delta = \delta\theta = \varepsilon$, then we call them *renaming substitutions*. A substitution θ is *more general* than a substitution δ if there exists a substitution γ such that $\delta = \theta\gamma$. According to this definition, θ is more general than itself.

Let Γ be a set of simple expressions. A substitution θ is called a *unifier* for Γ if $\Gamma\theta$ is a singleton. If $\Gamma\theta = \{\varphi\}$, then we say that θ unifies Γ (into φ). A unifier θ for Γ is called a *most general unifier* (mgu) for Γ if θ is more general than every unifier of Γ. There is an effective algorithm, called the *unification algorithm*, for checking

whether a set Γ of simple expressions is unifiable (i.e., has a unifier) and computing an idempotent mgu for Γ if Γ is unifiable (see, e.g., [28]).

2.2 Stratified Datalog$^\neg$

We recall here the definition of databases in Datalog and (stratified) Datalog$^\neg$.

Definition 1. A *safe Datalog$^\neg$ program clause* (w.r.t. the leftmost selection function) has the form $A \leftarrow B_1, \ldots, B_k$, with $k \geq 0$, and satisfies the following conditions:

1. A is an atom and each B_i is a literal, where negation is denoted by $\sim$,
2. every variable occurring in A also occurs in $(B_1, \ldots, B_k)$,
3. every variable occurring in a negative literal B_j also occurs in some positive literal B_i with $1 \leq i < j$.

The atom A is called the *head* and $(B_1, \ldots, B_k)$ the *body* of the program clause. When $k = 0$, the body is empty and the clause can be written without $\leftarrow$. If p is the predicate of A, then the program clause is called a *program clause defining* p. Such a program clause is treated as an expression (so we can talk about its instances).

A *safe Datalog$^\neg$ program* (w.r.t. the leftmost selection function) is a finite set of safe Datalog$^\neg$ program clauses. A safe Datalog$^\neg$ program without negative literals in the clauses' bodies is called a *safe Datalog program*. From now on, by a *Datalog$^\neg$* (resp. *Datalog*) *program* we mean a safe Datalog$^\neg$ (resp. Datalog) program. The second and third conditions in Definition 1 are called the *safety condition of Datalog$^\neg$*. The second condition itself is also called the *safety condition of Datalog*.

Given a Datalog$^\neg$ program P, a *stratification* of P is a partition $P = P_1 \cup \ldots \cup P_n$ such that, for each $1 \leq i \leq n$, we have that:[7]

- if an intensional predicate p occurs in a positive literal in the body of a clause from P_i, then the clauses defining p must belong to $P_1 \cup \ldots \cup P_i$,
- if an intensional predicate p occurs in a negative literal in the body of a clause from P_i, then $i > 1$ and the clauses defining p must belong to $P_1 \cup \ldots \cup P_{i-1}$.

Each P_i is called a *stratum* of the stratification. A Datalog$^\neg$ program is called a *stratified Datalog$^\neg$ program* if it has a stratification.

An *instance* of extensional predicates is a mapping I that associates each extensional n-ary predicate p to a finite set $I(p)$ of n-ary tuples of constants. Sometimes, I is treated as the set $\{p(\bar{t}) \mid \bar{t} \in I(p)\}$ and each $p(\bar{t}) \in I$ is treated as the program clause $p(\bar{t}) \leftarrow$. The *size* of I is defined to be the cardinality of the mentioned set.

A *Datalog$^\neg$* (resp. *Datalog*) *database* is a pair (P, I), where P is a Datalog$^\neg$ (resp. Datalog) program consisting of clauses defining intensional predicates and I is an instance of extensional predicates. A *stratified Datalog$^\neg$ database* is a Datalog$^\neg$ database (P, I) with P being a stratified Datalog$^\neg$ program.

[7] All of the sets $P_1, \ldots, P_n$ are assumed to be non-empty.

2.3 The Standard Semantics of Stratified Datalog$^\neg$

In this subsection, let (P, I) be a stratified Datalog$^\neg$ database. The *Herbrand universe* of (P, I), denoted by $U_{P,I}$, is the set of all constants occurring in (P, I). The *Herbrand base* of (P, I), denoted by $B_{P,I}$, is the set of all ground atoms of the form $p(t_1, \ldots, t_n)$, where p is a predicate used in (P, I) and each t_i belongs to $U_{P,I}$. A *Herbrand interpretation* for (P, I) is a subset of $B_{P,I}$.

If $\mathcal{I}$ is a Herbrand interpretation and $p(\bar{t})$ a ground atom, then by $\mathcal{I}(p(\bar{t}))$ we denote that $p(\bar{t}) \in \mathcal{I}$, and by $\mathcal{I}(\sim p(\bar{t}))$ we denote that $p(\bar{t}) \notin \mathcal{I}$.

Let $ground(P \cup I)$ be the set of all ground instances of clauses in $P \cup I$, and $\mathcal{I}$ a Herbrand interpretation for (P, I). The *immediate consequence operator* of (P, I), denoted by $T_{P,I}$, is defined on $\mathcal{I}$ as follows:

$$T_{P,I}(\mathcal{I}) = \{A \mid A \leftarrow B_1, \ldots, B_k \in ground(P \cup I) \text{ and } \mathcal{I}(B_i) \text{ holds for all } 1 \leq i \leq k\}.$$

Let $T_{P,I} \uparrow \omega$ be defined as follows:

$$\begin{aligned} T_{P,I} \uparrow 0 &= I, \\ T_{P,I} \uparrow (n+1) &= T_{P,I}(T_{P,I} \uparrow n) \cup T_{P,I} \uparrow n, \quad \text{for } n \in \mathbb{N}, \\ T_{P,I} \uparrow \omega &= \bigcup_{n=0}^{\omega} T_{P,I} \uparrow n. \end{aligned}$$

Let $P_1 \cup \ldots \cup P_n$ be a stratification of P. We define

$$\begin{aligned} M_{\emptyset,I} &= I, \\ M_{P_1,I} &= T_{P_1,I} \uparrow \omega, \\ M_{P_1 \cup P_2,I} &= T_{P_2,M_{P_1,I}} \uparrow \omega, \\ &\vdots \\ M_{P_1 \cup \ldots \cup P_n,I} &= T_{P_n,M_{P_1 \cup \ldots \cup P_{n-1},I}} \uparrow \omega. \end{aligned}$$

We call $M_{P,I} = M_{P_1 \cup \ldots \cup P_n,I}$ the *standard Herbrand model* of (P, I).

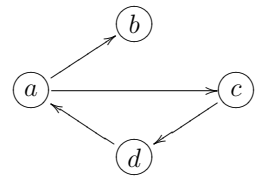
It can be shown that the standard Herbrand model of (P, I) does not depend on the chosen stratification of P (see, e.g., [2, Theorem 11]).

Example 1. Consider the stratified Datalog$^\neg$ database (P, I) given below, where P is a modified version of a Datalog$^\neg$ program given in [36], *path* and *acyclic* are intensional predicates, *edge* is an extensional predicate, x, y and z are variables and a–f are constants. An atom $edge(x, y)$ means that there is an edge from the node x to the node y. An atom $path(x, y)$ means that there exists a path (consisting of edges) that connects the node x to the node y. An atom $acyclic(x, y)$ means that the node x is connected by a path to the node y, but not vice versa.

- P consists of the following clauses:

$$\begin{aligned} path(x,y) &\leftarrow edge(x,y), \\ path(x,y) &\leftarrow path(x,z), edge(z,y), \\ acyclic(x,y) &\leftarrow path(x,y), \sim path(y,x). \end{aligned}$$

- I is specified and illustrated as follows: $I(edge) = \{(a,b),(a,c),(c,d),(d,a)\}$.



The only stratification of P is $P_1 \cup P_2$, where:

$$\begin{array}{ll} P_1: & path(x,y) \leftarrow edge(x,y), \\ & path(x,y) \leftarrow path(x,z), edge(z,y), \\ \hline P_2: & acyclic(x,y) \leftarrow path(x,y), \sim path(y,x). \end{array}$$

The standard Herbrand model $M_{P,I}$ of (P,I) is constructed as follows:

$$\begin{aligned} M_{\emptyset,I} &= I, \\ M_{P_1,I} &= M_{\emptyset,I} \cup \{path(x,y) \mid (x,y) \in \{a,c,d\} \times \{a,b,c,d\}\}, \\ M_{P_1 \cup P_2,I} &= M_{P_1,I} \cup \{acyclic(a,b), acyclic(c,b), acyclic(d,b)\}. \end{aligned}$$

Thus, $M_{P,I} = M_{P_1 \cup P_2,I}$ is the standard Herbrand model of (P,I).

We define a *query* to a stratified Datalog$^\neg$ database (P,I) to be a formula of the form $q(\overline{x})$, where q is an intensional predicate and $\overline{x}$ is a tuple of pairwise distinct variables (of the same arity as q). A *correct answer* for a query $q(\overline{x})$ to a stratified Datalog$^\neg$ database (P,I) is a tuple $\overline{t}$ of constants of the same arity as $\overline{x}$ such that $q(\overline{t}) \in M_{P,I}$. The *data complexity* of an algorithm for computing all (correct) answers for a query $q(\overline{x})$ to a stratified Datalog$^\neg$ database (P,I) is measured in the size of I.

Remark 1. Note that, if φ can be the body of a Datalog$^\neg$ program clause, then it can be treated as a query to a stratified Datalog$^\neg$ database (P,I) by adding to P a new program clause $q(\overline{x}) \leftarrow \varphi$ to obtain P', where q is a new intensional predicate and $\overline{x}$ consists of all the variables occurring in φ, and then using the query $q(\overline{x})$ to the stratified Datalog$^\neg$ database (P',I).

2.4 SLD-Resolution

SLD-resolution [2, 28] is a calculus for the Horn fragment of first-order logic, which is more general than Datalog in that function symbols are allowed and program clauses do not need to satisfy the safety condition. In this subsection, we recall a formulation of SLD-resolution for Datalog, which is useful for our introduction of QSQN in the next section.

If E is an expression or a substitution, then by $Vars(E)$ we denote the set of all variables occurring in E. We say that an expression E is a *variant* of an expression E' if there exist renaming substitutions θ and γ such that $E = E'\theta$ and $E' = E\gamma$. In a computational process, a *fresh variant* of φ, where φ can be a term, a tuple of terms, an atom or a program clause $A \leftarrow B_1, \ldots, B_k$, is $\varphi\theta$, where θ is a renaming substitution such that $dom(\theta) = Vars(\varphi)$ and $range(\theta)$ consists of variables that were not used earlier in the computation.

A *goal (without negation)* has the form $\leftarrow B_1, \ldots, B_k$, where $B_1, \ldots, B_k$ are atoms. If $k = 0$, then the goal is called the *empty goal* and denoted by $\Box$.

A goal G' is *derived* from a goal $G = \leftarrow A_1, \ldots, A_i, \ldots, A_k$ and a Datalog program clause $\varphi = (A \leftarrow B_1, \ldots, B_h)$ using an mgu θ and the *selected atom* A_i if θ is an mgu for A_i and A, and $G' = \leftarrow (A_1, \ldots, A_{i-1}, B_1, \ldots, B_h, A_{i+1}, \ldots, A_k)\theta$. In that case, G' is called a *resolvent* of G and φ. If $i = 1$, then we say that G' is derived from G and φ using *the leftmost selection function*.

In the rest of this subsection, let P be a Datalog program and G a goal.

An *SLD-derivation* from $P \cup \{G\}$ consists of a (finite or infinite) sequence $G_0 = G$, G_1, G_2, ... of goals, a sequence $\varphi_1, \varphi_2, \ldots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that each G_{i+1} is derived from G_i and φ_{i+1} using θ_{i+1}. Each φ_i is called an *input program clause*.

When constructing an SLD-derivation, for generality and clarity, it is assumed that each φ_i does not have any variable that already appears in the derivation up to G_{i-1}. The simplest way to guarantee this is to choose each φ_i as a fresh variant of a program clause from P.

An *SLD-refutation* of $P \cup \{G\}$ is a finite SLD-derivation from $P \cup \{G\}$ with the empty goal as the last goal in the derivation.

A *computed answer* θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition $\theta_1 \ldots \theta_n$ to the variables of G, where $\theta_1, \ldots, \theta_n$ is the sequence of mgu's occurring in an SLD-refutation of $P \cup \{G\}$.

3 QUERY-SUBQUERY NETS REVISITED

The notion of query-subquery net and the related evaluation framework QSQN for evaluating queries to Horn knowledge bases were introduced by us in [11, 9]. They can be used for evaluating queries to Datalog databases by setting the term-depth limit to 0. In this section, we present a thorough and more understandable description of QSQN by using a running example and relating QSQN to SLD-resolution with tabulation.

3.1 An Example Illustrating SLD-Resolution with Tabulation

Consider the stratum P_1 from Example 1 and let I be the extensional instance specified by $I(edge) = \{(a, b), (b, c)\}$. Let $P = P_1 \cup I$, with I treated as a set of atoms of *edge*. Thus, P is the Datalog program consisting of the following clauses:

$$\begin{array}{ll} (1)\ \ path(x, y) \leftarrow edge(x, y), & (3)\ \ edge(a, b), \\ (2)\ \ path(x, y) \leftarrow path(x, z), edge(z, y), & (4)\ \ edge(b, c). \end{array}$$

Consider the goal $\leftarrow path(x, y)$. It is easy to see that there are three computed answers for $P \cup \{\leftarrow path(x, y)\}$: $\{x/a, y/b\}$, $\{x/b, y/c\}$ and $\{x/a, y/c\}$. We first demonstrate how to use SLD-resolution (with the leftmost selection function) together with a technique called *tabulation* to obtain these answers and justify that there are no more computed answers. The process is summarized in Figure 1 and explained in detail below. We use one sequence of natural numbers to name clauses, tuples in relations, and steps in the process (enumerated in the following list). When a tuple is added to a relation at a step k, it is also numbered k.

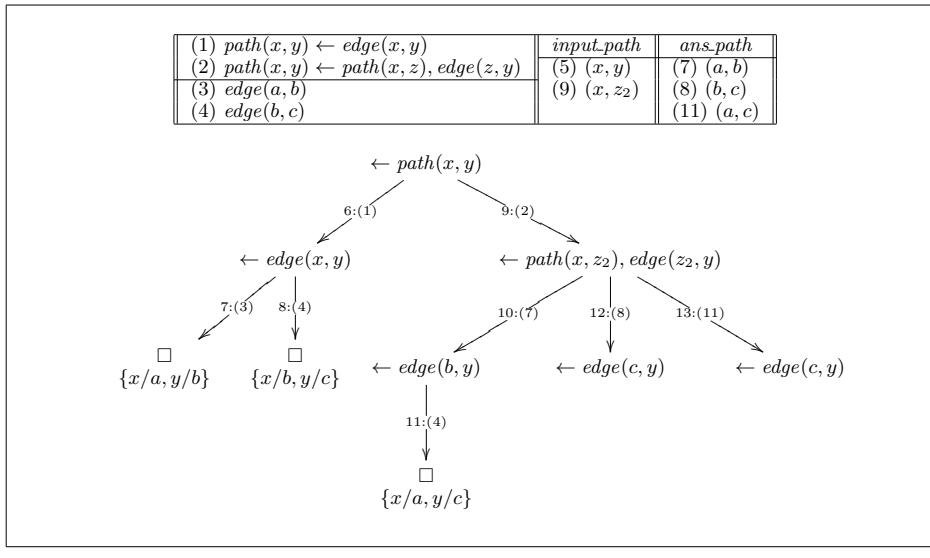


Figure 1. An illustration of SLD-resolution with tabulation. The considered Datalog program consists of the clauses (1)–(4), and the goal is $\leftarrow path(x, y)$. A label of the form $m : (n)$ of an edge from a node v to a node w in the displayed tree means that: at the step m the goal at the node v is resolved by using the clause or the *answer atom* numbered n, resulting in the goal (the resolvent) at the node w. Steps of the process are numbered from 6, as the smaller numbers are reserved for the clauses and the first *input atom*. Details are given in Section 3.1.

(5) As the clauses of P are numbered 1–4, we enumerate this list from 5. To keep the fact that we have started to deal with the goal $\leftarrow path(x, y)$, we store the tuple (x, y) in the relation named *input_path*. Tuples of this relation represent so called *input atoms* of the predicate *path*.

(6) Resolving the goal $\leftarrow path(x, y)$ by using a fresh variant $path(x_1, y_1) \leftarrow edge(x_1, y_1)$ of the clause (1) and the mgu $\{x_1/x, y_1/y\}$ results in the goal $\leftarrow edge(x, y)$.

(7) Resolving the goal $\leftarrow edge(x, y)$ by using the atom $edge(a, b)$ from I and the mgu $\{x/a, y/b\}$ results in the empty goal. We obtain the first computed answer $\{x/a, y/b\}$. As the main goal is $\leftarrow path(x, y)$, this computed answer can be represented by $path(a, b)$. To keep this answer, we store the tuple (a, b) in the relation named *ans_path*. Tuples of this relation represent so called *answer atoms* of the predicate *path*.

(8) Returning to the moment after Step 6 and resolving the goal $\leftarrow edge(x, y)$ by using the atom $edge(b, c)$ from I and the mgu $\{x/b, y/c\}$ results in the empty goal. We obtain the second computed answer $\{x/b, y/c\}$. Analogously as explained above, we add the tuple (b, c) to the relation *ans_path*.

(9) Returning to the moment after Step 5 and resolving the goal $\leftarrow path(x, y)$ by using a fresh variant $path(x_2, y_2) \leftarrow path(x_2, z_2), edge(z_2, y_2)$ of the clause (2) and the mgu $\{x_2/x, y_2/y\}$ results in the goal $\leftarrow path(x, z_2), edge(z_2, y)$. To resolve this goal using $path(x, z_2)$ as the selected atom (i.e., using the leftmost selection function), we can use:

- either a program clause defining *path* from P,
- or an answer atom of *path* that has been tabled earlier in *ans_path*.

Resolving the subgoal $\leftarrow path(x, z_2)$ by using a program clause defining *path* from P can be done in a similar way as we have been doing for the main goal $\leftarrow path(x, y)$ of the process. That is, one can continue by adding the tuple (x, z_2) to the relation *input_path* and so on. However, we first check whether this task can be ignored. Since (x, z_2) is an instance of (a fresh variant of) the existing tuple (x, y) in *input_path*, which stands for the goal $\leftarrow path(x, y)$ that has already been dealt with and is more general than the goal $\leftarrow path(x, z_2)$, storing (x, z_2) in *input_path* and processing the goal $\leftarrow path(x, z_2)$ in the usual way are unnecessary and therefore skipped.

Resolving the subgoal $\leftarrow path(x, z_2)$ by using an answer atom of *path* represented by a tuple in the relation *ans_path* is reported below. We have here a backtracking point, as there are a few of such tuples.

(10) Resolving the goal $\leftarrow path(x, z_2), edge(z_2, y)$ by selecting the atom $path(x, z_2)$ and using the tuple (a, b) in the relation *ans_path* and the mgu $\{x/a, z_2/b\}$ results in the goal $\leftarrow edge(b, y)$.

(11) Resolving the goal $\leftarrow edge(b, y)$ using the atom $edge(b, c)$ from I and the mgu $\{y/c\}$ results in the empty goal. We obtain the third computed answer $\{x/a,$

$y/c\}$ for the main goal $\leftarrow path(x, y)$, which is the restriction of the composition $\{x_2/x, y_2/y\}\{x/a, z_2/b\}\{y/c\}$ to the set $\{x, y\}$. We store this answer by adding the tuple (a, c) to the relation *ans_path*.

(12) Returning to the backtracking point mentioned at Step 9 and resolving the goal $\leftarrow path(x, z_2), edge(z_2, y)$ by selecting the atom $path(x, z_2)$ and using the tuple (b, c) in the relation *ans_path* and the mgu $\{x/b, z_2/c\}$ results in the goal $\leftarrow edge(c, y)$. This goal cannot be resolved by using any atom of *edge* from I. So, we finish this search branch (derivation).

(13) Returning to the backtracking point mentioned at Step 9 and resolving the goal $\leftarrow path(x, z_2), edge(z_2, y)$ by selecting the atom $path(x, z_2)$ and using the tuple (a, c) in the relation *ans_path* and the mgu $\{x/a, z_2/c\}$ results in the goal $\leftarrow edge(c, y)$. Once again, this goal cannot be resolved by using any atom of *edge* from I. As there are no active backtracking points, we finish the process. The computed answers are represented by the tuples in the relation *ans_path*.[8]

Consider the return to the backtracking point mentioned at Step 9 and the continuation at Step 13. It uses the tuple (a, c), which was added to the relation *ans_path* at Step 11, i.e., after the creation of the backtracking point. The considered example is simple and after Step 13 we can finish the process. But, what would happen if the used Datalog program was more complicated and the search tree (as displayed in Figure 1) had the third branch departing from the root with more computed answers? How could they be supplied for resolving the subgoal $\leftarrow path(x, z_2)$ in the second branch departing from the root? SLD-resolution systems with tabulation like OLDT use the "suspension-resumption mechanism" and the "stack-wise representation" to deal with this problem [42]. Both of these techniques are tuple-oriented (tuple-at-a-time) and not suitable for processing queries to deductive databases, as the task should be done set-at-a-time in order to deal with big data and reduce the number of accesses to the secondary storage.

3.2 Query-Subquery Nets as Data Structures for Processing Queries

Let us discuss a design of data structures for processing queries by simulating SLD-resolution with tabulation in the way so that the processing can be done set-at-a-time and every strategy for searching for answers (called a *control strategy*) can be applied. That leads to the notion of query-subquery net.

3.2.1 An Illustrative Example

Before defining query-subquery nets formally, we discuss the data structures needed for processing queries to the Datalog database (P_1, I) considered in Section 3.1.

[8] In the general case, only answer atoms that are instances of the main input atom are taken. In this concrete case, all the answer atoms $path(a, b)$, $path(b, c)$ and $path(a, c)$ are instances of the main input atom $path(x, y)$.

In general, we want to use a net for the processing, called a *query-subquery net* (or *QSQ-net* for short). It is a directed graph with nodes being appropriate data structures.

As discussed in Section 3.1, the approach uses the relations *input_path* and *ans_path*, so let the net have two nodes named ***input_path*** and ***ans_path***. Each of these nodes has an attribute named ***tuples***, which represents the corresponding relation. Thus, by writing *tuples*(*input_path*) (resp. *tuples*(*ans_path*)) we have in mind the relation *input_path* (resp. *ans_path*) mentioned in Section 3.1.

Tuples in the node *input_path* stand for goal atoms of the predicate *path*. That is, a tuple $\bar{t} \in tuples(input\_path)$ stands for the goal $\leftarrow path(\bar{t})$. Assume that, when a tuple is added to *tuples*(*input_path*), its variables have already been renamed so that they do not occur in the Datalog program P_1. In this way, when resolve a goal using a program clause of P_1, we do not have to rename variables of the program clause.

How can a tuple $\bar{t} \in tuples(input\_path)$ be processed? We can resolve the goal $\leftarrow path(\bar{t})$ by using the program clause (1) or (2).

- Consider the case when the goal $\leftarrow path(\bar{t})$ is resolved by using the program clause (1) (i.e., $path(x, y) \leftarrow edge(x, y)$). To do the task, let the node *input_path* have a connection to a node named $\boldsymbol{pre\_filter}_1$, where the subscript denotes the program clause's number. As attributes of pre_filter_1, we have $\boldsymbol{atom}(pre\_filter_1) = path(x, y)$, which is the head of the program clause, and $\boldsymbol{post\_vars}(pre\_filter_1) = \{x, y\}$, which is the set of variables occurring in the body of the program clause. Then, the task can be done by transferring the tuple $\bar{t}$ from the node *input_path* to the node pre_filter_1. At pre_filter_1, $path(\bar{t})$ is unified with $atom(pre\_filter_1)$ (i.e., $path(x, y)$) by using an mgu γ, and the pair $(\bar{t}\gamma, \gamma_{|post\_vars(pre\_filter_1)})$ is transfered to the unique successor of pre_filter_1, which is named $filter_{1,1}$. The tuple $\bar{t}\gamma$ in that pair is needed for further computation. In general, such tuples will be updated on-the-fly by taking into account subsequent mgu's in the derivation and at the end will represent answers for the goal related to the tuple taken from *input_path*. Similarly, the substitution $\gamma_{|post\_vars(pre\_filter_1)}$ is also needed for further computation. We restrict γ to $post\_vars(pre\_filter_1)$ because the other bindings in γ are redundant for further computation. We call the pair $(\bar{t}\gamma, \gamma_{|post\_vars(pre\_filter_1)})$ a *subquery*.

  After resolving the goal $\leftarrow path(\bar{t})$ using the program clause (1) (i.e., $path(x, y) \leftarrow edge(x, y)$) and the mgu γ, the resulting resolvent is $\leftarrow edge(x, y)\gamma$. So, the node $filter_{1,1}$ is related to processing this goal. In general, the subscript (i, j) of a node $\boldsymbol{filter}_{i,j}$ states that the node is related with the atom numbered j in the body of the program clause numbered i. As attributes of $filter_{1,1}$, at least we need $\boldsymbol{atom}(filter_{1,1}) = edge(x, y)$. For convenience, we also use the attributes $\boldsymbol{pred}(filter_{1,1}) = edge$ (the predicate of $atom(filter_{1,1})$) and $\boldsymbol{kind}(filter_{1,1}) = extensional$ (the kind of the predicate *edge*). For the general case of $filter_{i,j}$, we also need the attribute $\boldsymbol{pre\_vars}(filter_{i,j})$ (resp. $\boldsymbol{post\_vars}(filter_{i,j})$) for keeping variables occurring in the atoms numbered from j (resp. $j + 1$) in the body of

the program clause numbered i. For the currently considered example, we have $pre\_vars(filter_{1,1}) = \{x, y\}$ and $post\_vars(filter_{1,1}) = \emptyset$.

What should be done with the subquery $(\bar{t}\gamma, \gamma_{|post\_vars(pre\_filter_1)})$ transferred to $filter_{1,1}$ from pre_filter_1? We can either process it immediately or store it in $filter_{1,1}$ in order to accumulate more and more subqueries at the node $filter_{1,1}$ before processing them set-at-a-time. The Boolean option for this is called the *memorizing type* of $filter_{1,1}$ and denoted by $\boldsymbol{T}(filter_{1,1})$. In the case when $T(filter_{1,1}) = true$, if the subquery $(\bar{t}\gamma, \gamma_{|post\_vars(pre\_filter_1)})$ has not yet been considered for the node $filter_{1,1}$, we add it to the relations $\boldsymbol{subqueries}(filter_{1,1})$ and $\boldsymbol{unprocessed\_subqueries}(filter_{1,1})$, which are additional attributes of $filter_{1,1}$. A subquery is said to have already been considered for $filter_{1,1}$ if it is less general than a subquery from $subqueries(filter_{1,1})$ (see Definition 5). When a subquery from $unprocessed\_subqueries(filter_{1,1})$ is processed, it is deleted from this relation.

For abbreviation, let $(\bar{t}', \delta)$ denote $(\bar{t}\gamma, \gamma_{|post\_vars(pre\_filter_1)})$. How can the subquery $(\bar{t}', \delta)$ be processed at the node $filter_{1,1}$? That is, how can the earlier mentioned goal $\leftarrow edge(x, y)\gamma$ be processed at $filter_{1,1}$? We unify $atom(filter_{1,1})\delta$ (i.e., $edge(x, y)\gamma$) with each atom $edge(\bar{s}) \in I$ using an mgu γ' and transfer the tuple $\bar{t}'\gamma'$ to the unique successor of $filter_{1,1}$, which is named $post\_filter_1$. In general, each node $filter_{i,j}$ with $kind(filter_{i,j}) = extensional$ has exactly one successor, which is either $filter_{i,j+1}$ (if it exists) or $\boldsymbol{post\_filter}_i$. For the currently considered example, we do not have a node named $filter_{1,2}$ because the program clause (1) has only one atom in its body.

What should be done with the tuple $\bar{t}'\gamma'$ transferred to $post\_filter_1$ from $filter_{1,1}$? We have that $\gamma\gamma'_{|Vars(\bar{t})}$ is a computed answer for the goal $\leftarrow path(\bar{t})$. Thus, the tuple $\bar{t}'\gamma' = \bar{t}\gamma\gamma'$ specifies this computed answer. So, let the node $post\_filter_1$ have a connection to the node ans_path, then all we need to do is to transfer the tuple $\bar{t}'\gamma'$ through this connection to add it to the relation $tuples(ans\_path)$.

Summing up, to process a tuple from $input\_path$ by using the program clause (1) (i.e., $path(x, y) \leftarrow edge(x, y)$), the designed QSQ-net uses the following path of nodes with appropriate attributes:

$$input\_path \longrightarrow pre\_filter_1 \longrightarrow filter_{1,1} \longrightarrow post\_filter_1 \longrightarrow ans\_path.$$

- Consider the case when the goal $\leftarrow path(\bar{t})$ is resolved by using the program clause (2) (i.e., $path(x, y) \leftarrow path(x, z), edge(z, y)$). Analogously as for the previous case, the designed QSQ-net uses the following path of nodes:

  $$input\_path \longrightarrow pre\_filter_2 \longrightarrow filter_{2,1} \longrightarrow filter_{2,2} \longrightarrow post\_filter_2 \longrightarrow ans\_path$$

  where:

  - $atom(pre\_filter_2) = path(x, y)$,

- $atom(filter_{2,1}) = path(x, z)$ and $kind(filter_{2,1}) = intensional$,
- $atom(filter_{2,2}) = edge(z, y)$ and $kind(filter_{2,2}) = extensional$.

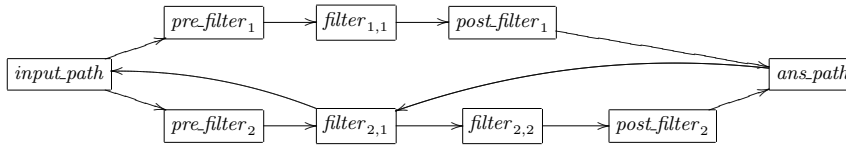
Consider a subquery $(\vec{t}', \delta)$ in the node $filter_{2,1}$. It stands for the goal $\leftarrow path(x, z)\delta, edge(z, y)\delta$. Let us pay attention to the subgoal $\leftarrow path(x, z)\delta$. As demonstrated for SLD-resolution with tabulation, to process this subgoal we should consider adding an appropriate tuple to *input_path*. In addition, to resolve the subgoal $\leftarrow path(x, z)\delta$ we can use not only the program clauses of P_1 but also the answer atoms of *path* represented by the tuples stored in *ans_path*.

To deal with the first matter, let the node $filter_{2,1}$ have a connection to the node *input_path*, then we can transfer the tuple $(x, z)\delta$ through that connection. If this tuple has not yet been considered for *input_path*, we add a fresh variant of it to $tuples(input\_path)$. A tuple is said to have already been considered for *input_path* if its fresh variant is an instance of a tuple from $tuples(input\_path)$.

Like the case of $filter_{1,1}$, the attribute $unprocessed\_subqueries(filter_{2,1})$ of $filter_{2,1}$ keeps subqueries that have not been processed at $filter_{2,1}$ to produce data to transfer to $filter_{2,2}$. The node $filter_{2,1}$ has, however, also a connection to the node *input_path*. So, we also need an attribute $\boldsymbol{unprocessed\_subqueries_2}(filter_{2,1})$ to keep subqueries that have not been processed at $filter_{2,1}$ to produce data to transfer to *input_path*.

At the node $filter_{2,1}$, to resolve the subgoal $\leftarrow path(x, z)\delta$ by using the answer atoms of *path* represented by the tuples stored in *ans_path*, we need a connection from the node *ans_path* to the node $filter_{2,1}$. New tuples added to *ans_path* are transferred through that connection and accumulated in the relation $\boldsymbol{unprocessed\_tuples}(filter_{2,1})$ before being processed (at some later steps). This relation is an additional attribute of $filter_{2,1}$.

Summing up, the QSQ-net designed for the considered Datalog program has the following topological structure:



It is illustrated in Figure 2 together with attributes of the nodes. The node *input_path* has an attribute $\boldsymbol{unprocessed}(E)$ for each outgoing edge E. A tuple $\overline{t} \in unprocessed(E)$ at the node *input_path* means that the tuple $\overline{t}$ has not been transferred from this node through the edge E (i.e., it has not been processed at the node *input_path* for the edge E). The case of the node *ans_path* is similar. Also note that we do not use the attribute $T(filter_{2,1})$ because $kind(filter_{2,1}) = intensional$. In other words, subqueries transferred to a node $filter_{i,j}$ with $kind(filter_{i,j}) = intensional$ are always memorized.

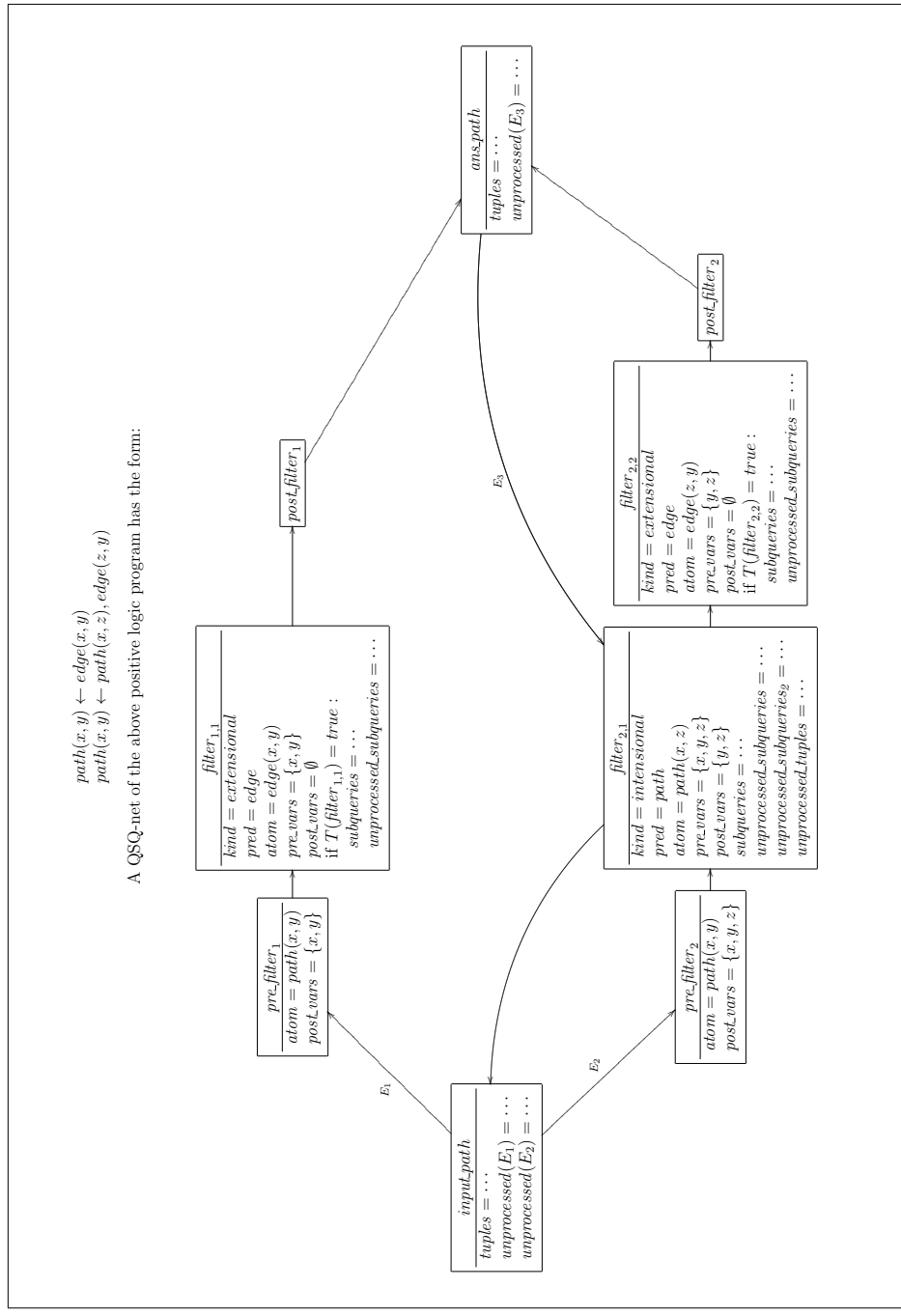


Figure 2. An example of a QSQ-net, where *path* is an intensional predicate, *edge* is an extensional predicate, and x, y, z are variables

3.2.2 A Formal Definition of Query-Subquery Nets

We now recall a formal definition of QSQ-nets [11]. From now to the end of Section 3, let P be a Datalog program and $\varphi_1, \ldots, \varphi_m$ all the program clauses of P, with $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$ for $1 \leq i \leq m$, where $n_i \geq 0$.

Definition 2. A *query-subquery net structure* (*QSQN structure* for short) of P is a tuple (V, E, T) such that:

- V is a set of nodes that consists of:
  - $input\_p$ and ans_p, for each intensional predicate p of P,
  - pre_filter_i, $filter_{i,1}$, ..., $filter_{i,n_i}$, $post\_filter_i$, for each $1 \leq i \leq m$.
- E is a set of edges that consists of:
  - $(filter_{i,1}, filter_{i,2})$, ..., $(filter_{i,n_i-1}, filter_{i,n_i})$, for each $1 \leq i \leq m$,
  - $(pre\_filter_i, filter_{i,1})$ and $(filter_{i,n_i}, post\_filter_i)$, for each $1 \leq i \leq m$ with $n_i \geq 1$,
  - $(pre\_filter_i, post\_filter_i)$, for each $1 \leq i \leq m$ with $n_i = 0$,
  - $(input\_p, pre\_filter_i)$ and $(post\_filter_i, ans\_p)$, for each $1 \leq i \leq m$, where p is the predicate of A_i,
  - $(filter_{i,j}, input\_p)$ and $(ans\_p, filter_{i,j})$, for each intensional predicate p and each $1 \leq i \leq m$ and $1 \leq j \leq n_i$ such that $B_{i,j}$ is a literal of p.[9]
- T is a function, called the *memorizing type* of the QSQN structure, mapping each node $filter_{i,j} \in V$ such that the predicate of $B_{i,j}$ is extensional to *true* or *false*.[10]

In a QSQN structure (V, E, T) of P, if $(v, w) \in E$ then we call w a *successor* of v, and v a *predecessor* of w. Note that V and E are uniquely specified by P. We call the pair (V, E) the *QSQN topological structure* of P.

By a *subquery* we mean a pair of the form $(\bar{t}, \delta)$, where $\bar{t}$ is a tuple of terms and δ an idempotent substitution such that $dom(\delta) \cap Vars(\bar{t}) = \emptyset$.

Definition 3. A *query-subquery net* (*QSQ-net* for short) of P is a tuple (V, E, T, C) such that (V, E, T) is a QSQN structure of P, C is a mapping that associates each node $v \in V$ with a structure called the *contents* of v, and the following conditions are satisfied:

- $C(v)$, where $v = input\_p$ or $v = ans\_p$, consists of:
  - $tuples(v)$: a set of tuples of terms with the same arity as p,

[9] We use the term "literal" here instead of "atom" or "positive literal" so that the definition can be extended for stratified Datalog$^\neg$ easily.

[10] Recall that the aim of T is that if $T(filter_{i,j}) = false$ then subqueries for $filter_{i,j}$ are always processed immediately without being accumulated at $filter_{i,j}$.

  - $unprocessed(v, w)$ for each $(v, w) \in E$: a subset of $tuples(v)$.

- $C(v)$, where $v = pre\_filter_i$, consists of:
  - $atom(v) = A_i$ and $post\_vars(v) = Vars((B_{i,1}, \ldots, B_{i,n_i}))$.
- $C(v)$, where $v = post\_filter_i$, is empty.
- $C(v)$, where $v = filter_{i,j}$ and p is the predicate of $B_{i,j}$, consists of:
  - $kind(v) = extensional$ if p is extensional, and $kind(v) = intensional$ otherwise,
  - $pred(v) = p$ and $atom(v) = B_{i,j}$,
  - $pre\_vars(v) = Vars((B_{i,j}, \ldots, B_{i,n_i}))$ and $post\_vars(v) = Vars((B_{i,j+1}, \ldots, B_{i,n_i}))$,
  - $subqueries(v)$: a set of subqueries $(\bar{t}, \delta)$ such that $\bar{t}$ has the same arity as the predicate of A_i,
  - $unprocessed\_subqueries(v) \subseteq subqueries(v)$,
  - in the case when p is intensional:
    - $unprocessed\_subqueries_2(v) \subseteq subqueries(v)$,
    - $unprocessed\_tuples(v)$: a set of tuples of terms with the same arity as p.
- if $v = filter_{i,j}$, $kind(v) = extensional$ and $T(v) = false$, then $subqueries(v) = \emptyset$ (and both $subqueries(v)$ and $unprocessed\_subqueries(v)$ can be ignored).

We use the term QSQ-net as a noun and QSQN mostly as an adjective. Both of them are abbreviations of "query-subquery net". When standing alone, QSQN refers to the related evaluation framework, which is specified and discussed in the next subsection.

An *empty QSQ-net* of P is a QSQ-net of P such that all the sets of the form $tuples(v)$, $unprocessed(v, w)$, $subqueries(v)$, $unprocessed\_subqueries(v)$, $unprocessed\_subqueries_2(v)$ or $unprocessed\_tuples(v)$ of the net are empty.

In a QSQ-net, if $v = pre\_filter_i$ or $v = post\_filter_i$ or ($v = filter_{i,j}$ and $kind(v) = extensional$), then v has exactly one successor, which we denote by $succ(v)$.

If v is $filter_{i,j}$ with $kind(v) = intensional$ and $pred(v) = p$, then v has exactly two successors. In that case, let $succ(v)$ be $filter_{i,j+1}$ if $n_i > j$, and $post\_filter_i$ otherwise, and let $succ_2(v) = input\_p$. The set $unprocessed\_subqueries(v)$ is used for the edge $(v, succ(v))$, while $unprocessed\_subqueries_2(v)$ is used for the edge $(v, succ_2(v))$.

For convenience, we denote $pre\_vars(post\_filter_i) = \emptyset$. Thus, if $v = succ(u)$, where u is pre_filter_i of $filter_{i,j}$, then $post\_vars(u) = pre\_vars(v)$.

3.3 The QSQN Framework for Evaluating Queries

Based on QSQ-nets, in [11, 9] we proposed the QSQN framework for evaluating queries to Horn knowledge bases. We recall here its version restricted to the case without function symbols for evaluating queries to Datalog databases.

The QSQN method for evaluating a query $q(\overline{x})$ to a Datalog database (P, I) can be described informally as follows:

1. create an empty QSQ-net (V, E, T, C) of P;
2. let $\overline{x}'$ be a fresh variant of $\overline{x}$;
3. add $\overline{x}'$ to $tuples(input\_q)$;
4. for each $(input\_q, v) \in E$, add $\overline{x}'$ to $unprocessed(input\_q, v)$;
5. while there are edges $(u, v) \in E$ such that there are data at u to be processed for the edge (u, v), do:
   (a) select such an edge (any strategy can be used);
   (b) process the data at u to produce data to transfer through the edge (u, v);
6. return $tuples(ans\_q)$.

A strategy for selecting an edge at the step 5a is called a *control strategy*. As every strategy can be used, the QSQN method is really a framework for evaluating queries.

Definition 4 (`active-edge`)**.** The data at a node u to be processed for an edge $(u, v) \in E$ are:

- $unprocessed(u, v)$ if u is $input\_p$ or ans_p (for some p),
- $unprocessed\_subqueries(u)$ if u is $filter_{i,j}$, $kind(u) = extensional$ and $T(u) = true$,
- $unprocessed\_subqueries(u) \cup unprocessed\_tuples(u)$ if u is $filter_{i,j}$, $kind(u) = intensional$ and $v = succ(u)$,
- $unprocessed\_subqueries_2(u)$ if u is $filter_{i,j}$, $kind(u) = intensional$ and $v = succ_2(u)$,
- the empty set otherwise.

Let `active-edge`(u, v) be the Boolean function stating that the set representing data at the node u to be processed for the edge (u, v) is not empty. If `active-edge`(u, v) holds, then we say that the edge (u, v) is *active*.

We will specify what is "processing the data at a node u to produce data to transfer through an edge (u, v)". We call that processing "firing the edge (u, v)" and let `fire`(u, v) be a procedure for doing it. Thus, the main loop of the QSQN method (i.e., the step 5) can be rewritten to: "while there exists $(u, v) \in E$ such that `active-edge`(u, v) holds, select such a pair (u, v) and `fire`(u, v)".

The types of data transferred through edges of a QSQ-net are as follows:

- data transferred through an edge of the form $(input\_p, v)$, $(v, input\_p)$, (v, ans_p) or (ans_p, v) are a set of tuples (of terms) of the same arity as p,

- data transferred through an edge (u, v) with $v = \mathit{filter}_{i,j}$ and u not being of the form $\mathit{ans\_p}$ are a set of subqueries that can be added to $\mathit{subqueries}(v)$,
- data transferred through an edge $(v, \mathit{post\_filter}_i)$ are a set of subqueries $(\bar{t}, \varepsilon)$ such that $\bar{t}$ is a tuple (of constants) of the same arity as the predicate of A_i.

Before specifying the transfer of data through edges, we first give some auxiliary definitions.

Definition 5. If $(\bar{t}, \delta)$ and $(\bar{t}', \delta')$ are subqueries that can be transferred through an edge to v, then we say that $(\bar{t}, \delta)$ is *more general* than $(\bar{t}', \delta')$ w.r.t. v, and $(\bar{t}', \delta')$ is *less general* than $(\bar{t}, \delta)$ w.r.t. v, if there exists a substitution γ such that $\bar{t}\gamma = \bar{t}'$ and $(\delta\gamma)_{|\mathit{pre\_vars}(v)} = \delta'$.

Definition 6 (`add-subquery`)**.** We define `add-subquery`$(\bar{t}, \delta, \Gamma, v)$ as a procedure that adds the subquery $(\bar{t}, \delta)$ to the set Γ but keeps in Γ only the most general subqueries w.r.t. the node v. It can be implemented as follows:

if no subquery in Γ is more general than $(\bar{t}, \delta)$ w.r.t. v then:

- delete from Γ all subqueries less general than $(\bar{t}, \delta)$ w.r.t. v;
- add $(\bar{t}, \delta)$ to Γ.

Definition 7 (`add-tuple`)**.** We define `add-tuple`$(\bar{t}, \Gamma)$ as a procedure that adds a fresh variant of the tuple $\bar{t}$ to the set Γ but keeps in Γ only the most general tuples. It can be implemented as follows:

- let $\bar{t}'$ be a fresh variant of $\bar{t}$;
- if $\bar{t}'$ is not an instance of any tuple from Γ then:
  - delete from Γ all tuples that are instances of $\bar{t}'$;
  - add $\bar{t}'$ to Γ.

Definition 8 (`transfer`)**.** We define `transfer`(D, u, v) as the following procedure for transferring the data D through the edge (u, v), using some subroutines specified later:

1. if $D = \emptyset$ then return;
2. if v is $\mathit{post\_filter}_i$ then `transfer`$(\{\bar{t} \mid (\bar{t}, \varepsilon) \in D\}, v, \mathit{succ}(v))$;
3. else if u is $\mathit{ans\_p}$ then $\mathit{unprocessed\_tuples}(v) := \mathit{unprocessed\_tuples}(v) \cup D$;
4. else if v is $\mathit{ans\_p}$ then `transfer`$_1(D, u, v)$;
5. else if v is $\mathit{input\_p}$ then `transfer`$_2(D, u, v)$;
6. else if u is $\mathit{input\_p}$ then `transfer`$_3(D, u, v)$;
7. else if v is $\mathit{filter}_{i,j}$, $\mathit{kind}(v) = \mathit{extensional}$ and $T(v) = \mathit{false}$ then `transfer`$_4(D, u, v)$;
8. else `transfer`$_5(D, u, v)$.

Observe that the case specified by the last "else" in the above definition (i.e., the step 8) is characterized by the conjunction that v is $filter_{i,j}$, ($kind(v) = extensional$ and $T(v) = true$ or $kind(v) = intensional$) and u is not of the form ans_p. The procedure $\texttt{transfer}_5(D, u, v)$ is defined only for this case, with D being a set of subqueries. Roughly speaking, it just accumulates the subqueries from D at v but ignores the ones that have already been considered for v and keeps only the most general ones.

Definition 9 ($\texttt{transfer}_5$)**.** The procedure $\texttt{transfer}_5(D, u, v)$ for the aforementioned case is (can be) implemented as follows:

for each $(\bar{t}, \delta) \in D$ do:

if no subquery in $subqueries(v)$ is more general than $(\bar{t}, \delta)$ w.r.t. v then

- delete from $subqueries(v)$ and $unprocessed\_subqueries(v)$ all subqueries less general than $(\bar{t}, \delta)$ w.r.t. v;
- add $(\bar{t}, \delta)$ to both $subqueries(v)$ and $unprocessed\_subqueries(v)$;
- if $kind(v) = intensional$ then:
  - delete from $unprocessed\_subqueries_2(v)$ all subqueries less general than $(\bar{t}, \delta)$ w.r.t. v;
  - add $(\bar{t}, \delta)$ to $unprocessed\_subqueries_2(v)$.

Definition 10 ($\texttt{transfer}_1$)**.** The procedure $\texttt{transfer}_1(D, u, v)$ for the case when $v = ans\_p$ (and $u = post\_filter_i$ for some i and D is a set of tuples of constants) is (can be) implemented as follows:

for each $\bar{t} \in D - tuples(v)$ do:

- add $\bar{t}$ to $tuples(v)$;
- for each $(v, w) \in E$, add $\bar{t}$ to $unprocessed(v, w)$.

The procedure $\texttt{transfer}_2(D, u, v)$ is defined only for the case when $v = input\_p$, with $u = filter_{i,j}$ (for some i and j) and D being a set of tuples of terms. Roughly speaking, it just accumulates fresh variants of the tuples from D at v but ignores the ones that have already been considered for v and keeps only the most general ones. Recall that we apply variable renaming for input atoms but not program clauses.

Definition 11 ($\texttt{transfer}_2$)**.** The procedure $\texttt{transfer}_2(D, u, v)$ for the case $v = input\_p$ is (can be) implemented as follows:

for each $\bar{t} \in D$ do:

- let $\bar{t}'$ be a fresh variant of $\bar{t}$;
- if $\bar{t}'$ is not an instance of any tuple from $tuples(v)$ then
  - delete from $tuples(v)$ all tuples that are instances of $\bar{t}'$;
  - add $\bar{t}'$ to $tuples(v)$;

- for each $(v, w) \in E$ do
  - delete from $unprocessed(v, w)$ all tuples that are instances of $\bar{t}'$;
  - add $\bar{t}'$ to $unprocessed(v, w)$.

The procedure $\texttt{transfer}_3(D, u, v)$ is defined only for the case when u is $input\_p$, with $v = pre\_filter_i$ for some i. It does not accumulate D at v but processes it immediately to create data, which are then transferred to $succ(v)$. The reader can recall the example on page 34 of using pre_filter_1 to process a tuple $\bar{t} \in tuples(input\_path)$. In general, the node pre_filter_i is used for resolving an input atom (represented by a tuple from D) by unifying it with the head of the program clause numbered i.

Definition 12 ($\texttt{transfer}_3$)**.** The procedure $\texttt{transfer}_3(D, u, v)$ for the case u is $input\_p$ is (can be) implemented as follows:

- $\Gamma := \emptyset$;
- for each $\bar{t} \in D$ do

  if $p(\bar{t})$ and $atom(v)$ are unifiable by an mgu γ then
  $\texttt{add-subquery}(\bar{t}\gamma, \gamma_{|post\_vars(v)}, \Gamma, succ(v))$;

- $\texttt{transfer}(\Gamma, v, succ(v))$.

The procedure $\texttt{transfer}_4(D, u, v)$ is defined only for the case when u is not of the form ans_p, v is $filter_{i,j}$, $kind(v) = extensional$ and $T(v) = false$. According to the intention of the memorizing type T, it does not accumulate the subqueries from D at v but processes them immediately to create data, which are then transferred to $succ(v)$. The reader can recall the example on page 35 of processing a subquery $(\bar{t}', \delta)$ at the node $filter_{1,1}$.

Definition 13 ($\texttt{transfer}_4$)**.** The procedure $\texttt{transfer}_4(D, u, v)$ for the aforementioned case is (can be) implemented as follows:

- let $p = pred(v)$ and set $\Gamma := \emptyset$;
- for each $(\bar{t}, \delta) \in D$ and each $\bar{t}' \in I(p)$ do

  if $atom(v)\delta$ and $p(\bar{t}')$ are unifiable by an mgu γ then
  $\texttt{add-subquery}(\bar{t}\gamma, (\delta\gamma)_{|post\_vars(v)}, \Gamma, succ(v))$;

- $\texttt{transfer}(\Gamma, v, succ(v))$.

We have fully specified the procedure $\texttt{transfer}(D, u, v)$ for transferring the data D through the edge (u, v). We now specify the procedure $\texttt{fire}(u, v)$ for "firing the edge (u, v)", i.e., for processing the data at u to produce data to transfer through the edge (u, v). Recall that this procedure is called only for active edges (u, v).

Definition 14 ($\texttt{fire}$)**.** The procedure $\texttt{fire}(u, v)$ is implemented as follows, using some subroutines specified later:

- if u is *input_p* or *ans_p* then
  - `transfer`$(\mathit{unprocessed}(u,v), u, v)$;
  - $\mathit{unprocessed}(u,v) := \emptyset$;
- else if v is *input_p* then `fire`$_1(u,v)$;
- else if u is $\mathit{filter}_{i,j}$ and $\mathit{kind}(u) = \mathit{extensional}$ then `fire`$_2(u,v)$;
- else if u is $\mathit{filter}_{i,j}$ and $\mathit{kind}(u) = \mathit{intensional}$ then `fire`$_3(u,v)$.

The procedure `fire`$_1(u,v)$ is defined only for the case when v is *input_p*, with u being $\mathit{filter}_{i,j}$ (for some i and j) and $\mathit{kind}(u) = \mathit{intensional}$. For each subquery $(\overline{t}, \delta)$ at u that has not yet been processed for the edge (u,v), the procedure transfers a fresh variant of $\overline{t}'$, where $p(\overline{t}') = \mathit{atom}(u)\delta$, through the edge (u,v) in order to add it to $\mathit{tuples}(\mathit{input\_p})$ if it has not been considered for v. The reader can recall the example on page 36 of dealing with the edge $(\mathit{filter}_{2,1}, \mathit{input\_path})$.

Definition 15 (`fire`$_1$)**.** The procedure `fire`$_1(u,v)$ for the case v is *input_p* is (can be) implemented as follows:

- let $p = \mathit{pred}(u)$ and set $\Gamma := \emptyset$;
- for each $(\overline{t}, \delta) \in \mathit{unprocessed\_subqueries}_2(u)$ do
  - let $p(\overline{t}') = \mathit{atom}(u)\delta$;
  - `add-tuple`$(\overline{t}', \Gamma)$;
- $\mathit{unprocessed\_subqueries}_2(u) := \emptyset$;
- `transfer`(Γ, u, v).

The procedure `fire`$_2(u,v)$ is defined only for the case when u is $\mathit{filter}_{i,j}$ and $\mathit{kind}(u) = \mathit{extensional}$. In this case, as the edge (u,v) is active, we must have that $T(u) = \mathit{true}$. Before specifying this procedure, recall the procedure `transfer`$_4(D, x, u')$ defined in Definition 13 for the case when u' is $\mathit{filter}_{i,j} = \mathit{succ}(x)$, $\mathit{kind}(u') = \mathit{extensional}$ and $T(u') = \mathit{false}$. This latter procedure processes the data D immediately at u' to create data, which are then transferred to $\mathit{succ}(u')$. The procedure `fire`$_2(u,v)$ processes $\mathit{unprocessed\_subqueries}(u,v)$ at u in a similar way and then empties $\mathit{unprocessed\_subqueries}(u,v)$. The reader can also recall the example on page 35 of processing a subquery at the node $\mathit{filter}_{1,1}$.

Definition 16 (`fire`$_2$)**.** The procedure `fire`$_2(u,v)$ for the aforementioned case is (can be) implemented as follows:

- let $p = \mathit{pred}(u)$ and set $\Gamma := \emptyset$;

- for each $(\bar{t}, \delta) \in \mathit{unprocessed\_subqueries}(u)$ and each $\bar{t}' \in I(p)$ do

  if $\mathit{atom}(u)\delta$ and $p(\bar{t}')$ are unifiable by an mgu γ then
  $\texttt{add-subquery}(\bar{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(u)}, \Gamma, v)$;

- $\mathit{unprocessed\_subqueries}(u) := \emptyset$;
- $\texttt{transfer}(\Gamma, u, v)$.

The procedure $\texttt{fire}_3(u, v)$ is defined only for the case when u is $\mathit{filter}_{i,j}$, $\mathit{kind} = \mathit{intensional}$ and $v = \mathit{succ}(u)$. Let $p = \mathit{pred}(u)$. For each subquery $(\bar{t}, \delta)$ at u and each tuple $\bar{t}' \in \mathit{tuples}(\mathit{ans\_}p)$, if they (as a pair) have not been processed at u, then the procedure processes them in a similar way as the procedure $\texttt{fire}_2(u, v)$ does for a subquery $(\bar{t}, \delta)$ at u and a tuple $\bar{t}'$ from the extensional relation of the predicate of $\mathit{atom}(u)$. Note, however, that for $\texttt{fire}_3(u, v)$ we have two subcases: either the subquery $(\bar{t}, \delta)$ has not been processed at u for the tuple $\bar{t}'$, or the tuple $\bar{t}'$ has not been processed at u for the subquery $(\bar{t}, \delta)$.

Definition 17 ($\texttt{fire}_3$)**.** The procedure $\texttt{fire}_3(u, v)$ for the aforementioned case is (can be) implemented as follows:

- let $p = \mathit{pred}(u)$ and set $\Gamma := \emptyset$;
- for each $(\bar{t}, \delta) \in \mathit{unprocessed\_subqueries}(u)$ and each $\bar{t}' \in \mathit{tuples}(\mathit{ans\_}p)$ do

  if $\mathit{atom}(u)\delta$ and $p(\bar{t}')$ are unifiable by an mgu γ then
  $\texttt{add-subquery}(\bar{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(u)}, \Gamma, v)$;

- $\mathit{unprocessed\_subqueries}(u) := \emptyset$;
- for each $(\bar{t}, \delta) \in \mathit{subqueries}(u)$ and each $\bar{t}' \in \mathit{unprocessed\_tuples}(u)$ do

  if $\mathit{atom}(u)\delta$ and $p(\bar{t}')$ are unifiable by an mgu γ then
  $\texttt{add-subquery}(\bar{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(u)}, \Gamma, v)$;

- $\mathit{unprocessed\_tuples}(u) := \emptyset$;
- $\texttt{transfer}(\Gamma, u, v)$.

We have fully specified the QSQN method for evaluating queries to Datalog databases. An example illustrating the QSQN method can be found in [9]. Note that, in this method, processing subqueries has been designed so that:

- every subquery that is subsumed by another one is ignored,
- for *input_* relations, every tuple that is subsumed by another one is ignored,
- the processing is divided into smaller steps which can be delayed at each node to maximize adjustability and allow various control strategies,
- the processing is done set-at-a-time (e.g., for all the unprocessed subqueries or tuples accumulated in a given node).

In [9, 32] we have proved that the QSQN method for evaluating queries to Datalog databases is sound, complete and has a PTIME data complexity.

4 QUERY-SUBQUERY NETS WITH STRATIFIED NEGATION

In this section, we present our framework called QSQN-STR for evaluating queries to stratified Datalog$^\neg$ databases. It is based on query-subquery nets that are extended to deal with stratified negation.

In what follows, let P be a stratified Datalog$^\neg$ program and $\varphi_1, \ldots, \varphi_m$ all the clauses of P, with $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$ for $1 \leq i \leq m$, where $n_i \geq 0$.

Definition 18. A *query-subquery net structure with stratified negation* (*QSQN-STR structure* for short) of P, is a tuple (V, E, T) defined as in the case of QSQN structure (see Definition 2) with the following modification:

> for each intensional predicate p and each $1 \leq i \leq m$ and $1 \leq j \leq n_i$ such that $B_{i,j}$ is a literal of p, the pair $(ans\_p, filter_{i,j})$ is an edge (i.e., belongs to E) iff $B_{i,j}$ is a positive literal.

The pair (V, E) is called the *QSQN-STR topological structure* of P.

Figure 3 illustrates the QSQN-STR topological structure of the stratified Datalog$^\neg$ program given in Example 1.

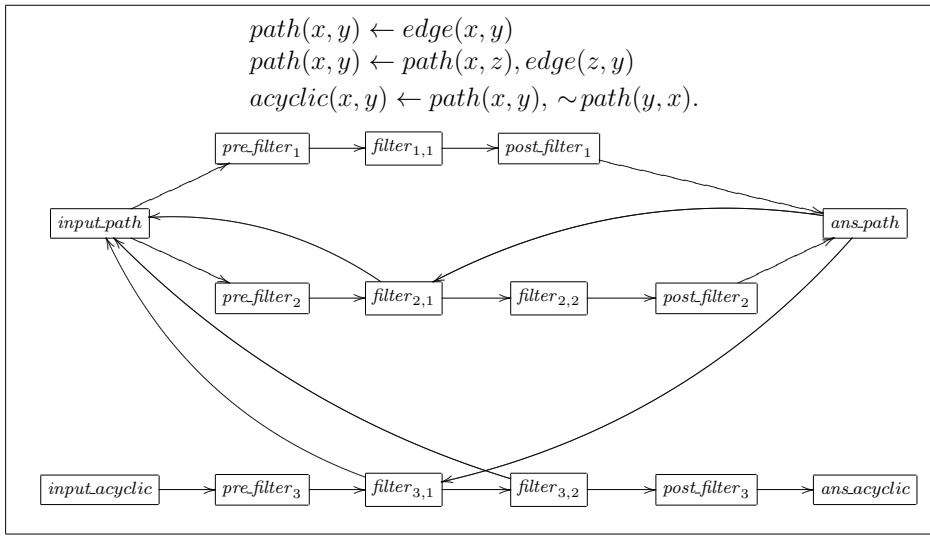


Figure 3. The QSQN-STR topological structure of the program given in Example 1

Definition 19. A *query-subquery net with stratified negation* (*QSQ-STR-net* for short)[11] of P, is a tuple (V, E, T, C) such that (V, E, T) is a QSQN-STR structure of P, and C is a mapping that associates each node $v \in V$ with a structure called

[11] We use the term QSQ-STR-net as a noun, and QSQN-STR mostly as an adjective.

the *contents* of v, which differs from the one for QSQ-net (see Definition 3) in the following:

If $v = \mathit{filter}_{i,j}$ and p is the predicate of $B_{i,j}$, then:

- $C(v)$ also contains $neg(v)$, where $neg(v) = \mathit{true}$ if $B_{i,j}$ is a negative literal, and $neg(v) = \mathit{false}$ otherwise,
- $atom(v)$ is redefined as follows: $atom(v) = B_{i,j}$ if $B_{i,j}$ is a positive literal, and $atom(v) = B'$ if $B_{i,j} = \sim B'$,
- in the case when p is intensional and $neg(v) = \mathit{true}$: $\mathit{unprocessed\_tuples}(v)$ is empty and can thus be ignored.

The notion of being *empty* is defined for QSQ-STR-net similarly as for QSQ-net.

The addition of the attribute $neg(v)$ and the modification of the attribute $atom(v)$ in the above definition are natural and do not require explanation. The below remark justifies the third difference stated in the above definition and the one in Definition 18.

From now on, by a *goal* we mean an expression of the form $\leftarrow B_1, \ldots, B_k$, where each B_i is a (positive or negative) literal.

Remark 2. Consider the QSQ-STR-net of the Datalog program P given in Example 1, whose structure is illustrated in Figure 3. We have $atom(\mathit{filter}_{3,2}) = path(y, x)$ and $neg(\mathit{filter}_{3,2}) = \mathit{true}$. Since P is safe, any $(\bar{t}, \delta) \in \mathit{subqueries}(\mathit{filter}_{3,2})$ has the properties that $\bar{t}$ is ground and δ has the form $\{x/c_1, y/c_2\}$ for some constants c_1 and c_2. The subquery $(\bar{t}, \delta)$ corresponds to the goal $\leftarrow \sim atom(\mathit{filter}_{3,2})\delta$, which is $\leftarrow \sim path(c_2, c_1)$. To resolve it, using the "negation as failure" approach, we deal with the goal $\leftarrow path(c_2, c_1)$. As this goal is ground, the answer can be either ε or failure. As usual, we transfer the tuple (c_2, c_1) through the edge $(\mathit{filter}_{3,2}, \mathit{input\_path})$ and add it to $\mathit{tuples}(\mathit{input\_path})$ if it is not an instance of another one in $\mathit{tuples}(\mathit{input\_path})$, and then proceed to check whether the tuple will be added to $\mathit{tuples}(\mathit{ans\_path})$. Despite that check, we do not need to transfer any tuple from $\mathit{ans\_path}$ through the edge $(\mathit{ans\_path}, \mathit{filter}_{3,2})$ to add to $\mathit{unprocessed\_tuples}(\mathit{filter}_{3,2})$. That is why we do not need the edge $(\mathit{ans\_path}, \mathit{filter}_{3,2})$ and the set $\mathit{unprocessed\_tuples}(\mathit{filter}_{3,2})$ will always be empty.

Based on QSQ-STR-nets we now specify our QSQN-STR method for evaluating queries to stratified Datalog$^\neg$ databases. We will use some subroutines defined earlier for the QSQN method, including `active-edge`(u, v), `add-subquery`$(\bar{t}, \delta, \Gamma, v)$ and `add-tuple`$(\bar{t}, \Gamma)$.

For transferring data D through an edge (u, v) the QSQN-STR method uses a procedure named `transfer`$'(D, u, v)$, which differs from `transfer`(D, u, v) only in processing the case when v is $\mathit{filter}_{i,j}$, $\mathit{kind}(v) = \mathit{extensional}$, $T(v) = \mathit{false}$ and $neg(v) = \mathit{true}$. In this case, a subquery $(\bar{t}, \delta) \in D$ corresponds to the goal $\leftarrow \sim atom(v)\delta, B_{i,j+1}\delta, \ldots, B_{i,n_i}\delta$. If $atom(v)\delta$ does not belong to the instance of the extensional predicate $p = \mathit{pred}(v)$, then the subquery is transferred further

to $succ(v)$ after restricting δ to $post\_vars(v)$. The task is done by the procedure $\texttt{transfer}'_{4b}(D, u, v)$ specified below.

Definition 20 ($\texttt{transfer}'_{4b}$)**.** The procedure $\texttt{transfer}'_{4b}(D, u, v)$ for the aforementioned case is (can be) implemented as follows:

- let $p = pred(v)$ and set $\Gamma := \emptyset$;
- for each $(\bar{t}, \delta) \in D$ do

  if $atom(v)\delta \notin \{p(\bar{t}') \mid \bar{t}' \in I(p)\}$ then
  $\texttt{add-subquery}(\bar{t}, \delta_{|post\_vars(u)}, \Gamma, succ(v))$;

- $\texttt{transfer}'(\Gamma, v, succ(v))$.

Recall that the procedures $\texttt{transfer}_1(D, u, v)$, $\texttt{transfer}_2(D, u, v)$ and $\texttt{transfer}_5(D, u, v)$ do not call the procedure $\texttt{transfer}$, but $\texttt{transfer}_3(D, u, v)$ and $\texttt{transfer}_4(D, u, v)$ do.

Definition 21 ($\texttt{transfer}'$, cf. Definition 8)**.** Let $\texttt{transfer}'_3(D, u, v)$ and $\texttt{transfer}'_4(D, u, v)$ be the procedures obtained from $\texttt{transfer}_3(D, u, v)$ and $\texttt{transfer}_4(D, u, v)$, respectively, by replacing the call $\texttt{transfer}(\Gamma, v, succ(v))$ with $\texttt{transfer}'(\Gamma, v, succ(v))$. Then, the procedure $\texttt{transfer}'(D, u, v)$ is (can be) implemented as follows:

1. if $D = \emptyset$ then return;
2. if v is $post\_filter_i$ then $\texttt{transfer}'(\{\bar{t} \mid (\bar{t}, \varepsilon) \in D\}, v, succ(v))$;
3. else if u is ans_p then $unprocessed\_tuples(v) := unprocessed\_tuples(v) \cup D$;
4. else if v is ans_p then $\texttt{transfer}_1(D, u, v)$;
5. else if v is $input\_p$ then $\texttt{transfer}_2(D, u, v)$;
6. else if u is $input\_p$ then $\texttt{transfer}'_3(D, u, v)$;
7. else if v is $filter_{i,j}$, $kind(v) = extensional$ and $T(v) = false$ then
   (a) if $neg(v) = false$ then $\texttt{transfer}'_4(D, u, v)$;
   (b) else $\texttt{transfer}'_{4b}(D, u, v)$;
8. else $\texttt{transfer}_5(D, u, v)$.

Assume that an edge (u, v) is active (i.e., $\texttt{active-edge}(u, v)$ holds) and is selected by an *admissible* control strategy, which will be specified later. Then, the QSQN-STR method uses a procedure named $\texttt{fire}'(u, v)$ instead of $\texttt{fire}(u, v)$ for processing the data at u to produce data to transfer through the edge (u, v). Without optimizations it would differ from $\texttt{fire}(u, v)$ only in processing the case when u is $filter_{i,j}$, $neg(u) = true$ and $v = succ(u)$. In this case, a subquery $(\bar{t}, \delta) \in unprocessed\_subqueries(u)$ corresponds to the goal $\leftarrow \sim atom(u)\delta, B_{i,j+1}\delta, \ldots, B_{i,n_i}\delta$.

Let $p = pred(u)$ and let R be $I(p)$ if $kind(u) = extensional$, and $tuples(ans\_p)$ otherwise. If $atom(u)\delta$ is different from $p(\overline{t}')$ for every $\overline{t}' \in R$, then the subquery is transferred to v after restricting δ to $post\_vars(u)$. The task is done by the procedure $\texttt{fire}'_4(u, v)$ specified below.

Definition 22 ($\texttt{fire}'_4$)**.** The procedure $\texttt{fire}'_4(u, v)$ for the aforementioned case is implemented as follows:

- let $p = pred(u)$ and set $\Gamma := \emptyset$;
- let R be $I(p)$ if $kind(u) = extensional$, and $tuples(ans\_p)$ otherwise;
- for each $(\overline{t}, \delta) \in unprocessed\_subqueries(u)$ do

  if $atom(u)\delta \notin \{p(\overline{t}') \mid \overline{t}' \in R\}$ then
  $\texttt{add-subquery}(\overline{t}, \delta_{|post\_vars(u)}, \Gamma, v)$;
- $unprocessed\_subqueries(u) := \emptyset$;
- $\texttt{transfer}'(\Gamma, u, v)$.

Observe that, when $u = filter_{i,j}$, $neg(u) = true$, $v = succ(u)$ and $\texttt{active-edge}(u, v)$ holds, if $kind(u) = extensional$, then $T(u) = true$. The procedure $\texttt{fire}'_4(u, v)$ for the subcase when $kind(u) = extensional$ (and $T(u) = true$) processes the data at u to transfer through the edge (u, v) in a similar way as the procedure $\texttt{transfer}'_{4b}(D, x, u')$ (see Definition 20) processes the data D immediately at u' to create data, which are then transferred to $succ(u')$. Here, u' plays a similar role as u, but $T(u') = false$, while $T(u) = true$.

Consider the procedure $\texttt{fire}'_4(u, v)$ for the subcase when $kind(u) = intensional$. For a subquery $(\overline{t}, \delta) \in unprocessed\_subqueries(u)$, the check whether $atom(u)\delta$ does not belong to $\{p(\overline{t}') \mid \overline{t} \in tuples(ans\_p)\}$ for $p = pred(u)$ should be done only at a suitable moment, i.e., when necessary work has been done to guarantee that an answer for the goal $\leftarrow atom(u)\delta$, if it exists, has been stored in $tuples(ans\_p)$ as the tuple $\overline{s}$ with $p(\overline{s}) = atom(u)\delta$. This means that, if $u = filter_{i,j}$, $neg(u) = true$, $kind(u) = intensional$ and $v = succ(u)$, then the edge (u, v) can be selected for "firing" (by $\texttt{fire}'(u, v)$, which calls $\texttt{fire}'_4(u, v)$) only when it is active and, additionally, satisfies appropriate conditions. In other words, the used control strategy (for selecting an edge to fire) should satisfy appropriate conditions. We will introduce a class of such control strategies shortly, which consists of so called *control strategies admissible w.r.t. strata's stability*.

The following definition formally specifies the procedure $\texttt{fire}'(u, v)$. Recall that this procedure is called only for active edges (u, v).

Definition 23 ($\texttt{fire}'$, cf. Definition 14)**.** Let $\texttt{fire}'_1(u, v)$, $\texttt{fire}'_2(u, v)$ and $\texttt{fire}'_3(u, v)$ be the procedures obtained from $\texttt{fire}_1(u, v)$, $\texttt{fire}_2(u, v)$ and $\texttt{fire}_3(u, v)$, respectively, by replacing the call $\texttt{transfer}(\Gamma, u, v)$ with $\texttt{transfer}'(\Gamma, u, v)$. Then, the procedure $\texttt{fire}'(u, v)$ is (can be) implemented as follows:

1. if u is *ans_p* then
   (a) $\texttt{transfer}'(\mathit{unprocessed}(u,v), u, v)$;
   (b) $\mathit{unprocessed}(u,v) := \emptyset$;
2. else if u is *input_p* then
   (a) $\texttt{transfer}'(\mathit{unprocessed}(u,v) - \mathit{tuples}(\mathit{ans\_p}), u, v)$;
   (b) $\mathit{unprocessed}(u,v) := \emptyset$;
3. else if v is *input_p* then $\texttt{fire}'_1(u,v)$;
4. else if u is $\mathit{filter}_{i,j}$, $\mathit{neg}(u) = \mathit{false}$ and $\mathit{kind}(u) = \mathit{extensional}$ then $\texttt{fire}'_2(u,v)$;
5. else if u is $\mathit{filter}_{i,j}$, $\mathit{neg}(u) = \mathit{false}$ and $\mathit{kind}(u) = \mathit{intensional}$ then $\texttt{fire}'_3(u,v)$;
6. else if u is $\mathit{filter}_{i,j}$ and $\mathit{neg}(u) = \mathit{true}$ then $\texttt{fire}'_4(u,v)$.

The exclusion of tuples belonging to $\mathit{tuples}(\mathit{ans\_p})$ from the transfer at the step 2a of $\texttt{fire}'(u,v)$ is an optimization. Note that every processed goal of the form $\leftarrow \sim p(\bar{t})$ is ground, and before processing the goal $\leftarrow p(\bar{t})$ (for "negation as failure") by using a program clause of P, one can check whether $\bar{t} \in \mathit{tuples}(\mathit{ans\_p})$. If so, then we already have the answer ε and can ignore the goal $\leftarrow p(\bar{t})$. One can also optimize the procedure $\texttt{transfer}_2(D, u, \mathit{input\_p})$ by excluding tuples belonging to $\mathit{tuples}(\mathit{ans\_p})$.

We now define control strategies admissible w.r.t. strata's stability.

From now on, let P be a stratified Datalog$^\neg$ program and $P_1 \cup \ldots \cup P_K$ a stratification of P. The notions defined below are dependent on this fixed stratification.

Given a QSQ-STR-net (V, E, T, C) of P, we say that a node $v \in V$ *belongs to the layer* k, where $1 \leq k \leq K$, if v is constructed by some program clauses in P_k.[12] In that case, we say that the layer number of v is k, denoted by $\mathit{layer}(v) = k$.

A QSQ-STR-net of P is said to be *stable* up to a layer k if every edge (u, v) such that the layer numbers of u and v are less than or equal to k is not active.

Definition 24 (Admissibility w.r.t. Strata's Stability)**.** A control strategy for a given QSQ-STR-net of P (i.e., a strategy for selecting an edge to apply the procedure $\texttt{fire}'$ to it) is said to be *admissible w.r.t. strata's stability* if at the moment when an edge $(v, \mathit{succ}(v))$ with $v = \mathit{filter}_{i,j}$ is selected, if $\mathit{neg}(v) = \mathit{true}$, $\mathit{layer}(v) = k$, $\mathit{pred}(v) = p$ and p is an intensional predicate with $\mathit{layer}(\mathit{input\_p}) = h$, then the QSQ-STR-net is stable up to the layer h and the edge $(v, \mathit{input\_p})$ is not active.

By restricting to the case $\mathit{neg}(v) = \mathit{true}$, the condition of admissibility w.r.t. strata's stability in the above definition is weaker than the ones in [8, 9]. That is, we have extended the class of control strategies admissible w.r.t. strata's stability in comparison with the ones in [8, 9].

Finally, our QSQN-STR method for evaluating queries to stratified Datalog$^\neg$ databases is formulated by Algorithm 1. To ease the checking we have gathered

[12] That is, P_k contains a clause φ_i such that v is of the form *input_p*, $\mathit{pre\_filter}_i$, $\mathit{filter}_{i,j}$, $\mathit{post\_filter}_i$, or *ans_p*, where p is the predicate of A_i.

Algorithm 1: evaluating a query $q(\overline{x})$ to a stratified Datalog$^\neg$ database (P, I).

Input: a stratified Datalog$^\neg$ database (P, I), a stratification $P = P_1 \cup \ldots \cup P_K$ of P and a query $q(\overline{x})$.

Output: the set of all correct answers for the query $q(\overline{x})$ on (P, I).

let (V, E, T) be a QSQN-STR structure of P;
// T can be chosen arbitrarily or appropriately
set C so that (V, E, T, C) is an empty QSQ-STR-net of P;
let $\overline{x}'$ be a fresh variant of $\overline{x}$;
tuples(*input_q*) := $\{\overline{x}'\}$;
foreach $(input\_q, v) \in E$ **do** *unprocessed*$(input\_q, v) := \{\overline{x}'\}$;
while *there exists* $(u, v) \in E$ *such that* active-edge(u, v) *holds* **do**
    select any edge $(u, v) \in E$ such that active-edge(u, v) holds and the selection satisfies the admissibility w.r.t. strata's stability;
    fire$'(u, v)$;
return *tuples*(*ans_q*);

its full pseudocode into the online appendix [13]. An example illustrating how Algorithm 1 works step by step is also provided in [13]. A more friendly presentation of that example in the PowerPoint-like mode is also available online [12].

Observe that, if P is a Datalog program, then a run of Algorithm 1 coincides with an application of the QSQN evaluation method. That is, QSQN-STR coincides with QSQN when restricted to Datalog.

Theorem 1. The QSQN-STR method formulated by Algorithm 1 for evaluating queries to stratified Datalog$^\neg$ databases is sound, complete and has a PTIME data complexity.

The proof of this theorem is provided in the online appendix [13].

5 CONCLUSIONS

The previously known methods that can be used for evaluating queries to stratified Datalog$^\neg$ databases, except QSQN-WF [14], are either breadth-first [4, 37, 25, 30] or depth-first (and recursive) [26, 41, 37, 15, 40]. There are cases when these control strategies are not the best ones. QSQN-WF [14] is an evaluation framework for (general) Datalog$^\neg$ under the well-founded semantics and is not efficient when applied to stratified Datalog$^\neg$ because it would execute a considerable amount of redundant computation in order to guarantee that the alternative fixpoint has been reached.

In this paper, we have developed QSQN-STR as a novel evaluation framework for stratified Datalog$^\neg$. It is goal-driven, set-at-a-time and adjustable w.r.t. control

strategies. These properties are important for efficient query evaluation on large and complex deductive databases. Every control strategy admissible w.r.t. strata's stability can be used for QSQN-STR. The generic method QSQN-STR is sound, complete and has a PTIME data complexity for evaluating queries to stratified Datalog$^\neg$ databases.

QSQN-STR extends QSQN [11, 9] with the ability to handle stratified negation. Restricting to Datalog, QSQN is similar to QSQ [43, 1] but has some essential differences. First, it is formulated so that all operations can be precisely specified. Second, it does not use adornments for intensional predicates and their corresponding *input* and *answer* relations. This has some advantages:

- Operations of the same kind on an intensional predicate can be grouped and done together, independently from the adornments. This allows reducing the number of accesses to the secondary storage. The matter of how to efficiently execute the evaluation by using relational operations like join and projection is left for the implementation phase.
- *Input* relations contain tuples of terms possibly with variables, and information about repeats of variables in a goal is exploited. In the case of QSQ, *input* relations contain tuples of constant symbols, and only the annotated version of QSQ keeps and exploits information about repeats of variables in a goal.
- Only the most general tuples are kept in *input* relations. (Similarly, only the most general tuples and subqueries are kept in the other relations.) This allows reducing redundant computation. In the case of QSQ, tuples from different adorned *input* relations of the same intensional predicate are not compared to each other, and thus QSQ executes certain amount of redundant recomputation.

QSQN-STR inherits the aforementioned good properties of QSQN. As future work, further (conditional) optimizations can be incorporated into QSQN-STR. For example, we can extend QSQN-STR with tail-recursion elimination [38] in the way as QSQN-TRE [10, 9] extends QSQN.

Acknowledgements

We would like to thank the anonymous reviewers for many comments and suggestions, which have helped us to improve the paper significantly.

REFERENCES

[1] Abiteboul, S.—Hull, R.—Vianu, V.: Foundations of Databases. Addison Wesley, 1995.

[2] Apt, K. R.—Blair, H. A.—Walker, A.: Towards a Theory of Declarative Knowledge. Chapter 2. In: Minker, J. (Ed.): Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann, 1988, pp. 89–148, doi: 10.1016/C2013-0-07699-2. ISBN 978-0-934613-40-8.

[3] Apt, K. R.—Bol, R. N.: Logic Programming and Negation: A Survey. The Journal of Logic Programming, Vol. 19-20, Suppl. 1, 1994, pp. 9–71, doi: 10.1016/0743-1066(94)90024-8.

[4] Balbin, I.—Port, G. S.—Ramamohanarao, K.—Meenakshi, K.: Efficient Bottom-Up Computation of Queries on Stratified Databases. The Journal of Logic Programming, Vol. 11, 1991, No. 3-4, pp. 295–344, doi: 10.1016/0743-1066(91)90030-s.

[5] Bancilhon, F.—Maier, D.—Sagiv, Y.—Ullman, J. D.: Magic Sets and Other Strange Ways to Implement Logic Programs. Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS 1986), ACM, 1986, pp. 1–15, doi: 10.1145/6012.15399.

[6] Beeri, C.—Ramakrishnan, R.: On the Power of Magic. The Journal of Logic Programming, Vol. 10, 1991, No. 3-4, pp. 255–299, doi: 10.1016/0743-1066(91)90038-q.

[7] Calì, A.—Gottlob, G.—Lukasiewicz, T.: A General Datalog-Based Framework for Tractable Query Answering over Ontologies. Journal of Web Semantics, Vol. 14, 2012, pp. 57–83, doi: 10.1016/j.websem.2012.03.001.

[8] Cao, S. T.: Query-Subquery Nets with Stratified Negation. In: Le, T. H., Nguyen, N., Do, T. (Eds.): Advanced Computational Methods for Knowledge Engineering. Springer, Cham, Advances in Intelligent Systems and Computing, Vol. 358, 2015, pp. 355–366, doi: 10.1007/978-3-319-17996-4_32.

[9] Cao, S. T.: Methods for Evaluating Queries to Horn Knowledge Bases in First-Order Logic. Ph.D. thesis, University of Warsaw, 2016.

[10] Cao, S. T.—Nguyen, L. A.: An Empirical Approach to Query-Subquery Nets with Tail-Recursion Elimination. In: Bassiliades, N. et al. (Eds.): New Trends in Database and Information Systems II (ADBIS 2014). Springer, Cham, Advances in Intelligent Systems and Computing, Vol. 312, 2015, pp. 109–120, doi: 10.1007/978-3-319-10518-5_9.

[11] Cao, S. T.—Nguyen, L. A.: Query-Subquery Nets for Horn Knowledge Bases in First-Order Logic. Journal of Information and Telecommunication, Vol. 1, 2017, No. 1, pp. 79–99, doi: 10.1080/24751839.2017.1295664.

[12] Cao, S. T.—Nguyen, L. A.: Online Appendix: A Demonstration of the QSQN-STR Method for Evaluating Queries to Stratified Datalog$^{\neg}$. Available at: `http://mimuw.edu.pl/~nguyen/QSQN-STR-demonstration.pdf`, 2018.

[13] Cao, S. T.—Nguyen, L. A.: An Online Appendix for the Current Paper. Available at: `http://mimuw.edu.pl/~nguyen/QSQN-STR-appendix.pdf`, 2018.

[14] Cao, S. T.—Nguyen, L. A.—Nguyen, N. T.: Extending Query-Subquery Nets for Deductive Databases under the Well-Founded Semantics. Cybernetics and Systems, Vol. 48, 2017, No. 3, pp. 249–266, doi: 10.1080/01969722.2016.1276777.

[15] Chen, W.—Swift, T.—Warren, D. S.: Efficient Top-Down Computation of Queries under the Well-Founded Semantics. The Journal of Logic Programming, Vol. 24, 1995, No. 3, pp. 161–199, doi: 10.1016/0743-1066(94)00028-5.

[16] Dung, P. M.: On the Relations Between Stable and Well-Founded Semantics of Logic Programs. Theoretical Computer Science, Vol. 105, 1992, No. 1, pp. 7–25, doi: 10.1016/0304-3975(92)90285-n.

[17] Gebser, M.—Kaufmann, B.—Neumann, A.—Schaub, T.: *clasp*: A Conflict-Driven Answer Set Solver. In: Baral, C., Brewka, G., Schlipf, J. (Eds.): Logic Programming and Nonmonotonic Reasoning (LPNMR 2007). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 4483, 2007, pp. 260–265, doi: 10.1007/978-3-540-72200-7_23.

[18] Van Gelder, A.: The Alternating Fixpoint of Logic Programs with Negation. Journal of Computer and System Sciences, Vol. 47, 1993, No. 1, pp. 185–221, doi: 10.1016/0022-0000(93)90024-q.

[19] Van Gelder, A.—Ross, K. A.—Schlipf, J. S.: The Well-Founded Semantics for General Logic Programs. Journal of the ACM, Vol. 38, 1991, No. 3, pp. 619–649, doi: 10.1145/116825.116838.

[20] Gelfond, M.—Lifschitz, V.: The Stable Model Semantics for Logic Programming. Proceedings of International Logic Programming Conference and Symposium (ICLP/SLP 1988), MIT Press, 1988, pp. 1070–1080.

[21] Gire, F.: Equivalence of Well-Founded and Stable Semantics. The Journal of Logic Programming, Vol. 21, 1994, No. 2, pp. 95–111, doi: 10.1016/0743-1066(94)90002-7.

[22] Greco, S.—Molinaro, C.—Trubitsyna, I.—Zumpano, E.: NP Datalog: A Logic Language for Expressing Search and Optimization Problems. Theory and Practice of Logic Programming, Vol. 10, 2010, No. 2, pp. 125–166, doi: 10.1017/S1471068409990251.

[23] Green, T. J.—Huang, S. S.—Loo, B. T.—Zhou, W.: Datalog and Recursive Query Processing. Foundations and Trends in Databases, Vol. 5, 2013, No. 2, pp. 105–195, doi: 10.1561/1900000017.

[24] Huang, S. S.—Green, T. J.—Loo, B. T.: Datalog and Emerging Applications: An Interactive Tutorial. Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD 2011), ACM, 2011, pp. 1213–1216, doi: 10.1145/1989323.1989456.

[25] Kemp, D. B.—Srivastava, D.—Stuckey, P. J.: Bottom-Up Evaluation and Query Optimization of Well-Founded Models. Theoretical Computer Science, Vol. 146, 1995, Nos. 1–2, pp. 145–184, doi: 10.1016/0304-3975(94)00153-a.

[26] Kemp, D. B.—Topor, R. W.: Completeness of a Top-Down Query Evaluation Procedure for Stratified Databases. Proceedings of International Logic Programming Conference and Symposium (ICLP/SLP 1988), MIT Press, 1988, pp. 178–194.

[27] Leone, N.—Pfeifer, G.—Faber, W.—Eiter, T.—Gottlob, G.—Perri, S.—Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic, Vol. 7, 2006, No. 3, pp. 499–562, doi: 10.1145/1149114.1149117.

[28] Lloyd, J. W.: Foundations of Logic Programming. 2nd edition. Springer, 1987, doi: 10.1007/978-3-642-83189-8.

[29] Madalińska-Bugaj, E.—Nguyen, L. A.: A Generalized QSQR Evaluation Method for Horn Knowledge Bases. ACM Transactions on Computational Logic, Vol. 13, 2012, No. 4, Art. No. 32, doi: 10.1145/2362355.2362360.

[30] Morishita, S.: An Extension of Van Gelder's Alternating Fixpoint to Magic Programs. Journal of Computer and System Sciences, Vol. 52, 1996, No. 3, pp. 506–521, doi: 10.1006/jcss.1996.0038.

[31] Nejdl, W.: Recursive Strategies for Answering Recursive Queries – the RQA/FQI Strategy. Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '87), Morgan Kaufmann, 1987, pp. 43–50, doi: 10.1007/978-3-642-46620-5_3.

[32] Nguyen, L. A.—Cao, S. T.: Query-Subquery Nets. CoRR, abs/1201.2564, 2012.

[33] Nilsson, U.—Maluszynski, J.: Logic, Programming and Prolog. 2nd edition. John Wiley and Sons, Inc., 1995. ISBN: 978-0471959960.

[34] Przymusinski, T. C.: Every Logic Program Has a Natural Stratification and an Iterated Least Fixed Point Model. Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 1989), ACM Press, 1989, pp. 11–21, doi: 10.1145/73721.73723.

[35] Przymusinski, T. C.: The Well-Founded Semantics Coincides with the Three-Valued Stable Semantics. Fundamenta Informaticae, Vol. 13, 1990, No. 4, pp. 445–463.

[36] Ramamohanarao, K.—Harland, J.: An Introduction to Deductive Database Languages and Systems. The VLDB Journal, Vol. 3, 1994, No. 2, pp. 107–122, doi: 10.1007/bf01228878.

[37] Ross, K. A.: Modular Stratification and Magic Sets for Datalog Programs with Negation. Journal of the ACM, Vol. 41, 1994, No. 6, pp. 1216–1266, doi: 10.1145/195613.195646.

[38] Ross, K. A.: Tail Recursion Elimination in Deductive Databases. ACM Transactions on Database Systems, Vol. 21, 1996, No. 2, pp. 208–237, doi: 10.1145/232616.232628.

[39] Sáenz-Pérez, F.: DES: A Deductive Database System. Electronic Notes in Theoretical Computer Science, Vol. 271, 2011, pp. 63–78, doi: 10.1016/j.entcs.2011.02.011.

[40] Sagonas, K. F.—Swift, T.—Warren, D. S.: XSB as an Efficient Deductive Database Engine. Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94), ACM Press, 1994, pp. 442–453.

[41] Seki, H.—Itoh, H.: A Query Evaluation Method for Stratified Programs under the Extended CWA. Proceedings of International Logic Programming Conference and Symposium (ICLP/SLP 1988), MIT Press, 1988, pp. 195–211.

[42] Tamaki, H.—Sato, T.: OLD Resolution with Tabulation. In: Shapiro, E. (Ed.): Third International Conference on Logic Programming (ICLP 1986). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 225, 1986, pp. 84–98, doi: 10.1007/3-540-16492-8_66.

[43] Vieille, L.: Recursive Axioms in Deductive Databases: The Query/Subquery Approach. Proceedings of the First International Conference on Expert Database Systems, 1986, pp. 179–194.

[44] Vieille, L.: Recursive Query Processing: The Power of Logic. Theoretical Computer Science, Vol. 69, 1989, No. 1, pp. 1–53, doi: 10.1016/0304-3975(89)90088-1.

Son Thanh Cao received his Ph.D. degree in computer science from the University of Warsaw in 2016. He is currently Lecturer at Vinh University in Vietnam. His research interests include logic programming, deductive database, description logic, semantic Web and artificial intelligence.

Linh Anh Nguyen received the degrees of Ph.D. and Dr.Sc. (Habilitation) in computer science from the University of Warsaw in 2000 and 2009, respectively. He is currently Associate Professor with the Institute of Informatics, University of Warsaw in Poland. Since 2014 he has been cooperating with the Faculty of Information Technology, Ton Duc Thang University in Vietnam. His research interests include modal and description logics, automated reasoning, rule-based languages, deductive databases, and concept learning.