Query languages and query processing are important topics in computer science and
information technology, as they are widely used in practical applications. Query lan-
guages are usually designed so that intensional predicates can be defined by logic rules
and query evaluation can be done in polynomial time in the size of the extensional
relations. Studying query processing for a query language involves developing an eval-
uation method with a good data complexity for the language (e.g., with polynomial
time data complexity if the language allows) and then developing various optimizations
for the method.

Datalog is a well-known rule language for deductive databases. It expresses a Horn
fragment without function symbols of first-order logic. Programs in Datalog are required
to be range-restricted in the sense that every variable occurring in the head of a program
clause also occurs in the body of the clause. Datalog uses the traditional monotonic
semantics. The extension Datalog\(\neg\) of Datalog allows negation in the bodies of program
clauses and uses a non-monotonic semantics like the standard semantics for stratified
Datalog\(\neg\) programs and the well-founded semantics for the general case.

Interest in deductive databases and methods for evaluating Datalog or Datalog\(\neg\)
queries intensified in the eighties and early nineties, but “a perceived lack of compelling
applications at the time ultimately forced Datalog research into a long dormancy” [19].
As also observed by Huang et al. in their SIGMOD’2011 paper [19]:

“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. […]
As the list of applications above indicates, interest today in Datalog extends
well beyond the core database community. Indeed, the successful Datalog 2.0
Workshop held in March 2010 at Oxford University attracted over 100 attend-
ees from a wide range of areas (including databases, programming languages,
verification, security, and AI).”

During the last decade, rule-based query languages, including languages related to
Datalog, were also intensively studied for the Semantic Web (e.g., in [4, 7, 13, 14, 15,
16, 20, 22, 23, 28, 30]). In general, since deductive databases and knowledge bases are
widely used in practical applications, improvements for processing recursive queries are
always desirable. Due to the importance of the topic, it is worth doing further research
on the topic.
Horn knowledge bases are extensions of Datalog deductive databases without the range-restrictedness and function-free conditions [1]. As argued in [22], the Horn fragment of first-order logic plays an important role in knowledge representation and reasoning. A Horn knowledge base consists of a positive logic program for defining intensional predicates and an instance of extensional predicates. When the knowledge base is too big, not all of the extensional and intensional relations may be totally kept in the computer memory and query evaluation may not be totally done in the computer memory. In such cases, the system usually has to load (resp. unload) relations from (resp. to) the secondary storage. Thus, in contrast to logic programming, for Horn knowledge bases efficient access to the secondary storage is a very important aspect.

In the dissertation, we study query processing for Horn knowledge bases. Particularly, we concentrate on developing efficient methods for evaluating queries to Horn knowledge bases. In addition, query evaluation for stratified knowledge bases is also investigated. This topic has not been well studied as query processing for the Datalog-like deductive databases or the theory and techniques of logic programming.

1 Related Work and Motivation

The mentioned topic is worth studying due to the following reasons:
1. The most well-known methods for evaluating queries to Datalog deductive databases or Horn knowledge bases are QSQR [1, 22] and Magic-Sets [5, 6, 27] (by Magic-Sets we mean the evaluation method that combines the magic-set transformation with the improved semi-naive bottom-up evaluation method). Both of these methods are goal-directed. However, as observed by Vieille [31], the QSQR approach is like iterative deepening search. It allows redundant recomputations (see [22, Remark 3.2]). On the other hand, the Magic-Sets method applies breadth-first search. The following example shows that the breadth-first approach is not always efficient [24].

Example 1. The order of program clauses and the order of atoms in the bodies of program clauses may be essential, e.g., when the positive logic program that defines intensional predicates is specified using the Prolog programming style. In such cases, the top-down depth-first approach may be much more efficient than the breadth-first approach. Here is such an example, in which \( p, q_1 \) and \( q_2 \) are intensional predicates, \( r_1 \) and \( r_2 \) are extensional predicates, \( x, y \) and \( z \) are variables, \( a_i \) and \( b_{i,j} \) are constant symbols:

- the positive logic program:

\[
\begin{align*}
p & \leftarrow q_1(a_0, a_m) \\
p & \leftarrow q_2(a_0, a_m) \\
q_1(x, y) & \leftarrow r_1(x, y) \\
q_1(x, y) & \leftarrow r_1(x, z), q_1(z, y) \\
q_2(x, y) & \leftarrow r_2(x, y) \\
q_2(x, y) & \leftarrow r_2(x, z), q_2(z, y)
\end{align*}
\]
the extensional instance (illustrated in Figure 1):

\[
I(r_1) = \{(a_i, a_{i+1}) \mid 0 \leq i < m\}
\]

\[
I(r_2) = \{(a_0, b_{1,j}) \mid 1 \leq j \leq n\} \cup \{(b_{i,j}, b_{i+1,j}) \mid 1 \leq i < m - 1 \text{ and } 1 \leq j \leq n\} \cup \{(b_{m-1,j}, a_m) \mid 1 \leq j \leq n\}
\]

the query: \( \leftarrow p \).

Fig. 1: An illustration for the extensional instance given in Example 1.

Notice that the depth-first approach needs only \( \Theta(m) \) steps for evaluating the query, while the breadth-first approach performs \( \Theta(m \cdot n) \) steps. When \( n \) is comparable to \( m \), the difference is too big. The magic-sets transformation does not help for this case.

Our postulate is that the breadth-first approach (including the Magic-Sets evaluation method) is inflexible and not always efficient. Of course, depth-first search is not always good either. It is worth developing evaluation methods for evaluating queries to Horn knowledge bases that are more efficient than the QSQR evaluation method and more adjustable than the Magic-Sets evaluation method. In particular, good methods should be not only set-oriented and goal-directed but should also reduce computational redundancy as much as possible and allow various control strategies. For this purpose, we formulate query-subquery nets and use them to develop the first framework for developing algorithms for evaluating queries to Horn knowledge bases. The framework forms a generic evaluation method called QSQN.

2. 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 runtime. Such a memory consumption may be 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. A tail-recursive program can be compiled as efficiently as iterative programs by applying tail-recursion elimination. Ross’ work [29] contains a very good example about the usefulness of tail-recursion elimination. Let’s consider a slightly modified version of that example.

Example 2. Let \( P \) be the positive logic program consisting of the following clauses:
\[
\begin{align*}
p(x, y) & \leftarrow e(x, z), p(z, y) \\
p(m, x) & \leftarrow t(x)
\end{align*}
\]
where \( p \) is an intensional predicate, \( e \) and \( t \) are extensional predicates, \( m \) is a natural number (a constant) and \( x, y, z \) are variables. Let \( p(1, x) \) be the query, \( n \) a natural number, and let the extensional instance \( I \) for \( e \) and \( t \) be as follows:
\[
I(e) = \{(1, 2), (2, 3), \ldots, (m-1, m), (m, 1)\},
\]
\[
I(t) = \{1, \ldots, n\}.
\]
To make this example more concrete, suppose that: \( e(x, z) \) holds when there is a way to get from town \( x \) to town \( z \), where the towns are numbered from 1 to \( m \) and \( m \) denotes the capital; \( t(x) \) holds when item \( x \) is available in the capital; items are numbered from 1 to \( n \) and all items are available in the capital; \( p(z, y) \) holds if it is possible to get from town \( z \) to a town that has item \( y \). For the query \( p(1, x) \), the task is to find all available items starting from town 1. To answer the query, methods such as QSQR, QSQN, Magic-Sets would evaluate every subquery of the form \( p(i, x) \), where \( 1 \leq i \leq m \), and thus store \( m \times n \) tuples \((i, j)\) in the answer relation for \( p \), where \( 1 \leq i \leq m \) and \( 1 \leq j \leq n \). As can be seen, for answering the query \( p(1, x) \), we do not need to store the intermediate answer tuples \((i, j)\) with \( i > 1 \) for \( p \) if we apply tail-recursion elimination. We only need to store \( n \) answer tuples \((1, j)\) with \( 1 \leq j \leq n \) and \( m \) subqueries \((i, x)\) with \( 1 \leq i \leq m \) for \( p \). That is, we need to store only \( m + n \) instead of \( m \times n \) tuples. The example in Ross’ work [29] considers \( m = 100 \) towns and \( n = 1000 \) items, and it is easy to see how big the difference is.

It is desirable to study how to incorporate tail-recursion elimination into query-subquery nets and develop the corresponding variants of the QSQN evaluation method.

3. As mentioned earlier, Datalog is a well-known rule language for deductive databases. It uses the traditional monotonic semantics. The extension Datalog\(^\neg\) of Datalog allows negation in the bodies of program clauses and uses a non-monotonic semantics like the standard semantics for stratified Datalog\(^\neg\) programs and the well-founded semantics for the general case [2, 3, 17, 18]. It is worth studying how to incorporate stratified negation into query-subquery nets for evaluating queries to stratified knowledge bases.
2 Definitions for Horn Knowledge Bases and Stratified Knowledge Bases

This section recalls the classical notions and definitions from first-order logic and database theory which can be found, e.g., in [1, 21]. Most of our exposition here is taken from Section 2 of [22], with minor modifications.

A signature for first-order logic consists of constant symbols, variable symbols, function symbols, and predicate symbols. A term is either a constant, a variable or of the form \( f(t_1, \ldots, t_n) \), where \( f \) is a function symbol and each \( t_i \) is a term. An atom is an expression of the form \( p(t_1, \ldots, t_m) \), where \( m \geq 0 \), \( p \) is an \( m \)-ary predicate and each \( t_i \) is a term. Formulas are defined in the usual way. If \( \varphi \) is a formula then by \( \forall(\varphi) \) we denote the universal closure of \( \varphi \), which is the formula obtained by adding a universal quantifier for every variable having a free occurrence in \( \varphi \). An expression is either a term, a tuple of terms, a formula without quantifiers or a list of formulas without quantifiers.

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 term-depth of an expression (resp. a substitution) is the maximal nesting depth of function symbols occurring in that expression (resp. substitution).

**Definition 2.1 (Positive Program Clause).** A positive (or definite) program clause is a formula of the form \( \forall (A \lor \neg B_1 \lor \ldots \lor \neg B_k) \) with \( k \geq 0 \), written as \( A \leftarrow B_1, \ldots, B_k \), where \( A, B_1, \ldots, B_k \) are atoms. \( A \) is called the head, and \( (B_1, \ldots, B_k) \) the body of the program clause. If \( k = 0 \) then the clause is called a unit clause (i.e., a definite program clause with an empty body). If \( p \) is the predicate of \( A \) then the program clause is called a program clause defining \( p \).

**Definition 2.2 (Positive Logic Program).** A positive (or definite) logic program is a finite set of (positive) program clauses.

**Definition 2.3 (Goal).** A goal (also called a negative clause) is a formula of the form \( \forall (\neg B_1 \lor \ldots \lor \neg B_k) \), written as \( \langle B_1, \ldots, B_k \rangle \), where \( B_1, \ldots, B_k \) are atoms. If \( k = 1 \) then the goal is called a unary goal. If \( k = 0 \) then the goal stands for falsity and is called the empty goal (or the empty clause) and denoted by \( \square \).

**Definition 2.4 (Tail-Recursion).** A program clause \( A \leftarrow B_1, \ldots, B_k \), for \( k > 0 \), is said to be recursive whenever some \( B_i \) (\( 1 \leq i \leq k \)) has the same predicate as \( A \). If \( B_k \) has the same predicate as \( A \) then the clause is tail-recursive and in this case the predicate of \( B_k \) is a tail-recursive predicate.

Similarly as for deductive databases, 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.

**Definition 2.5 (Horn Knowledge Base).** A Horn knowledge base is defined to be a pair \((P, I)\), where \( P \) is a positive logic program for defining intensional predicates,
and \( I \) is a \textit{generalized extensional instance}, which is a mapping that associates each extensional \( n \)-ary predicate with an \( n \)-ary generalized relation.

Given a Horn knowledge base specified by a positive logic program \( P \) and an extensional instance \( I \), a \textit{query} to the knowledge base is a positive formula \( \varphi(x) \) without quantifiers, where \( x \) is a tuple of all the variables of \( \varphi \).\(^1\) A (correct) \textit{answer} for the query is a tuple \( t \) of terms of the same length as \( x \) such that \( P \cup I \models \forall \varphi(t) \). When measuring \textit{data complexity}, we assume that \( P \) and \( \varphi \) are fixed, while \( I \) varies. Thus, the pair \( (P, \varphi(x)) \) is treated as a \textit{query} to the extensional instance \( I \). We will use the term “query” in this meaning.

It can be shown that every query \( (P, \varphi(x)) \) can be transformed in polynomial time to an equivalent query of the form \( (P', q(x)) \) over a signature extended with new intensional predicates, including \( q \). Without loss of generality, we will consider only queries of the form \( (P, q(x)) \), where \( q \) is an intensional predicate. Answering such a query on an extensional instance \( I \) is to find (correct) answers for \( P \cup I \cup \{ \leftarrow q(x) \} \).

**Definition 2.6 (Literal).** A \textit{literal} is an atom or the negation of an atom. A \textit{positive literal} is an atom. A \textit{negative literal} is the negation of an atom.

**Definition 2.7 (Safe Logic Program).** A \textit{safe program clause} (w.r.t. the leftmost selection function) is an expression of the form \( A \leftarrow B_1, \ldots, B_k \) with \( k \geq 0 \), such that:
- \( A \) is an atom and each \( B_i \) is a literal,
- every variable occurring in \( A \) occurs also in \( B_1, \ldots, B_k \),
- every variable occurring in a negative literal \( B_j \) in the body of a program clause occurs also in some positive literals \( B_i \) in the body of that clause such that \( i < j \).

A \textit{safe logic program} (w.r.t. the leftmost selection function) is a finite set of safe program clauses.

**Definition 2.8 (Stratification).** Given a safe logic program \( P \), a \textit{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 the following properties:
- if an intensional predicate \( p \) occurs in a positive literal 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 of a clause from \( P_i \) with \( i > 1 \), then the clauses defining \( p \) must belong to \( P_1 \cup \ldots \cup P_{i-1} \).

Each \( P_i \) is called a \textit{stratum} of the stratification.

**Definition 2.9 (Stratified Logic Program).** A safe logic program is called a \textit{stratified logic program} if it has a stratification.

**Definition 2.10 (Stratified Knowledge Base).** A \textit{stratified knowledge base} is defined to be a pair \( (P, I) \), where \( P \) is a stratified logic program for defining intensional predicates and \( I \) is an instance of extensional predicates.

\(^1\)A \textit{positive formula without quantifiers} is a formula built up from atoms using only connectives \( \land \) and \( \lor \).
3 Our Contributions

This dissertation studies query processing for Horn knowledge bases, which is a topic that has not been well studied as query processing for the Datalog-like deductive databases or the theory and techniques of logic programming. In addition, query evaluation for stratified knowledge bases is also investigated. Our main contributions are the following:

Chapter 3:

− We formulate query-subquery nets and use them to develop the first framework for developing algorithms for evaluating queries to Horn knowledge bases with the following good properties:
  ● the approach is goal-directed,
  ● each subquery is processed only once,
  ● each supplement tuple, if desired, is transferred only once,
  ● operations are done set-at-a-time,
  ● any control strategy can be used.

The intention of our framework is to increase efficiency of query processing by eliminating redundant computation, increasing adjustability\(^2\) and reducing the number of accesses to the secondary storage. QSQ-nets are a more intuitive representation than the description of the QSQ approach of Datalog given in [1]. Particularly, we transform a logic program into an equivalent net structure and use it to determine which set of tuples or subqueries should be evaluated at each step, in an efficient way. Our notion of QSQ-net makes a connection to flow networks and is intuitive for developing efficient evaluation algorithms. One of the key differences is that we do not use adornments and annotations, but use substitutions instead. This is natural for the case with function symbols and without the range-restrictedness condition.

The framework forms a generic evaluation method called QSQN. It is sound and complete, and has polynomial time data complexity when the term-depth bound is fixed. This method is designed so that the query processing is divided into appropriate steps which can be delayed to maximize adjustability and allow various control strategies. In comparison with the most well-known evaluation methods, the generic QSQN evaluation method does not do redundant recomputations as the QSQR evaluation method and is more adjustable and thus has essential advantages over the Magic-Sets evaluation method. The results were published in [8, 24, 25].

Chapter 4:

− We incorporate tail-recursion elimination into query-subquery nets in order to obtain the QSQN-TRE evaluation method for Horn knowledge bases. The aim is to reduce materializing the intermediate results during the processing of a query

\(^{2}\)By “adjustability” we mean easiness in adopting advanced control strategies.
with tail-recursion. We give an intuition and a formal definition of the QSQN-TRE method as well as explanations, an illustrative example and a pseudocode of the evaluation algorithm. The results were published in [12].

− We prove the soundness and completeness of the QSQN-TRE method and show that when the term-depth bound is fixed, the method has polynomial time data complexity.

− We extend QSQN-TRE to obtain another evaluation method called QSQN-rTRE, which can eliminate not only tail-recursive predicates but also intensional predicates that appear rightmost in the bodies of the program clauses. The aim is to reduce materializing the intermediate results (when desired) during the processing. The method was published in [9].

Chapter 5:

− We incorporate stratified negation into query-subquery nets to obtain a method called QSQN-STR for evaluating queries to stratified knowledge bases. We give a formal definition of the QSQN-STR method, an illustrative example and a pseudocode of the evaluation algorithm. The proposed method was published in [10].

− We prove the soundness and completeness of QSQN-STR for the case without function symbols.

Chapter 6:

− We first present the IDFS control strategy, which was published in [11] and can be used for the evaluation methods QSQN, QSQN-TRE and QSQN-rTRE. In order to compare our methods with the well-known evaluation methods such that QSQR and Magic-Sets, we have implemented all of these methods. We then provide the experimental results and a discussion on the performance of the proposed evaluation methods. We compare them using representative examples that appear in many articles on deductive databases as well as new ones. We also report experimental results of QSQN-STR using a control strategy called IDFS2, which is a modified version of IDFS. The experimental results confirm the efficiency and usefulness of the proposed evaluation methods.

References


