Evaluating Datalog via Structure-Aware Rewriting

Hangdong Zhao[1], Paraschos Koutris[1] and Shaleen Deep[2]

[1]University of Wisconsin-Madison, Madison, USA
[2]Microsoft

Abstract

Datalog is a powerful yet elegant language that allows expressing recursive computation. Although Datalog evaluation has been extensively studied in the literature, so far, only loose upper bounds are known on how fast a Datalog program can be evaluated. In this work, we ask the following question: given a Datalog program, what is the tightest possible runtime? To this end, our main contribution is a structure-aware query evaluation technique that computes a Datalog program by first rewriting it to generate the smallest possible grounding. Using this technique we can obtain state-of-the-art theoretical runtime results. We also give some preliminary evidence that the rewriting method can lead to practical runtime improvements.

Keywords

Datalog, tree decomposition, rewriting

1. Introduction

Datalog is a recursive query language that has gained prominence due to its expressivity and rich applications across multiple domains, including graph processing [1], declarative program analysis [2, 3], and business analytics [4]. Many graph analytics tasks (e.g., pattern finding) and program analysis tasks such as Dyck-reachability [5, 6, 7], context-free reachability [8], and Andersen's analysis [9] can be naturally cast as Datalog programs.

Much prior work has sought to characterize the complexity of Datalog evaluation. The general data complexity of Datalog is P-complete [10, 11]. Some fragments of Datalog can have lower data complexity: the evaluation for non-recursive Datalog is in AC_0, whereas evaluation for Datalog with linear rules is in NC and thus efficiently parallelizable [12, 13]. However, all such results do not tell us how efficiently we can evaluate a given Datalog program. Furthermore, the current general algorithmic techniques for Datalog evaluation typically aim for an imprecise polynomial bound instead of specifying the tightest possible exponent. Semi-naïve or naïve evaluation only provides upper bounds on the number of iterations, ignoring the computational cost of each iteration.

Endeavors to pinpoint the exact data complexity for Datalog fragments have focused on the class of Conjunctive Queries (CQs) [14, 15, 16] and union of CQs [17]. When recursion is involved, however, exact runtimes are known only for special classes of Datalog [18, 19, 20, 21]. The seminal work of Yannakakis [22] established a $O(n^3)$ runtime for chain Datalog programs, where n is the size of the active domain. Such programs have a direct correspondence to context-free grammars and capture a fundamental class of static analysis known as context-free reachability (CFL-reachability). When the chain Datalog program corresponds to a regular grammar, the runtime can be further improved to $O(m \cdot n)$, where m denotes the size of the input data. An $O(n^3)$ algorithm was proposed for the Datalog program that captures Andersen's analysis [8]. Recently, Casel and Schmid [23] studied the fine-grained complexity of evaluation, enumeration, and counting problems over regular path queries (also a Datalog fragment) with similar upper bounds. However, none of these techniques generalize to arbitrary Datalog programs.

Our Contribution In [24], we address the following question: given a Datalog program P, what is the *tightest possible runtime as a function of the input size m and the size of the active domain n*?

Datalog 2.0 2024: 5th International Workshop on the Resurgence of Datalog in Academia and Industry, October 11, 2024, Dallas, Texas, USA

hzhao284@wisc.edu (H. Zhao); paris@cs.wisc.edu (P. Koutris)

To answer this question, we propose a general framework for evaluating P based on rule rewriting. The key idea is based on the fundamental result that a *grounded* Datalog program (i.e., one where every rule has constants) can be evaluated in time linear w.r.t. its size. Hence, the goal of the framework is to transform P into an equivalent grounded program that is as small as possible. Though constructing grounded programs naïvely is rather straightforward, we show that groundings of smaller size are attainable via rewriting P using tree decomposition techniques. We apply our framework to prove state-of-the-art and new results for practical Datalog fragments (e.g. linear Datalog).

This paper summarizes the key results from [24] and seeks to experimentally evaluate the idea of rewriting a Datalog program P to obtain the smallest possible grounding. We show that the rewriting can often lead to significant speed-ups in evaluation.

2. Preliminaries

Datalog consists of a set of rules over a set of extensional and intensional relations (simply referred to as EDB and IDB respectively, henceforth). EDBs correspond to relations in a given database, each comprising a set of EDB tuples, whereas IDBs are derived by the rules. The head of each rule is an IDB, and the body of the rule consists of zero or more EDBs and IDBs as a Conjunctive Query defining how the head IDB is derived. We illustrate with the example of transitive closure on a binary EDB R denoting edges in a directed graph, a single IDB T, and two rules:

$$\mathsf{T}(x, y) \leftarrow \mathsf{R}(x, y).$$
$$\mathsf{T}(x, y) \leftarrow \mathsf{R}(x, z), \mathsf{T}(z, y).$$

For every Datalog program, we will assume that there is a unique IDB that we identify as the target (or output) of the program. We use $\mathsf{arity}(P)$ to denote the maximum number of variables contained in any IDB of a program P. We say that a Datalog program P is *monadic* if every IDB is unary; a Datalog program is *linear* if the body of every rule contains at most one IDB. A *chain* Datalog program is a program where every rule has the following form:

$$\mathsf{T}(x_1, x_{k+1}) \leftarrow \mathsf{T}_1(x_1, x_2), \mathsf{T}_2(x_2, x_3), \ldots, \mathsf{T}_k(x_k, x_{k+1}).$$

A chain Datalog program corresponds to a Context-Free Grammar (CFG).

We use m to denote the sum of sizes of the input EDB relations to a Datalog program. We use n to denote the size of the active domain of EDB relations (i.e., the number of distinct constants that occur in the input). We assume data complexity, i.e. the program size (the total number of predicates and the variables) is a constant. We use the standard word-RAM model with $O(\log m)$-bit words and unit-cost operations for all complexity results.

3. Main Results

In this section, we sketch the main results from [24] on structure-aware Datalog rewriting. The readers may refer to [24] for the formal algorithm and analysis. Our first result considers Datalog programs that are *rulewise-acyclic*. A rule is said to be acyclic if the body of the rule, viewed as a Conjunctive Query, admits a join tree. We say that a program is rulewise-acyclic if it has only acyclic rules.

Theorem 3.1 ([24])**.** Let P be a rulewise-acyclic Datalog program with input size m, active domain size n, and arity k. Then, we can evaluate P in time $O(n^{k-1} \cdot (m + n^k))$.

We give below an example of how our algorithm for the theorem works, using **even-hop backward reachability** in directed graphs:

$$\begin{aligned} &\mathsf{evenPath}(x) \leftarrow \mathsf{sink}(x). \\ &\mathsf{evenPath}(x) \leftarrow \mathsf{edge}(x, y), \mathsf{edge}(y, z), \mathsf{evenPath}(z). \end{aligned} \tag{1}$$

Both rules here are acyclic, and the arity is $k = 2$. Observe that producing directly all groundings of the second rule would generate a grounded program of size $O(m \cdot n)$. Our algorithm first rewrites the second rule following the structure of its join tree:

$$\begin{aligned} &\mathsf{evenPath}(x) \leftarrow \mathsf{sink}(x). \\ &\mathsf{oddPath}(y) \leftarrow \mathsf{edge}(y, z), \mathsf{evenPath}(z). \\ &\mathsf{evenPath}(x) \leftarrow \mathsf{edge}(x, y), \mathsf{oddPath}(y). \end{aligned} \tag{2}$$

Now grounding the resulting rules will produce a grounded program of only size $O(m)$, which implies an evaluation algorithm with runtime $O(m)$. In fact, a direct corollary of Theorem 3.1 is that for monadic acyclic Datalog we have a linear runtime of $O(m)$, which is essentially optimal.

Theorem 3.1 can be generalized from acyclic rules to rules bounded by submodular width.

Theorem 3.2 ([24])**.** Let P be a Datalog program where $\mathsf{arity}(P) \leq k$, subw is the maximum submodular width across all rules of P, and suppose the input size is m, and the active domain size is n. Then, we can evaluate P in time $\widetilde{O}(n^{k-1} \cdot (m^{\mathsf{subw}} + n^{k \cdot \mathsf{subw}}))$, where $\widetilde{O}$ hides a polylogarithmic factor.

As an example, consider the following Datalog program, which computes a **diamond-pattern reachability** starting from some node set $\mathsf{source}(x)$:

$$\begin{aligned} &\mathsf{T}(x) \leftarrow \mathsf{source}(x). \\ &\mathsf{T}(x) \leftarrow \mathsf{T}(y), \mathsf{edge}(y, z), \mathsf{edge}(z, x), \mathsf{edge}(x, w), \mathsf{edge}(w, y). \end{aligned} \tag{3}$$

For this program, the submodular width of the second rule is $3/2$. Further, its arity is $k = 1$. Thus, we can evaluate the above program in time $\widetilde{O}(m^{3/2})$.

As a corollary of Theorem 3.2, monadic Datalog (i.e. when $k = 1$) can be evaluated in time $\widetilde{O}(m^{\mathsf{subw}})$. This is also the best-known runtime for Boolean Conjunctive Queries [16]. Hence, this result tells us that the addition of recursion with unary IDBs does not really add to the runtime of evaluation!

When the rules in the Datalog program are *linear*, we can obtain even stronger runtime results.

Theorem 3.3 ([24])**.** Let P be a linear and rulewise-acyclic Datalog program with input size m, active domain size n, and arity ≤ 2. Then, we can evaluate P in time $O(m \cdot n)$.

Theorem 3.4 ([24])**.** Let P be a linear Datalog program with input size m, active domain size n, arity $\leq k$, and submodular width subw. Then, we can evaluate P in time $O(n^{k-1} \cdot m^{\mathsf{subw}-1} \cdot (m + n^k))$.

4. Experiments

In this section, we demonstrate that our rewriting technique can result in significant runtime speed-ups in practice. We ran all our experiments on a bare-metal single-node virtual machine on CloudLab [25] that has two Intel E5-2683 v3 14-core CPUs and 256GB main memory.

We test our structure-aware rewritings via three programs on `Soufflé`. `Soufflé` [2] is a widely used high-performance Datalog system that is primarily designed for program analysis and uses advanced optimization techniques such as efficient program synthesis. All results reported are configured to run single-threaded in main memory. The three programs are: ❶ same generation [26, 27], i.e.

$$\begin{aligned} &\mathsf{sg}(x, y) \leftarrow \mathsf{edge}(p, x), \mathsf{edge}(p, y), x \neq y. \\ &\mathsf{sg}(x, y) \leftarrow \mathsf{edge}(a, x), \mathsf{sg}(a, b), \mathsf{edge}(b, y). \end{aligned} \tag{4}$$

and its rewriting (using our technique)

$$\begin{aligned} &\mathsf{sg}(x, y) \leftarrow \mathsf{edge}(p, x), \mathsf{edge}(p, y), x \neq y. \\ &\mathsf{fatherOf}(b, x) \leftarrow \mathsf{edge}(a, x), \mathsf{sg}(a, b). \\ &\mathsf{sg}(x, y) \leftarrow \mathsf{fatherOf}(b, x), \mathsf{edge}(b, y). \end{aligned} \tag{5}$$

Dataset	# nodes	# edges
G5k	5000	25046
G10k	10000	99805
wiki-Vote	7115	103689
ego-Twitter	81306	1768149
web-Stanford	281903	2312497
soc-Livejournal	4848571	68993773

Table 1
Number of nodes and edges of all graph datasets used in the experiments

Programs	Dataset	Runtime	Runtime after rewriting
same generation	G5k	125s	**54s**
	G10k	2078s	**372s**
	wiki-Vote	328s	**32s**
two-hop backward reachability	ego-Twitter	8.3s	**3.2s**
	web-Stanford	68.8s	**3.5s**
	soc-Livejournal	435s	**136s**
diamond-pattern reachability	wiki-Vote	7s	**1.5s**
	ego-Twitter	145s	**73s**
	web-Stanford	**11s**	52s

Table 2
Runtime results of the original programs and the rewritten programs

❷ even-hop backward reachability and its rewriting (both outlined in Section 3), and ❸ diamond-pattern reachability (3) (it is not rulewise-acyclic) and its rulewise-acyclic rewriting as follows:

$$\begin{aligned}
&\mathsf{T}(x) \leftarrow \mathsf{source}(x).\\
&\mathsf{Z}(x,y) \leftarrow \mathsf{T}(y), \mathsf{edge}(y,z), \mathsf{edge}(z,x).\\
&\mathsf{W}(x,y) \leftarrow \mathsf{T}(y), \mathsf{edge}(x,w), \mathsf{edge}(w,y).\\
&\mathsf{T}(x) \leftarrow \mathsf{Z}(x,y), \mathsf{W}(x,y).
\end{aligned} \tag{6}$$

We use graph datasets (wiki-Vote, ego-Twitter, web-Stanford and soc-Livejournal) from the Stanford Large Network Dataset Collection [28] as program inputs. The G5k and G10k datasets are random graphs of $5k$ and $10k$ nodes sampled from the Erdős-Rényi model [29], where each edge exists with probability 0.001. All datasets are listed in Table 1. Whenever necessary, we randomly sample 100 nodes from the graph and designate them as the source nodes $\mathsf{source}(x)$ or the sink nodes $\mathsf{sink}(x)$.

Table 2 summarizes the runtime results on Soufflé. We find a consistent 2.5 up to ~20x speed-up on the first two programs. In the last row, one regression (~5x slowdown) is observed on web-Stanford because the final result set is small but the introduced $\mathsf{Z}(x,y), \mathsf{W}(x,y)$ IDBs are very large and hence expensive to materialize in the rewritten Soufflé program (6).

5. Conclusion

We propose a structure-aware rewriting technique that allows us to obtain and even refine the best-known theoretical runtime results. The theoretical results are based on our recent PODS 2024 publication [24]. In addition, our rewriting exhibits significant speed-ups on Soufflé across multiple recursive graph workloads. Future directions include an in-depth understanding of the tightness of our results and a more comprehensive investigation of the practicality of our technique in modern Datalog engines like Soufflé.

References

[1] J. Seo, J. Park, J. Shin, M. S. Lam, Distributed socialite: A datalog-based language for large-scale graph analysis, Proc. VLDB Endow. 6 (2013) 1906–1917. URL: http://www.vldb.org/pvldb/vol6/p1906-seo.pdf. doi:10.14778/2556549.2556572.

[2] B. Scholz, H. Jordan, P. Subotic, T. Westmann, On fast large-scale program analysis in datalog, in: CC, ACM, 2016, pp. 196–206.

[3] Y. Smaragdakis, G. Balatsouras, Pointer analysis, Found. Trends Program. Lang. 2 (2015) 1–69.

[4] T. J. Green, S. S. Huang, B. T. Loo, W. Zhou, Datalog and recursive query processing, Found. Trends Databases 5 (2013) 105–195.

[5] T. W. Reps, Program analysis via graph reachability, Inf. Softw. Technol. 40 (1998) 701–726.

[6] Y. Li, Q. Zhang, T. Reps, Fast graph simplification for interleaved dyck-reachability, in: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020, pp. 780–793.

[7] Y. Li, Q. Zhang, T. Reps, On the complexity of bidirected interleaved dyck-reachability, Proceedings of the ACM on Programming Languages 5 (2021) 1–28.

[8] A. A. Mathiasen, A. Pavlogiannis, The fine-grained and parallel complexity of andersen's pointer analysis, Proc. ACM Program. Lang. 5 (2021) 1–29.

[9] L. O. Andersen, Program analysis and specialization for the C programming language, Ph.D. thesis, Citeseer, 1994.

[10] M. R. Garey, Computers and intractability: A guide to the theory of np-completeness, freeman, Fundamental (1997).

[11] S. Cosmadakis, Inherent complexity of recursive queries, in: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 1999, pp. 148–154.

[12] J. D. Ullman, A. V. Gelder, Parallel complexity of logical query programs, Algorithmica 3 (1988) 5–42.

[13] F. Afrati, C. H. Papadimitriou, The parallel complexity of simple logic programs, Journal of the ACM (JACM) 40 (1993) 891–916.

[14] M. Joglekar, C. Ré, It's all a matter of degree - using degree information to optimize multiway joins, Theory Comput. Syst. 62 (2018) 810–853.

[15] H. Q. Ngo, E. Porat, C. Ré, A. Rudra, Worst-case optimal join algorithms, J. ACM 65 (2018) 16:1–16:40.

[16] M. A. Khamis, H. Q. Ngo, D. Suciu, What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another?, in: PODS, ACM, 2017, pp. 429–444.

[17] N. Carmeli, M. Kröll, On the enumeration complexity of unions of conjunctive queries, ACM Transactions on Database Systems (TODS) 46 (2021) 1–41.

[18] Y. A. Liu, S. D. Stoller, From datalog rules to efficient programs with time and space guarantees, in: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming, PPDP '03, Association for Computing Machinery, New York, NY, USA, 2003, p. 172–183.

[19] K. T. Tekle, Y. A. Liu, Precise complexity analysis for efficient datalog queries, in: Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, PPDP '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 35–44. URL: https://doi.org/10.1145/1836089.1836094. doi:10.1145/1836089.1836094.

[20] K. T. Tekle, Y. A. Liu, Precise complexity guarantees for pointer analysis via datalog with extensions, Theory Pract. Log. Program. 16 (2016) 916–932.

[21] K. T. Tekle, Y. A. Liu, More efficient datalog queries: subsumptive tabling beats magic sets, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 661–672.

[22] M. Yannakakis, Graph-theoretic methods in database theory, in: PODS, ACM Press, 1990, pp. 230–242.

[23] K. Casel, M. L. Schmid, Fine-grained complexity of regular path queries, in: ICDT, volume 186 of

LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 19:1–19:20.
[24] H. Zhao, S. Deep, P. Koutris, S. Roy, V. Tannen, Evaluating datalog over semirings: A grounding-based approach, Proc. ACM Manag. Data 2 (2024) 90.
[25] Cloudlab, https://www.cloudlab.us/, 2018.
[26] F. Bancilhon, D. Maier, Y. Sagiv, J. D. Ullman, Magic sets and other strange ways to implement logic programs, in: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems, 1985, pp. 1–15.
[27] Z. Fan, S. Mallireddy, P. Koutris, Towards better understanding of the performance and design of datalog systems., Datalog 2 (2022) 166–180.
[28] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection, http://snap.stanford.edu/data, 2014.
[29] D. A. Bader, K. Madduri, Gtgraph: A synthetic graph generator suite (2006).