Integrating Multi-threading into Query-Subquery Nets

Son Thanh Cao(✉), Phan Anh Phong, and Le Quoc Anh

School of Engineering and Technology, Vinh University, 182 Le Duan Street, Vinh, Nghe An, Vietnam
{sonct,phongpa,anhlq}@vinhuni.edu.vn

Abstract. In this paper, we propose a new method, named QSQN-MT, for the evaluation of queries to Horn knowledge bases. Particularly, we integrate multi-threading into query-subquery nets to reduce the execution time for evaluating a query over a logic program regardless of the order of clauses. The usefulness of the proposed method is indicated by the experimental results.

Keywords: Horn knowledge bases · Query processing · Deductive databases · QSQN · QSQN-MT · Multi-threading

1 Introduction

In first-order logic (FOL), the Horn fragment has received much attention from researchers because of its important roles in the logic programming and deductive database communities. Horn knowledge bases (Horn KBs) are an extension of Datalog deductive databases [1]. Various methods have been arised for Datalog or Horn KBs such as (i) the top-down methods including QSQ [14], QSQR [9], QSQN [5,11] and (ii) the bottom-up method including Magic-Set [2].

Normally, clauses in a logic program are processed in order. This means that a clause is executed when the processing of preceding clauses has finished. Particularly, there is always only one process being executed at a specific time. Multi-threading has been adopted in logic program implementations in order to improve the execution time [10,12,13]. By using multi-threading, we can perform multiple operations at once (simultaneously) in a program. This integration allows a single processor to share multiple and concurrent threads. Each thread executes its own sequence of instructions or clauses.

Example 1. When executing a logic program, the order of clauses in this program may be important and can affect the execution time to find solutions. For instance, consider the logic program P including (i) intensional predicates: $reachable$, $reachable_1$, $reachable_2$ and $reachable_3$; (ii) extensional predicates: $link_1$, $link_2$, and $link_3$; (iii) variables: x, y and z; (iv) constant symbols: a_i, b_i and c_i; and (v) a natural number: n.

D.-T. Tran et al. (Eds.): ICISN 2021, LNNS 243, pp. 189–196, 2021.
https://doi.org/10.1007/978-981-16-2094-2_24

– the logic program P (for defining $reachable$, $reachable_1$, $reachable_2$ and $reachable_3$):

$$reachable(x, y) \leftarrow reachable_1(x, y) \quad (1)$$
$$reachable(x, y) \leftarrow reachable_2(x, y) \quad (2)$$
$$reachable(x, y) \leftarrow reachable_3(x, y) \quad (3)$$
$$reachable_1(x, y) \leftarrow link_1(x, y) \quad (4)$$
$$reachable_1(x, y) \leftarrow link_1(x, z), reachable_1(z, y) \quad (5)$$
$$reachable_2(x, y) \leftarrow link_2(x, y) \quad (6)$$
$$reachable_2(x, y) \leftarrow link_2(x, z), reachable_2(z, y) \quad (7)$$
$$reachable_3(x, y) \leftarrow link_3(x, y) \quad (8)$$
$$reachable_3(x, y) \leftarrow link_3(x, z), reachable_3(z, y). \quad (9)$$

– the extensional instance I (for specifying $link_1$, $link_2$ and $link_3$) is demonstrated in Fig. 1),
– the query: $reachable(a_0, a_n)$. ◁

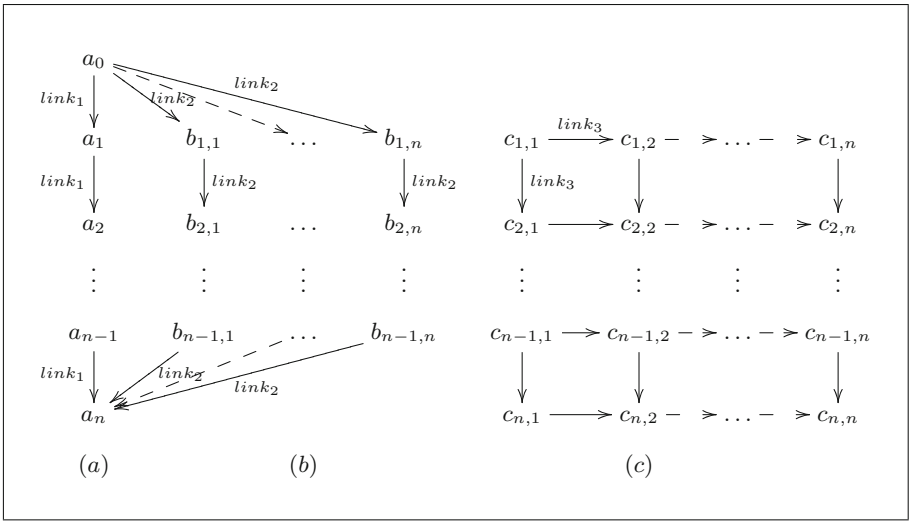

Fig. 1. The extensional instance I: (a) $I(link_1)$, (b) $I(link_2)$, and (c) $I(link_3)$.

As can easily be seen in Fig. 1, the relation $link_1$ (reps. $link_2$ and $link_3$) contains n (reps. n^2 and $2(n^2 - n)$) instances. For instance, if $n = 100$ then the relations $link_1$, $link_2$ and $link_3$ include 100, 10 000 and 19 800 instances, respectively. The relations $link_1$ and $link_2$ contain instances related to answer the query $reachable(a_0, a_n)$ but the relation $link_3$ does not.

Normally, a logic program is executed sequentially until getting results (e.g., the order of executing the program P is a sequence of the clauses (1), (2) and (3),

together with the related ones). The question is, what will happen if we swap the order of program clauses in P so that the clause (2) is executed first? Clearly, this takes a long time to get the answer for the mentioned query since the size (i.e., the number of instances) of relation $link_2$ (which is defined by the clauses (6) and (7)) is much bigger than $link_1$ (which is defined by the clauses (4) and (5)). It is worth studying how to compute the answers to a query over a logic program regardless of the order of program clauses.

In this paper, we integrate multi-threading into QSQN framework to develop a new method for evaluating queries to Horn KBs, named QSQN-MT. Our intention is to reduce the execution time for evaluating a query over a logic program regardless of the order of clauses in this program. The experimental results indicate the outperformance of the QSQN-MT method. Due to space limitations, the reader could refer to [5,8] for the basic notions and definitions such as *term*, *atom*, *substitution*, *predicate*, *unification*, *Horn KBs*, *query* and other related ones. The rest of the paper is structured as follows. Sect. 2 outlines an overview of QSQN[1] and presents a new method called QSQN-MT. The tested results are provided in Sect. 3. Section 4 gives conclusions of the paper.

2 Query-Subquery Nets with Multi-threading

In this section, we first give an overview of the QSQN method and then present a new method for evaluating queries to Horn KBs, named QSQN-MT.

2.1 An Overview of Query-Subquery Nets

In [5,11], Nguyen and Cao formulated a framework *query-subquery net*, which is used to develop methods for evaluating queries to Horn KBs with the intension of improving the efficiency of query processing by (*i*) decreasing redundant computation, (*ii*) increasing flexibility, and (*iii*) minimizing the number of read/write operations to disk. Using this framework, we proposed an evaluation method named QSQN. The method is goal-directed, set-at-a-time, and has been developed to allow dividing the query processing into smaller steps to maximize adjustability (i.e., we can apply various flow-of-control strategies in QSQN, which are similar to search strategies in a graph and called *control strategies* for short). In particular, the given logic program is transformed into a corresponding net structure, which is used to specify set of tuples/subqueries in each node should be processed at each step. The proofs given in [3] showed that the QSQN evaluation method (as well as its extensions) is sound, complete and has PTIME data complexity with a condition of fixing the term-depth bound. For a more explanation of running example and relating QSQN to SLD-Resolution with tabulation, the reader could refer to [6, Section 3] for further reading. The other definitions related to QSQN structure, QSQN and a *subquery* are provided

[1] A demonstration in the PowerPoint-like mode to help the readers figure out the gist of QSQN is provided in [4].

in [11]. Also, to help the readers figure out the gist of QSQN, a detailed demonstration in the PowerPoint-like mode is provided in [4]. The experimental results shown in [3,6] indicate the usefulness of the QSQN evaluation method and its extensions.

2.2 Integrating Multi-threading into Query-Subquery Nets

In this subsection, we present an extension of the QSQN method proposed in [11] by integrating multi-threading into QSQN forming a new evaluation method called QSQN-MT.

The definition of a QSQN-MT structure (reps. QSQN-MT) is analogous to the definition of QSQN structure (reps. QSQN). Due to space limitations, we omit to present the details for brevity. We refer the reader to [5,11] for further understanding. In [5], we proposed an algorithm for evaluating queries to Horn KBs. We now present an extension of this algorithm to deal with multi-threading. From now on in this section, a logic program is denoted by P.

Algorithm 1: `evaluating the query` $(P, q(\overline{x}))$ `on EDB instance` I.

```
1 initialize a QSQN-MT of P and related ones;
2 let n be the number of threads detected;
3 for i = 0 to n − 1 do
4 |  thread[i] = new Thread();                    // initialize the thread i
5 for i = 0 to n − 1 do
6 |  thread[i].start();          // excute the procedure run(i) for thread i
7 for i = 0 to n − 1 do
8 |  thread[i].join();                          // wait until sub-threads finish
9 return the results.
```

Procedure run(i)

Purpose: `running the thread` i.

```
1 while there exists (u, v) ∈ E w.r.t. thread i s.t. active-edge(u, v) returns true
  do
2 |  select (u, v) ∈ E w.r.t. thread i s.t. active-edge(u, v) return true;
  |     // arbitrary control strategies for the selection of (u, v)
3 |  fire(u, v);
```

Algorithm 1 describes steps of the QSQN-MT evaluation method for Horn KBs, which is a modified version of the one given in [5] for QSQN by integrating multi-threading into QSQN. The algorithm first automatically detects the number of threads and then concurrently executes these threads until getting results (the method `start()` in the step 6 of Algorithm 1 calls the procedure `run(i)` w.r.t. the thread i). Each thread represents a flow-of-control strategies in QSQN-MT.

The procedure `run` (on page 4) uses the function `active-edge`(u, v) (specified in [5]). For an edge (u, v), the function `active-edge`(u, v) returns *true* if there

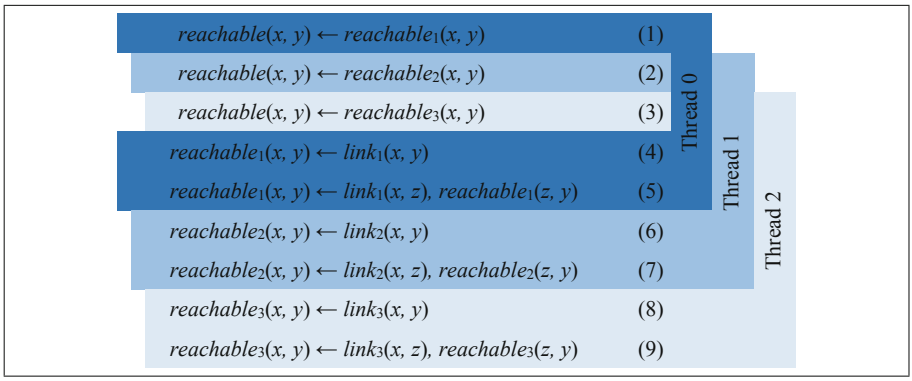

Fig. 2. An example of splitting the program P stated in Example 1 into 3 threads.

are some collected data in u (i.e., tuples or subqueries) that can be evaluated to generate data and transfer through (u, v), otherwise, this function returns *false*. If `active-edge`(u, v) returns *true*, the procedure `fire`(u, v) (specified in [5]) will evaluate unprocessed data collected in u and then transfer relevant data through (u, v). The procedure `fire`(u, v) calls the procedure `transfer`(D, u, v) (stated in [5]), which is used to determine the effectiveness of transferring data D through (u, v). These functions and procedures are also used for QSQN-MT.

An example of splitting the program P given in Example 1 into 3 threads is illustrated in Fig. 2. The first thread (thread 0) includes clauses (1), (4) and (5) of P. The second thread (thread 1) consists of clauses (2), (6) and (7) of P. The last one (thread 2) includes clauses (3), (8) and (9) of P. These threads run concurrently and independently of each other. Hence, some threads may run faster than others w.r.t. the execution time according to the size of EDB relations.

3 Preliminary Experiments

In [6], we have made a comparison between QSQN (together with its extensions) and DES-DBMS[2] as well as SWI-Prolog w.r.t the execution time. The experimental results described in [6] indicate the usefulness of QSQN as well as its extensions.

3.1 Experimental Settings

We have implemented prototypes of QSQN and QSQN-MT in Java, using a control strategy name IDFS proposed in [3], which is intuitively demonstrated in [4]. These prototypes use extensional relations which are stored in a MySQL

[2] The Datalog Education System (DES), a deductive database system with a DBMS via ODBC, available at http://des.sourceforge.net.

database. All the tests were executed on the Microsoft Windows 10 (64 bit) platform with Intel(R) Core(TM) i3-2350M CPU @ 2×2.30 GHz and 8GB of RAM. The current prototypes of QSQN and QSQN-MT allow to evaluate the query of the following syntax: $q(\bar{t})$, in which, $\bar{t}$ is a tuple of terms. For the query that has one answer being either *true* or *false* (e.g., $reachable(a_0, a_n)$), we use a flag to break the computation at the time getting the *true* answer. The package [4] also contains all of the below tests as well as prototypes of QSQN and QSQN-MT.

Reconsider the program P and the EDB instance I specified in Example 1. As mentioned, the clauses (4) and (5) (reps. (6), (7) and (8), (9)) are used for defining clause (1) (reps. (2) and (3), respectively). Thus, the program is executed mainly based on the order of clauses (1), (2) and (3). We examine the tests specified by changing the order of the first three rules of program clauses in P as follows: Test 1 ((1), (2), (3)); Test 2 ((2), (1), (3)) and Test 3 ((3), (2), (1)). Each test is performed with the query $reachable(a_0, a_n)$ using the following values of n: 20, 40, 60, 80 and 100, respectively.

3.2 Experimental Results

Figure 3 illustrates a comparison between our prototype of QSQN-MT and QSQN w.r.t. the execution time for Tests 1–3. We have executed each test case ten times to measure the execution time in milliseconds and averaged the results. To provide better data visualization, the average execution time of each method reported in Fig. 3 was converted to log_{10}.

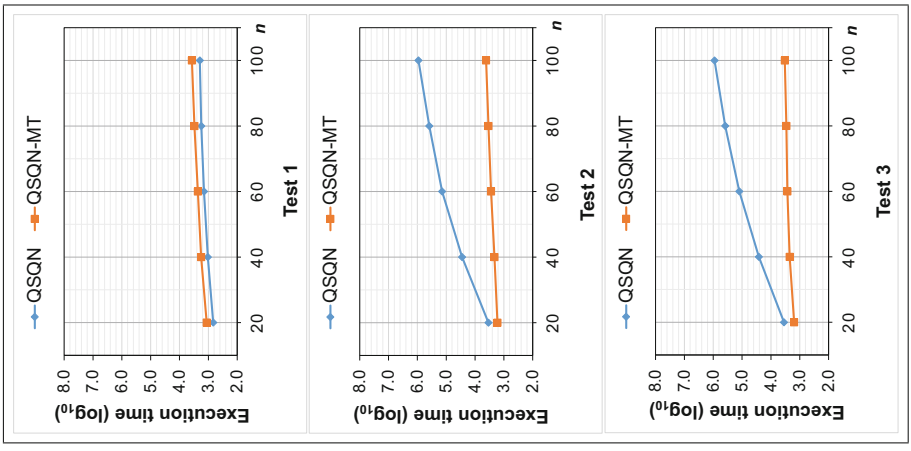

Fig. 3. Experimental results for Tests 1, 2 and 3.

As can be seen in Fig. 3 for Tests 1, 2 and 3, the execution time for the QSQN method is different when the order of program clauses are changed. In Test 1, the

QSQN-MT method takes a little bit more time than the QSQN method since it has to wait until all threads terminate. In all mentioned tests, when integrating multi-threading in QSQN, the execution time of the QSQN-MT is almost the same regardless of the order of program clauses.

4 Conclusions

A method, named QSQN-MT, for evaluating queries over a logic program has been proposed. With multi-threading, the logic program does multiple tasks concurrently in order to increase the performance. The results of experiments indicated that QSQN together with multi-threading can make the proposed method performs better than it would be with a single thread. The flow of answering a query using multi-threading in QSQN can be treated as a control strategy, thus QSQN-MT inherits all good properties of QSQN such that: goal-directed, set-at-a-time, sound, complete and having PTIME data complexity. Of course, having multiple threads is not always efficient and there are many other issues related to multi-threading that we need to be concerned about. As future work, we will make a comparison between the proposed method and other related applications as well as apply our method in parallel computation with multi-processors [7].

Acknowledgments. We are extremely grateful to dr hab. L.A. Nguyen from the Institute of Informatics, University of Warsaw, Poland for his helpful comments.

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.: Methods for evaluating queries to Horn knowledge bases in first-order logic. Ph.D. dissertation. University of Warsaw (2016). http://mimuw.edu.pl/~sonct/stc-thesis.pdf
4. Cao, S.T.: A prototype implemented in Java of the QSQN and QSQN-MT evaluation methods (2020). http://mimuw.edu.pl/~sonct/QSQN-MT.zip
5. Cao, S.T., Nguyen, L.A.: Query-subquery nets for Horn knowledge bases in first-order logic. J. Inf. Telecommun. **1**(1), 77–99 (2017)
6. Cao, S.T., Nguyen, L.A.: Incorporating stratified negation into query-subquery nets for evaluating queries to stratified deductive databases. Comput. Inform. **38**, 19–56 (2019)
7. Fidjeland, A.K., Luk, W., Muggleton, S.H.: Customisable multi-processor acceleration of inductive logic programming. In: Latest Advances in Inductive Logic Programming, pp. 123–141 (2014)
8. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer (1987)
9. Madalińska-Bugaj, E., Nguyen, L.A.: A generalized QSQR evaluation method for Horn knowledge bases. ACM Trans. Comput. Log. **13**(4), 32 (2012)

10. Marques, R., Swift, T., Cunha, J.: Extending tabled logic programming with multi-threading : a systems perspective. In: Proceedings of CICLOPS 2008, pp. 91–106 (2008)
11. Nguyen, L.A., Cao, S.T.: Query-subquery nets. In: Proceedings of ICCCI 2012. LNCS, vol. 7635, pp. 239–248. Springer (2012)
12. Taokok, S., Pongpanich, P., Kerdprasop, N., Kerdprasop, K.: A multi-threading in prolog to implement K-mean clustering. In: Latest Advances in Systems Science and Computational Intelligence, pp. 120–126. WSEAS Press (2012)
13. Umeda, M., Katamine, K., Nagasawa, I., Hashimoto, M., Takata, O.: Multi-threading inside prolog for knowledge-based enterprise applications. In: Proceedings of INAP 2005, pp. 200–214. Springer (2005)
14. Vieille, L.: Recursive axioms in deductive databases: the query/subquery approach. In: Proceedings of Expert Database Systems, pp. 179–193 (1986)