An Empirical Approach to Query-Subquery Nets with Tail-Recursion Elimination

Son Thanh Cao[1,2] and Linh Anh Nguyen[2,3]

[1] Faculty of Information Technology, Vinh University
182 Le Duan Street, Vinh, Nghe An, Vietnam
sonct@vinhuni.edu.vn
[2] Institute of Informatics, University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
nguyen@mimuw.edu.pl
[3] Faculty of Information Technology
VNU University of Engineering and Technology
144 Xuan Thuy, Hanoi, Vietnam

Abstract. We propose a method called QSQN-TRE for evaluating queries to Horn knowledge bases by integrating Query-Subquery Nets with a form of tail-recursion elimination. The aim is to improve the QSQN method by avoiding materialization of intermediate results during the processing. We illustrate the efficiency of our method by empirical results, especially for tail-recursive cases.

1 Introduction

Query optimization has received much attention from researchers in the database community. Several optimization methods and techniques have been developed to improve performance of query evaluation. One of them is to reduce the number of materialized intermediate results during the processing by using the tail-recursion elimination. The general form of recursion requires the compiler to allocate storage on the stack at run-time. Such a memory consumption may be very costly. A call is tail-recursive if no work remains to be done after the call returns. Tail recursion is a special case of recursion that is semantically equivalent to the iteration construct, so a tail-recursive program can be compiled as efficiently as iterative programs.

This work studies optimizing query evaluation for Horn knowledge bases.

1.1 Related Work

Horn knowledge bases are a generalization of Datalog deductive databases as they allow function symbols and do not require the range-restrictedness condition. Researchers have developed a number of evaluation methods for Datalog deductive databases such as the top-down methods QSQ [15,1], QSQR [15,1,9], QoSaQ [16], QSQN [10] and the bottom-up method Magic-Set [1,2]. By Magic-Set we mean the evaluation method that combines the

N. Bassiliades et al. (eds.), *New Trends in Database and Information Systems II*,
Advances in Intelligent Systems and Computing 312, DOI: 10.1007/978-3-319-10518-5_9

magic-set transformation with the improved semi-naive evaluation method. In [9], Madalińska-Bugaj and Nguyen generalized the QSQR method for Horn knowledge bases. Some authors also extended the magic-set technique together with the breadth-first approach for Horn knowledge bases [12,8]. One can also try to adapt computational procedures of logic programming that use tabled SLD-resolution [14,16,17] for evaluating queries to Horn knowledge bases.

In [10], we formulated the Query-Subquery Nets (QSQN) framework for evaluating queries to Horn knowledge bases. The aim was to increase efficiency of query processing by eliminating redundant computation, increasing flexibility and reducing the number of accesses to the secondary storage. The preliminary comparison between QSQN, QSQR and Magic-Set reported in [3] justifies the usefulness of QSQN.

In [13], Ross proposed an optimization technique that integrates Magic-Set with a form of tail-recursion elimination. It improves the performance by not representing intermediate results, as can be seen in the following example.

Example 1. This example shows the inefficiency of a logic program without a tail-recursive evaluation. It is a modified version of Example 1.1 from [13]. Consider the following positive logic program P:

$$p(x, y) \leftarrow e(x, z), p(z, y)$$
$$p(n, x) \leftarrow t(x).$$

where p is an *intensional* predicate, e and t are *extensional* predicates, n is a natural number (constant) and x, y, z are variables. Let $p(1, x)$ be the query, m a natural number, and let the *extensional* instance I for e and t be as follows:

$$I(t) = \{(1), (2), \ldots, (m-1), (m)\},$$
$$I(e) = \{(1, 2), (2, 3), \ldots, (n-1, n), (n, 1)\}.$$

In order to answer the query, a method such as QSQR, QSQN, Magic-Set would evaluate every tuple of the form $p(i, j)$, where $1 \leq i \leq n$ and $1 \leq j \leq m$. Thus, it stores a set of $n \times m$ tuples, but many of them are not closely related to the query in question. As we shall see, for answering the query $p(1, x)$, we do not need to evaluate $p(i, j)$ for $i > 1$ if we apply a tail-recursive evaluation. We only need to evaluate $p(1, j)$ for $1 \leq j \leq m$ and additional tuples for some newly introduced relations. Thus, the total number of evaluated tuples is smaller than that of the standard approach. ◁

1.2 Our Contributions

In this paper, we propose a method called QSQN-TRE for evaluating queries to Horn knowledge bases. The method integrates the QSQN method from [10] with a form of tail-recursion elimination.

Our method has many advantages: it reduces the number of evaluated intermediate tuples/subqueries during processing, increases flexibility and eliminates redundant computation. Furthermore, since unnecessary intermediate results are

not stored, it usually performs better than the QSQN method for the case as in Example 1. To deal with function symbols, we use a term-depth bound for atoms and substitutions occurring in the computation and propose to use iterative deepening search which iteratively increases the term-depth bound. Similarly to the QSQN method, our new method allows various control strategies such as Depth-First Search (DFS), Improved Depth-First Search (IDFS) and Disk Access Reduction (DAR), which have been proposed in [10,3,5].

2 Preliminaries

We assume that the reader is familiar with the notions of *term, atom, predicate, substitution, unification, mgu (most general unifier)* and related ones. We refer the reader to [1,11] for further reading.

We classify each predicate either as *intensional* or as *extensional*. A *generalized tuple* is a tuple of terms, which may contain function symbols and variables. A *generalized relation* is a set of generalized tuples of the same arity.

A *program clause* is a formula of the form $(A \vee \neg B_1 \vee \ldots \vee \neg B_n)$ with $n \geq 0$, written as $A \leftarrow B_1, \ldots, B_n$, where $A, B_1, \ldots, B_n$ are atoms. A is called the *head*, and $B_1, \ldots, B_n$ the *body* of the program clause. If p is the predicate of A then the program clause is called a program clause defining p.

A *positive* (or *definite*) *logic program* is a finite set of program clauses. From now on, by a "program" we will mean a positive logic program.

A *goal* is a formula of the form $(\neg B_1 \vee \ldots \vee \neg B_n)$, written as $\leftarrow B_1, \ldots, B_n$, where $B_1, \ldots, B_n$ are atoms and $n \geq 0$. If $n = 1$ then the goal is called a *unary goal*. If $n = 0$ then the goal is referred to as the *empty goal*.

Given substitutions θ and δ, the composition of θ and δ is denoted by $\theta\delta$, the domain of θ is denoted by $dom(\theta)$, the range of θ is denoted by $range(\theta)$, and the restriction of θ to a set X of variables is denoted by $\theta_{|X}$. The *term-depth* of an expression (resp. a substitution) is the maximal nesting depth of function symbols occurring in that expression (resp. substitution). Given a list/tuple α of terms or atoms, by $Vars(\alpha)$ we denote the set of variables occurring in α.

A *Horn knowledge base* is defined as a pair that consists of a positive logic program P (which may contain function symbols and not be "range-restricted") for defining *intensional* predicates and a *generalized extensional instance* I, which is a function mapping each extensional n-ary predicate to an n-ary generalized relation.

A *query* to a Horn knowledge base (P, I) is a formula $q(\overline{x})$, where q is an n-ary *intensional* predicate and $\overline{x}$ is a tuple of n pairwise different variables. An *answer* to the query is an n-ary tuple $\overline{t}$ of terms such that $P \cup I \models q(\overline{t})$, treating I as the corresponding set of atoms of the extensional predicates.

Definition 1 (Tail-recursion). A program clause $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, for $n_i > 0$, is said to be *recursive* whenever some $B_{i,j}$ $(1 \leq j \leq n_i)$ has the same predicate as A_i. If B_{i,n_i} has the same predicate as A_i then the clause is *tail-recursive*. ◁

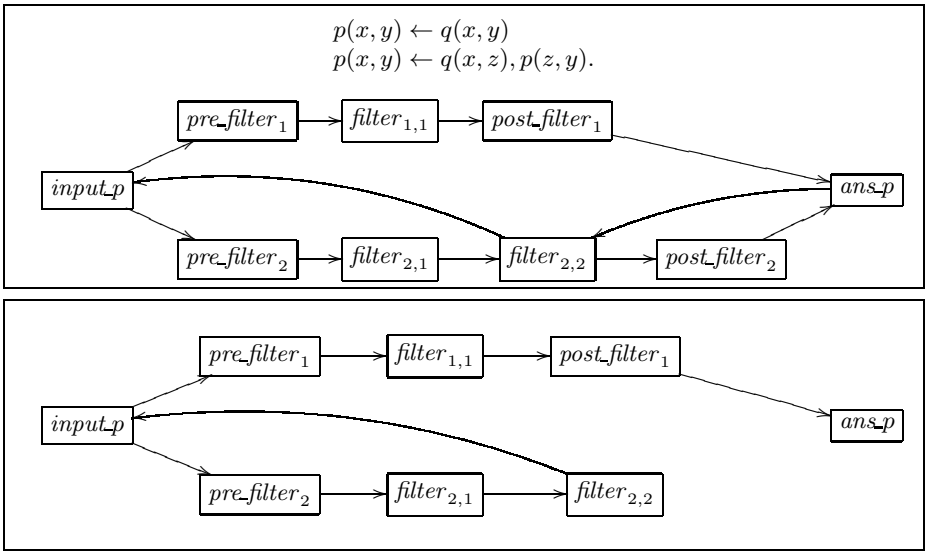

Fig. 1. The QSQ topological structure and the QSQN-TRE topological structure of the program given in Example 2

3 QSQ-Nets with Tail-Recursion Elimination

In this section we specify the notion QSQN-TRE (QSQ-net with tail-recursion elimination) and describe our QSQN-TRE evaluation method for Horn knowledge bases. QSQN-TRE is an extension of QSQN introduced in [11,10] and can be viewed as a flow control network for determining which set of tuples/subqueries should be processed next.

Example 2. This example is taken from [10]. The upper part of Figure 1 illustrates a logic program and its QSQ topological structure, where p is an *intensional* predicate, q is an *extensional* predicate and x, y, z are variables. ◁

In what follows, P is a positive logic program and all $\varphi_1, \ldots, \varphi_m$ are the program clauses of P, with $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, for $1 \leq i \leq m$ and $n_i \geq 0$. The following definition shows how to make a QSQN-TRE structure from the given positive logic program P.

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

- T is a pair (T_{edb}, T_{idb}), called the *type* of the net structure.
- T_{idb} is a function that maps each *intensional* predicate to *true* or *false*. (If $T_{idb}(p) = true$ then the *intensional* relation p will be computed using tail-recursion elimination).
- V is a set of nodes that includes:
  - *input_p* and *ans_p*, for each *intensional* predicate p of P,

  - pre_filter_i, $filter_{i,1}$, ..., $filter_{i,n_i}$, for each $1 \leq i \leq m$,
  - $post\_filter_i$ if either φ_i is not tail-recursive or $T_{idb}(p) = false$, for each $1 \leq i \leq m$, where p is the predicate of A_i.
- E is a set of edges that includes:
  - $(input\_p, pre\_filter_i)$, for each $1 \leq i \leq m$, where p is the predicate of A_i,
  - $(pre\_filter_i, filter_{i,1})$, for each $1 \leq i \leq m$ such that $n_i \geq 1$,
  - $(filter_{i,1}, filter_{i,2})$, ..., $(filter_{i,n_i-1}, filter_{i,n_i})$, for each $1 \leq i \leq m$,
  - $(filter_{i,n_i}, post\_filter_i)$, for each $1 \leq i \leq m$ such that $n_i \geq 1$ and either φ_i is not tail-recursive or $T_{idb}(p) = false$, where p is the predicate of A_i,
  - $(pre\_filter_i, post\_filter_i)$, for each $1 \leq i \leq m$ such that $n_i = 0$,
  - $(post\_filter_i, ans\_p)$, for each $1 \leq i \leq m$ such that either φ_i is not tail-recursive or $T_{idb}(p) = false$, where p is the predicate of A_i,
  - $(filter_{i,j}, input\_p)$, for all $1 \leq i \leq m$ and $1 \leq j \leq n_i$ such that the predicate p of $B_{i,j}$ is an *intensional* predicate,
  - $(ans\_p, filter_{i,j})$, for each *intensional* predicate p and for all $1 \leq i \leq m$ and $1 \leq j \leq n_i$ such that $B_{i,j}$ is an atom of p and either φ_i is not tail-recursive or $T_{idb}(p) = false$.
- T_{edb} is a function that maps each $filter_{i,j} \in V$ such that the predicate of $B_{i,j}$ is *extensional* to *true* or *false*. (If $T_{edb}(filter_{i,j}) = false$ then subqueries for $filter_{i,j}$ are always processed immediately without being accumulated at $filter_{i,j}$). ◁

From now on, $T(v)$ denotes $T_{edb}(v)$ if v is a node $filter_{i,j}$ such that $B_{i,j}$ is an *extensional* predicate, and $T(p)$ denotes $T_{idb}(p)$ for an *intensional* predicate p. Thus, T can be called a *memorizing type* for *extensional* nodes (as in QSQ-net structures), and a *tail-recursion-elimination type* for *intensional* predicates.

We call the pair (V, E) the QSQN-TRE topological structure of P w.r.t. T_{idb}. The lower part of Figure 1 illustrates the QSQN-TRE topological structure of the positive logic program given in Example 2 w.r.t. the T_{idb} with $T_{idb}(p) = true$.

We now specify the notion QSQN-TRE and the related ones. We also show how the data is transferred through the edges in QSQN-TRE.

Definition 3 (QSQN-TRE). A *query-subquery net with tail-recursion elimination* (QSQN-TRE for short) of P is a tuple $N = (V, E, T, C)$ such that (V, E, T) is a QSQN-TRE 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:

- If either ($v = input\_p$ and $T(p) = false$) or $v = ans\_p$ then $C(v)$ consists of:
  - $tuples(v)$: a set of generalized tuples of the same arity as p,
  - $unprocessed(v, w)$ for $(v, w) \in E$: a subset of $tuples(v)$.
- If $v = input\_p$ and $T(p) = true$ then $C(v)$ consists of:
  - $tuple\_pairs(v)$: a set of pairs of generalized tuples of the same arity as p,
  - $unprocessed(v, w)$ for $(v, w) \in E$: a subset of $tuple\_pairs(v)$.
- If $v = pre\_filter_i$ then $C(v)$ consists of:
  - $atom(v) = A_i$ and $post\_vars(v) = Vars((B_{i,1}, \ldots, B_{i,n_i}))$.
- If $v = post\_filter_i$ then $C(v)$ is empty, but we assume $pre\_vars(v) = \emptyset$.

- If $v = \mathit{filter}_{i,j}$ and p is the predicate of $B_{i,j}$ then $C(v)$ consists of:
  - $\mathit{kind}(v) = \mathit{extensional}$ if p is extensional, and $\mathit{kind}(v) = \mathit{intensional}$ otherwise,
  - $\mathit{pred}(v) = p$ (called the predicate of v) and $\mathit{atom}(v) = B_{i,j}$,
  - $\mathit{pre\_vars}(v) = \mathit{Vars}((B_{i,j}, \ldots, B_{i,n_i}))$ and $\mathit{post\_vars}(v) = \mathit{Vars}((B_{i,j+1}, \ldots, B_{i,n_i}))$,
  - $\mathit{subqueries}(v)$: a set of pairs of the form $(\overline{t}, \delta)$, where $\overline{t}$ is a generalized tuple of the same arity as the predicate of A_i and δ is an idempotent substitution such that $\mathit{dom}(\delta) \subseteq \mathit{pre\_vars}(v)$ and $\mathit{dom}(\delta) \cap \mathit{Vars}(\overline{t}) = \emptyset$,
  - $\mathit{unprocessed\_subqueries}(v) \subseteq \mathit{subqueries}(v)$,
  - in the case p is *intensional*: $\mathit{unprocessed\_subqueries}_2(v) \subseteq \mathit{subqueries}(v)$, $\mathit{unprocessed\_tuples}(v)$: a set of generalized tuples of the same arity as p;
  - if $v = \mathit{filter}_{i,n_i}$, $\mathit{kind}(v) = \mathit{intensional}$, $\mathit{pred}(v) = p$ and $T(p) = \mathit{true}$ then $\mathit{unprocessed\_subqueries}(v)$ and $\mathit{unprocessed\_tuples}(v)$ are empty (and can thus be ignored).
- If $v = \mathit{filter}_{i,j}$, $\mathit{kind}(v) = \mathit{extensional}$ and $T(v) = \mathit{false}$ then $\mathit{subqueries}(v)$ and $\mathit{unprocessed\_subqueries}(v)$ are empty (and can thus be ignored).

A QSQN-TRE of P is *empty* if all the sets of the form $\mathit{tuple\_pairs}(v)$, $\mathit{tuples}(v)$, $\mathit{unprocessed}(v, w)$, $\mathit{subqueries}(v)$, $\mathit{unprocessed\_subqueries}(v)$, $\mathit{unprocessed\_subqueries}_2(v)$ or $\mathit{unprocessed\_tuples}(v)$ are empty. ◁

If $(v, w) \in E$ then w is referred to as a *successor* of v. Observe that:

- if $v \in \{\mathit{pre\_filter}_i, \mathit{post\_filter}_i\}$ or $v = \mathit{filter}_{i,j}$ and $\mathit{kind}(v) = \mathit{extensional}$ then v has exactly one successor, which we denote by $\mathit{succ}(v)$;
- if v is filter_{i,n_i} with $\mathit{kind}(v) = \mathit{intensional}$, $\mathit{pred}(v) = p$ and $T(p) = \mathit{true}$ then v has exactly one successor, which we denote by $\mathit{succ}_2(v) = \mathit{input\_p}$;
- if v is $\mathit{filter}_{i,j}$ with $\mathit{kind}(v) = \mathit{intensional}$, $\mathit{pred}(v) = p$ and either $j < n_i$ or $T(p) = \mathit{false}$ then v has exactly two successors: $\mathit{succ}(v) = \mathit{filter}_{i,j+1}$ if $n_i > j$; $\mathit{succ}(v) = \mathit{post\_filter}_i$ otherwise; and $\mathit{succ}_2(v) = \mathit{input\_p}$.

By a *subquery* we mean a pair of the form $(\overline{t}, \delta)$, where $\overline{t}$ is a generalized tuple and δ is an idempotent substitution such that $\mathit{dom}(\delta) \cap \mathit{Vars}(\overline{t}) = \emptyset$. The set $\mathit{unprocessed\_subqueries}_2(v)$ (resp. $\mathit{unprocessed\_subqueries}(v)$) contains the subqueries that were not transferred through the edge $(v, \mathit{succ}_2(v))$ (resp. $(v, \mathit{succ}(v))$ – when it exists).

For an *intensional* predicate p with $T(p) = \mathit{true}$, the intuition behind a pair $(\overline{t}, \overline{t}') \in \mathit{tuple\_pairs}(\mathit{input\_p})$ is that:

- $\overline{t}$ is a usual input tuple for p, but the intended goal at a higher level is $\leftarrow p(\overline{t}')$,
- any correct answer for $P \cup I \cup \{\leftarrow p(\overline{t})\}$ is also a correct answer for $P \cup I \cup \{\leftarrow p(\overline{t}')\}$,
- if a substitution θ is a computed answer of $P \cup I \cup \{\leftarrow p(\overline{t})\}$ then we will store in $\mathit{ans\_p}$ the tuple $\overline{t}'\theta$ instead of $\overline{t}\theta$.

We say that a tuple pair $(\overline{t}, \overline{t}')$ is *more general* than $(\overline{t}_2, \overline{t}'_2)$, and $(\overline{t}_2, \overline{t}'_2)$ is an *instance* of $(\overline{t}, \overline{t}')$, if there exists a substitution θ such that $(\overline{t}, \overline{t}')\theta = (\overline{t}_2, \overline{t}'_2)$.

For $v = \mathit{filter}_{i,j}$ and p being the predicate of A_i, the meaning of a subquery $(\overline{t}, \delta) \in \mathit{subqueries}(v)$ is as follows: if $T(p) = \mathit{false}$ (resp. $T(p) = \mathit{true}$) then there exists $\overline{s} \in \mathit{tuples}(\mathit{input\_p})$ (resp. $(\overline{s}, \overline{s}') \in \mathit{tuple\_pairs}(\mathit{input\_p})$) such that for processing the goal $\leftarrow p(\overline{s})$ using the program clause $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, unification of $p(\overline{s})$ and A_i as well as processing of the subgoals $B_{i,1}, \ldots, B_{i,j-1}$ were done, amongst others, by using a sequence of mgu's $\gamma_0, \ldots, \gamma_{j-1}$ with the property that $\overline{t} = \overline{s}\gamma_0 \ldots \gamma_{j-1}$ (resp. $\overline{t} = \overline{s}'\gamma_0 \ldots \gamma_{j-1}$) and $\delta = (\gamma_0 \ldots \gamma_{j-1})_{|\mathit{Vars}((B_{i,j}, \ldots, B_{i,n_i}))}$. Informally, a subquery $(\overline{t}, \delta)$ transferred through an edge to v is processed as follows:

- if $v = \mathit{filter}_{i,j}$, $\mathit{kind}(v) = \mathit{extensional}$ and $\mathit{pred}(v) = p$ then, for each $\overline{t}' \in I(p)$, if $\mathit{atom}(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\overline{t}')$ by an mgu γ then transfer the subquery $(\overline{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(v)})$ through $(v, \mathit{succ}(v))$.
- if $v = \mathit{filter}_{i,j}$, $\mathit{kind}(v) = \mathit{intensional}$, $\mathit{pred}(v) = p$ and either $T(p) = \mathit{false}$ or ($T(p) = \mathit{true}$ and (either $j < n_i$ or p is not the predicate of A_i)) then
  - if $T(p) = \mathit{false}$ then transfer the input tuple $\overline{t}'$ such that $p(\overline{t}') = \mathit{atom}(v)\delta = B_{i,j}\delta$ through $(v, \mathit{input\_p})$ to add a fresh variant of it to $\mathit{tuples}(\mathit{input\_p})$,
  - if $T(p) = \mathit{true}$ and either $j < n_i$ or p is not the predicate of A_i then transfer the input tuple pair $(\overline{t}', \overline{t}')$ such that $p(\overline{t}') = \mathit{atom}(v)\delta = B_{i,j}\delta$ through $(v, \mathit{input\_p})$ to add a fresh variant of it to $\mathit{tuple\_pairs}(\mathit{input\_p})$,
  - for each currently existing $\overline{t}' \in \mathit{tuples}(\mathit{ans\_p})$, if $\mathit{atom}(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\overline{t}')$ by an mgu γ then transfer the subquery $(\overline{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(v)})$ through $(v, \mathit{succ}(v))$,
  - store the subquery $(\overline{t}, \delta)$ in $\mathit{subqueries}(v)$, and later, for each new $\overline{t}'$ added to $\mathit{tuples}(\mathit{ans\_p})$, if $\mathit{atom}(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\overline{t}')$ by an mgu γ then transfer the subquery $(\overline{t}\gamma, (\delta\gamma)_{|\mathit{post\_vars}(v)})$ through $(v, \mathit{succ}(v))$.
- if $v = \mathit{filter}_{i,n_i}$, $\mathit{kind}(v) = \mathit{intensional}$, $\mathit{pred}(v) = p$, $T(p) = \mathit{true}$ and p is the predicate of A_i then transfer the input tuple pair $(\overline{t}', \overline{t})$ such that $p(\overline{t}') = \mathit{atom}(v)\delta = B_{i,n_i}\delta$ through $(v, \mathit{input\_p})$ to add a fresh variant of it to $\mathit{tuple\_pairs}(\mathit{input\_p})$.
- if $v = \mathit{post\_filter}_i$ and p is the predicate of A_i then transfer the answer tuple $\overline{t}$ through $(\mathit{post\_filter}_i, \mathit{ans\_p})$ to add it to $\mathit{tuples}(\mathit{ans\_p})$.

Formally, the processing of a subquery, an input/answer tuple or an input tuple pair in a QSQN-TRE is designed so that:

- every subquery or input/answer tuple or input tuple pair that is subsumed by another one or has a term-depth greater than a fixed bound l is ignored;
- the processing is divided into smaller steps which can be delayed at each node to maximize flexibility and allow various control strategies;
- the processing is done set-at-a-time (e.g., for all the unprocessed subqueries accumulated in a given node).

The procedure `transfer2`(D, u, v) given in [11] specifies the effects of transferring data D through an edge (u, v) of a QSQN-TRE. If v is of the form pre_filter_i or $post\_filter_i$ or ($v = filter_{i,j}$ and $kind(v) = extensional$ and $T(v) = false$) then the input D for v is processed immediately and an appropriate data Γ is produced and transferred through $(v, succ(v))$. Otherwise, the input D for v is not processed immediately, but accumulated into the structure of v in an appropriate way. The function `active-edge`(u, v) given in [11] returns *true* for an edge (u, v) if the data accumulated in u can be processed to produce some data to transfer through (u, v), and returns *false* otherwise. If `active-edge`(u, v) is *true*, the procedure `fire2`(u, v) given in [11][1] processes the data accumulated in u that has not been processed before to transfer appropriate data through the edge (u, v). Both the procedures `fire2`(u, v) and `transfer2`(D, u, v) use a parameter l as a term-depth bound for tuples and substitutions.

Algorithm 2 of [11] presents our QSQN-TRE evaluation method for Horn knowledge bases. It repeatedly selects an active edge and fires the operation for the edge. Such a selection is decided by the adopted control strategy, which can be arbitrary. If there is no tail-recursion to eliminate or $T(p) = false$ for every *intensional* predicate p, the QSQN-TRE method reduces to the QSQN evaluation method. See [11] for properties on soundness, completeness and data complexity of the QSQN-TRE method.

4 Preliminary Experiments

This section presents our experimental results and a discussion about the performance of the QSQN-TRE evaluation method in comparison with the QSQN method using the IDFS control strategy [5]. All the experiments have been performed using our Java codes [4] and *extensional* relations stored in a MySQL database. The package [4] also contains all the experimental results reported below. In the following tests, we use typical examples that appear in many well-known articles related to deductive databases, including tail/non-tail recursive logic programs as well as logic programs with or without function symbols. Our implementation allows queries of the form $q(\overline{t})$, where $\overline{t}$ is a tuple of terms.

4.1 The Settings

The experiments are divided into two stages. All the experimental results reported below are for the first stage.

1. In the first stage, we assume that the computer memory is large enough to load all the involved *extensional* relations and keep all the intermediate relations. During execution of the program, for each operation of reading from a relation (resp. writing a set of tuples to a relation), we increase the counter of read (resp. write) operations on this relation by one. For counting the maximum number of kept tuples/subqueries in the memory, we increase (resp. decrease)

[1] The step 3 in the macro `compute-gamma` for the procedure `fire2` in [11] should be replaced by "**else if** $j < n_i$ or p is not the predicate of A_i **then**".

Table 1. A comparison between the QSQN and QSQN-TRE methods w.r.t. the number of read/write operations. The "Reading *inp_/ans_/sup_/edb*" column means the number of read operations from *input/answer/supplement/extensional* relations, respectively. Similarly, the "Writing *inp_/ans_/sup_*" column means the number of write operations to *input/answer/supplement* relations, respectively. The last column shows the maximum number of kept tuples in the memory for each test.

Tests	**Methods**	**Reading (times)** *inp_/ans_/sup_/edb*	**Writing (times)** *inp_/ans_/sup_*	**Max No. of kept tuples**
Test 1 (a)	QSQN	156 (40+38+57+21)	58 (20+19+19)	248
	QSQN-TRE	100 (40+1+38+21)	40 (20+1+19)	97
Test 1 (b)	QSQN	64 (3+38+21+2)	21 (1+19+1)	229
	QSQN-TRE	100 (40+1+38+21)	40 (20+1+19)	781
Test 2 (a)	QSQN	190 (41+59+69+21)	69 (20+29+20)	992
	QSQN-TRE	101 (40+1+39+21)	40 (20+1+19)	151
Test 2 (b)	QSQN	95 (3+59+31+2)	31 (1+29+1)	963
	QSQN-TRE	151 (60+1+59+31)	60 (30+1+29)	3573
Test 3	QSQN	43 (5+21+16+1)	13 (1+9+3)	101
	QSQN-TRE	58 (19+15+19+5)	19 (5+5+9)	237
Test 4	QSQN	56 (15+14+20+7)	20 (7+6+7)	136
	QSQN-TRE	39 (14+4+14+7)	15 (7+2+6)	93
Test 5	QSQN	403 (101+101+150+51)	150 (50+50+50)	10,350
	QSQN-TRE	253 (101+1+100+51)	101 (50+1+50)	600
Test 6	QSQN	184 (48+46+66+24)	67 (23+22+22)	930
	QSQN-TRE	126 (48+8+46+24)	49 (23+4+22)	566
Test 7	QSQN	91 (7+39+25+20)	25 (3+19+3)	195
	QSQN-TRE			

the counter of kept tuples by two if a tuple pair is added to (resp. removed from) *tuple_pairs*(*input_p*), otherwise we increase (resp. decrease) it by one. The returned value is the maximum value of this counter.

2. The second stage follows the first one. We will limit the space available in computer memory for storing the tuples/subqueries on each test. This will require load and unload operations on disk when the computer memory is not enough to hold all the relations. The aim of this stage is to estimate the number of disk accesses. This stage is still in progress.

4.2 Experimental Results

We compare the QSQN-TRE and QSQN methods with respect to the number of accesses to the intermediate relations and *extensional* relations as well as the number of kept tuples/subqueries in the memory for the following tests.

Test 1. Reconsider the logic program from Example 2, where p is an *intensional* predicate, q is an *extensional* predicate, and x, y, z are variables. Let the *extensional* instance I for q be as follows: $I(q) = \{(a_i, a_{i+1}) \mid 1 \leq i < n\}$, where a_i are constant symbols and n is a natural number.

$$p(x, y) \leftarrow q(x, y)$$
$$p(x, y) \leftarrow q(x, z), p(z, y).$$

We perform this test using the following queries: (a) $p(a_1, x)$, (b) $p(x, y)$.

Similar to the discussion in Example 1, in order to answer a query as in the part (a) or (b), QSQN has to evaluate all tuples of the form (a_i, a_j), where $1 \leq i < j \leq n$. However, for the query $p(a_1, x)$ as in the part (a), by applying tail-recursion elimination, QSQN-TRE only needs to evaluate a set of tuples of the form (a_1, a_j) with $1 < j \leq n$. Thus, in this case, the number of evaluated tuples for QSQN-TRE is much smaller than QSQN. In contrast, for queries without any bound parameter such as $p(x, y)$ as in the part (b), QSQN-TRE has to evaluate also all the related tuples and may be worse than QSQN. The reason is that, after the processing at node $v = \mathit{filter}_{i,n_i}$ with $p = \mathit{pred}(v)$, if $T(p) = \mathit{true}$, QSQN-TRE produces a set of tuple pairs and accumulates them in $\mathit{tuple\_pairs}(\mathit{input\_p})$, which are not instances of each other. Meanwhile, QSQN adds the answers to $\mathit{tuples}(\mathit{ans\_p})$ for later processing, and also transfers data through $(v, \mathit{input\_p})$ without adding any new tuple to $\mathit{tuples}(\mathit{input\_p})$ because it already contains a fresh variant of (x, y) that is more general than all the other tuples. As the result, in this case, QSQN-TRE may keep more tuples than QSQN. We use $n = 20$ for this test.

Test 2. This test uses the logic program P and the queries as in Test 1, but the *extensional* instance I for q is extended to contain cycles as follows, where a_i and b_i are constant symbols:

$$I(q) = \{(a_i, a_{i+1}) \mid (1 \leq i < 20)\} \cup \{(a_{20}, a_1)\} \cup \{(a_1, b_1)\} \cup \{(b_i, b_{i+1}) \mid 1 \leq i < 10\} \cup \{(b_{10}, a_1)\}.$$

Test 3. This test involves the transitive closure of a binary relation [2,3]. Consider the following logic program P, where *path* is an *intensional* predicate, *arc* is an *extensional* predicate, and x, y, z are variables. The query is $path(x, y)$ and the *extensional* instance I is specified by $I(arc) = \{(1, 2), (2, 3), \ldots, (9, 10)\}$.

$$path(x, y) \leftarrow arc(x, y)$$
$$path(x, y) \leftarrow path(x, z), path(z, y).$$

Test 4. This test is taken from [9]. Consider the following program P, where p, s are *intensional* predicates, q is an *extensional* predicate, and x, y, z are variables. The query is $s(x)$ and the *extensional* instance I for q consists of the following pairs, where $a - o$ are constant symbols: $I(q) = \{(a, b), (b, c), (c, d), (d, e), (e, f), (f, g), (a, h), (h, i), (i, j), (i, d), (j, k), (k, f), (a, l), (l, m), (l, i), (m, n), (n, o), (n, k), (o, g)\}$.

$$p(x, y) \leftarrow q(x, y)$$
$$p(x, y) \leftarrow q(x, z), p(z, y)$$
$$s(x) \leftarrow p(a, x).$$

Test 5. This test is taken from Example 1 using $m = 200$, $n = 50$. As shown in Table 1, the maximum number of kept tuples for the QSQN evaluation method is much larger than for the QSQN-TRE evaluation method.

Test 6. This test is taken from [3]. Consider the following program P and the following *extensional* instance I, where p, q_1, q_2 are *intensional* predicates, r_1, r_2 are *extensional* predicates, x, y, z are variables, and a_i, $b_{i,j}$ are constant symbols.

– the positive logic program P:

$$\begin{aligned} p(x,y) &\leftarrow q_1(x,y)\\ p(x,y) &\leftarrow q_2(x,y)\\ q_1(x,y) &\leftarrow r_1(x,y)\\ q_1(x,y) &\leftarrow r_1(x,z), q_1(z,y)\\ q_2(x,y) &\leftarrow r_2(x,y)\\ q_2(x,y) &\leftarrow r_2(x,z), q_2(z,y). \end{aligned}$$

– the *extensional* instance I:

$$\begin{aligned} I(r_1) &= \{(a_i, a_{i+1}) \mid 0 \le i < 10\}\\ I(r_2) &= \{(a_0, b_{1,j}) \mid 1 \le j \le 9\} \cup\\ &\quad \{(b_{i,j}, b_{i+1,j}) \mid 1 \le i < 9 \\ &\qquad \text{and } 1 \le j \le 9\} \cup\\ &\quad \{(b_{9,j}, a_{10}) \mid 1 \le j \le 9\}. \end{aligned}$$

– the query: $p(a_0, x)$.

Test 7. This test is taken from Test 2 of [5] to show a case with function symbols. It is a non-tail-recursive program. In this case, the QSQN-TRE evaluation method reduces to the QSQN method, and they have the same results. See [5] for details of the logic program and its *extensional* instance.

4.3 Discussion

Table 1 shows the comparison between the QSQN and QSQN-TRE evaluation methods. As can be seen in this table, if we use a tail-recursive program with at least a bound parameter either in the query as in Tests $[1(a), 2(a), 5, 6]$ or in the body of a rule that is related to a tail-recursive predicate as in Test 4, by not representing intermediate results during the computation, the QSQN-TRE method usually outperforms the QSQN method. In these cases, as shown in Table 1, the QSQN-TRE method reduces not only the number of accesses to the mentioned relations but also the number of kept tuples/subqueries in the memory in comparison with the QSQN method.

In contrast, for queries without any bound parameter as in Tests $[1(b), 2(b)]$ and for cases with a tail-recursive clause with more than one *intensional* predicate p in the body such that $T(p) = true$ as in Test 3, QSQN-TRE may be worse than QSQN. The explanation is similar to that of Test 1.

5 Conclusions

We have proposed the QSQN-TRE method for evaluating queries to Horn knowledge bases. It extends the QSQN method with tail-recursion elimination that allows to avoid materializing intermediate results during the processing. Similarly to QSQN, our new method also allows various control strategies such as DFS, IDFS and DAR [10,3,5].

The experimental results in Table 1 show that QSQN-TRE is better than QSQN for tail-recursive cases with at least a bound parameter in the query, especially for the positive logic program and the query given in Example 1 (as shown for Test 5 in Table 1). The preliminary comparison between QSQN, QSQR and Magic-Set reported in [3] justifies the usefulness of QSQN and hence also the usefulness of QSQN-TRE. As a future work, we will compare the methods in more detail, especially w.r.t. the number of accesses to the secondary storage when the computer memory is limited, as well as apply our method to Datalog-like rule languages for the Semantic Web [6,7].

Acknowledgments. This work was supported by Polish National Science Centre (NCN) under Grants No. 2011/02/A/HS1/00395 (for the first author) and 2011/01/B/ST6/02759 (for the second author). The first author would like to thank the Warsaw Center of Mathematics and Computer Science for support. We would also like to express our special thanks to Dr. Joanna Golińska-Pilarek from the University of Warsaw for very helpful comments and suggestions.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley (1995)
2. Beeri, C., Ramakrishnan, R.: On the power of magic. J. Log. Program. 10, 255–299 (1991)
3. Cao, S.T.: On the efficiency of Query-Subquery Nets: an experimental point of view. In: Proceedings of SoICT 2013, pp. 148–157. ACM (2013)
4. Cao, S.T.: An implementation of the QSQN-TRE evaluation methods (2014), `http://mimuw.edu.pl/~sonct/QSQNTRE14.zip`
5. Cao, S.T., Nguyen, L.A.: An Improved Depth-First Control Strategy for Query-Subquery Nets in evaluating queries to Horn knowledge bases. In: van Do, T., Thi, H.A.L., Nguyen, N.T. (eds.) Advanced Computational Methods for Knowledge Engineering. AISC, vol. 282, pp. 281–296. Springer, Heidelberg (2014)
6. Cao, S.T., Nguyen, L.A., Szalas, A.: The Web ontology rule language OWL 2 RL+ and Its extensions. T. Computational Collective Intelligence 13, 152–175 (2014)
7. Cao, S.T., Nguyen, L.A., Szalas, A.: WORL: a nonmonotonic rule language for the Semantic Web. Vietnam J. Computer Science 1(1), 57–69 (2014)
8. Freire, J., Swift, T., Warren, D.S.: Taking I/O seriously: Resolution reconsidered for disk. In: Naish, L. (ed.) Proc. of ICLP 1997, pp. 198–212. MIT Press (1997)
9. Madalińska-Bugaj, E., Nguyen, L.A.: A generalized QSQR evaluation method for Horn knowledge bases. ACM Trans. on Computational Logic 13(4), 32 (2012)
10. Nguyen, L.A., Cao, S.T.: Query-Subquery Nets. In: Nguyen, N.-T., Hoang, K., Jędrzejowicz, P. (eds.) ICCCI 2012, Part I. LNCS, vol. 7653, pp. 239–248. Springer, Heidelberg (2012)
11. Nguyen, L.A., Cao, S.T.: Query-Subquery Nets (2012), `http://arxiv.org/abs/1201.2564`
12. Ramakrishnan, R., Srivastava, D., Sudarshan, S.: Efficient bottom-up evaluation of logic programs. In: Vandewalle, J. (ed.) The State of the Art in Computer Systems and Software Engineering. Kluwer Academic Publishers (1992)
13. Ross, K.A.: Tail recursion elimination in deductive databases. ACM Trans. Database Syst. 21(2), 208–237 (1996)
14. Tamaki, H., Sato, T.: OLD resolution with tabulation. In: Shapiro, E. (ed.) ICLP 1986. LNCS, vol. 225, pp. 84–98. Springer, Heidelberg (1986)
15. Vieille, L.: Recursive axioms in deductive databases: The query/subquery approach. In: Proceedings of Expert Database Conf., pp. 253–267 (1986)
16. Vieille, L.: Recursive query processing: The power of logic. Theor. Comput. Sci. 69(1), 1–53 (1989)
17. Zhou, N.-F., Sato, T.: Efficient fixpoint computation in linear tabling. In: Proceedings of PPDP 2003, pp. 275–283. ACM (2003)