The Magic of Pushing Extrema into Recursion: Simple and Powerful Datalog Programs

Carlo Zaniolo, Mohan Yang, Ariyam Das, and Matteo Interlandi

University of California, Los Angeles
{zaniolo, yang, ariyam, minterlandi}@cs.ucla.edu

Abstract. We introduce a novel query optimization method based on pushing extrema and other integrity constraints into recursion. This optimization produces an efficient operational semantics for the goals declaratively specified by the original program that is stratified with respect to negation and aggregates. Complex algorithms can now be specified via simpler programs, and then implemented via optimized rules that use min and max aggregates in the seminaive fixpoint computation to achieve safe and efficient execution. This optimization dovetails with recent advances that entail the use of monotonic aggregates in recursion and leads to efficient implementation, as demonstrated by the scalable Datalog systems we have recently developed.

Introduction

A growing body of research on scalable data analytics has brought a renaissance of interest in Datalog due to its ability to declaratively specify advanced applications that execute efficiently over different architectures, including massively parallel ones [6–11]. As part of this renaissance, a solution [3, 4] was recently proposed for the difficult problem of using aggregates in recursion which had remained open for more than 20 years; the new solution enables the concise expression and efficient support of graph algorithms and data mining methods. In this short paper, we discuss an even more recent discovery that makes possible a further simplification of these recursive programs, which is described next.

Rather than writing programs with monotonic min and max in recursion, it is often easier to write stratified programs where the recursive rules use no min or max, which are instead used in the next stratum to define the correct logical result. Indeed, in this paper we show that it is often simple for the compiler to optimize the computation of those programs by moving the extrema into the actual rules used in the fixpoint iteration. This optimization by constraint-pushing approach produces a simple declarative semantics, combined with a very efficient operational semantics—a combination that makes it easy to express concisely and implement efficiently complex algorithms, as illustrated by Example 1 below.

The syntax `min⟨Dy⟩` in our example illustrates the head notation for aggregates that is used in many Datalog systems, and follows SQL-2 approach of allowing zero, one or more group-by variables for each aggregate.

Example 1. Shortest path from node a.

$$
\begin{array}{ll}
(\mathtt{r}_1) & \mathtt{path(Y, Dy)} \leftarrow \quad \mathtt{arc(a, Y, Dy)}. \\
(\mathtt{r}_2) & \mathtt{path(Y, Dy)} \leftarrow \quad \mathtt{path(X, Dx), arc(X, Y, Dxy), Dy = Dx + Dxy}. \\
(\mathtt{r}_3) & \mathtt{spath(Y, min\langle Dy\rangle)} \leftarrow \mathtt{path(Y, Dy)}.
\end{array}
$$

Thus in our example, Dy is the aggregate variable and Y is the group-by variable. We will often refer to the min and max variables as *cost variables.* Now, the semantics of extrema (i.e., min and max) can be easily reduced to that of negation, whereby the last rule in Example 1 can be replaced by:

Example 2. Min via negation in stratified programs.

$$
\begin{array}{ll}
\mathtt{spath(Y, Dy)} \leftarrow & \mathtt{path(Y, Dy)}, \neg\mathtt{betterpath(Y, Dy)}. \\
\mathtt{betterpath(Y, Dy)} \leftarrow & \mathtt{path(Y, Dyy)}, \mathtt{Dyy < Dy}.
\end{array}
$$

Re-expressing our min via negation also makes manifest the non-monotonic nature of extrema aggregates, whereby their usage in recursive predicates is incompatible with the declarative least-fixpoint semantics of the programs—a topic that dovetails with the solution proposed in this paper and is discussed in [13].

Stratification can be used to avoid the semantic problems caused by using aggregates in recursion. For instance, in Example 1, spath belongs to a stratum that is above that of path, whereby our program is assured to have a *perfect-model semantics* [5]. The perfect model of a stratified program is unique and can be computed using an *iterated fixpoint computation*, whereby the least fixpoint is computed starting at the bottom stratum and moving up to higher strata. In our example, therefore, all the possible paths will be computed using rules $\mathtt{r}_1$ and $\mathtt{r}_2$, before selecting values that are minimal using $\mathtt{r}_3$. This is the approach used by current Datalog compilers, and it can be very inefficient or even non-terminating when the original graph contains cycles. In this paper, we will show that, for large classes of programs, the computation can be significantly optimized by simply pushing constraints into the fixpoint computation. In Example 1 above, the constraint is imposed by the last rule that, for each node produced, selects the minimal value of its distance from a. This non-monotonic constraint can be pushed into the recursive rules whereby our program becomes:

Example 3. Optimized shortest path from node a.

$$
\begin{array}{ll}
\mathtt{path(Y, min\langle Dy\rangle)} \leftarrow & \mathtt{arc(a, Y, Dy)}. \\
\mathtt{path(Y, min\langle Dy\rangle)} \leftarrow & \mathtt{path(X, Dx), arc(X, Y, Dxy), Dy = Dx + Dxy}. \\
\mathtt{spath(Y, Dy)} \leftarrow & \mathtt{path(Y, Dy)}.
\end{array}
$$

Because of the use of the non-monotonic constructs in its recursive rules, the program so obtained does not have a declarative semantics. However, the Immediate Consequence Operator (ICO) can still be defined for these rules, and so is *fixpoint iteration* $T_P^{\uparrow\omega}(\emptyset)$, that defines the operational semantics of these rules. It can be shown that this operational semantics produces the same spath values as those in the perfect model of the original program [13]. Thus this optimized computation will be used for spath instead of the iterated-fixpoint computations of perfect models normally used for stratified programs.

Moreover this optimization can be applied to large classes of programs where the rules have the simple properties described in [13]. Two simple examples of such programs are described next. In Example 4, we compute the connected components of an undirected graph via label propagation. The final label of a node Z is the minimum node id among all the nodes that can reach Z. All nodes with the same label belong to the same component.

Example 4. Connected components of an undirected graph.

$$\begin{aligned}
&\texttt{reach}(\texttt{X},\texttt{X}) \leftarrow \quad \texttt{arc}(\texttt{X},_).\\
&\texttt{reach}(\texttt{X},\texttt{Z}) \leftarrow \quad \texttt{reach}(\texttt{X},\texttt{Y}),\texttt{arc}(\texttt{Y},\texttt{Z}).\\
&\texttt{cc}(\texttt{Z},\texttt{min}\langle\texttt{X}\rangle) \leftarrow \texttt{reach}(\texttt{X},\texttt{Z}).
\end{aligned}$$

The pushing of the min constraint produces the following optimized program.

Example 5. HCC algorithm [2] in Datalog.

$$\begin{aligned}
&\texttt{reach}(\texttt{min}\langle\texttt{X}\rangle,\texttt{X}) \leftarrow \texttt{arc}(\texttt{X},_).\\
&\texttt{reach}(\texttt{min}\langle\texttt{X}\rangle,\texttt{Z}) \leftarrow \texttt{reach}(\texttt{X},\texttt{Y}),\texttt{arc}(\texttt{Y},\texttt{Z}).\\
&\texttt{cc}(\texttt{Z},\texttt{X}) \leftarrow \texttt{reach}(\texttt{X},\texttt{Z}).
\end{aligned}$$

Conjunction of Inequalities and Extrema Constraints. Let us now consider a constraint involving a conjunction of a min with an inequality constraint requiring the minimized value to not exceed a given K value.

Example 6. Shortest path with min and upperbound constraints.

$$\begin{aligned}
&\texttt{apath}(\texttt{Y},\texttt{Dy}) \leftarrow \texttt{arc}(\texttt{a},\texttt{Y},\texttt{Dy}).\\
&\texttt{apath}(\texttt{Y},\texttt{Dy}) \leftarrow \texttt{apath}(\texttt{X},\texttt{Dx}),\texttt{arc}(\texttt{X},\texttt{Y},\texttt{Dxy}),\texttt{Dy} = \texttt{Dx} + \texttt{Dxy},\texttt{Dy} > \texttt{Dx}.\\
&\texttt{minboundpath}(\texttt{Y},\texttt{min}\langle\texttt{Dy}\rangle) \leftarrow \texttt{apath}(\texttt{Y},\texttt{Dy}),\texttt{aconstant}(\texttt{K}),\texttt{Dy} < \texttt{K}.
\end{aligned}$$

Then we have the following *sound* and *complete* rewriting w.r.t. minboundpath:

Example 7. Example 6 optimized by both inequality and min.

$$\begin{aligned}
&\texttt{apath}(\texttt{Y},\texttt{min}\langle\texttt{Dy}\rangle) \leftarrow \quad \texttt{arc}(\texttt{a},\texttt{Y},\texttt{Dy}),\texttt{aconstant}(\texttt{K}),\texttt{Dy} < \texttt{K}.\\
&\texttt{apath}(\texttt{Y},\texttt{min}\langle\texttt{Dy}\rangle) \leftarrow \quad \texttt{apath}(\texttt{X},\texttt{Dx}),\texttt{arc}(\texttt{X},\texttt{Y},\texttt{Dxy}),\texttt{Dy} = \texttt{Dx} + \texttt{Dxy},\texttt{Dy} > \texttt{Dx},\\
&\qquad\qquad\qquad\qquad\quad \texttt{aconstant}(\texttt{K}),\texttt{Dy} < \texttt{K}.\\
&\texttt{minboundpath}(\texttt{Y},\texttt{Dy}) \leftarrow \texttt{apath}(\texttt{Y},\texttt{Dy}).
\end{aligned}$$

Correctness of Constraint Pushing into Recursion (CPR)

A CPR-structured segment of a stratified program P, consists of (i) rules defining one or more recursive predicates, and (ii) a constraint rule specifying one or more constraints on the recursive predicates defined by (i). If we let γ denote the constraints defined in (ii), then the CPR optimization consists in removing the constraints from (ii) to add them to the rules in (i) defining the corresponding recursive predicates. Then γ is applied to the rules or facts defining the recursive predicates, and the old constraint rule becomes a *copy rule* that just renames the atoms produced by the fixpoint iteration.

If T_ρ is the ICO for the rules and facts defining the recursive predicates, then the ICO for those rules and facts after they undergo the CPR optimization will be denoted by $T_{\gamma\rho}$: $T_{\gamma\rho}(I) = \gamma(T_\rho(I))$.

Observe that the results produced by the fixpoint iteration on $T_{\gamma\rho}$ are *sound* since they are a subset of those produced by the iterated fixpoint on the original program. However, completeness, is also needed to assure the correctness of the CPR optimization. Our strict requirement for completeness is that the following property must hold for every positive integer n: $\gamma\big(T_\rho^{\uparrow n}(\emptyset)\big) = T_{\gamma\rho}^{\uparrow n}(\emptyset)$. When this property holds for each n, then $\gamma\big(T_\rho^{\uparrow \omega}(\emptyset)\big) = T_{\gamma\rho}^{\uparrow \omega}(\emptyset)$ also holds.

Simple sufficient correctness conditions that assure correctness for most programs of interest are given in [13]. For instance, the recursive rule in Example 6 is such that when both $\texttt{apath}(\texttt{X}, \texttt{Dx})$ and $\texttt{apath}(\texttt{X}, \texttt{Dx}')$ are true and $\texttt{Dx}' < \texttt{Dx}$, then if the rule holds for the former, it must also hold for the latter making the former expendable in the computation of `min`. However if we change $\texttt{Dy} < \texttt{K}$ into $\texttt{Dy} > \texttt{K}$ the completeness of CPR might no longer hold. By using such criteria the DeALS compiler [1] is able to determine that the CPR optimization is correct for all examples in this paper [13].

Conclusion

The ability to specify complex algorithms by adding postconditions to simpler recursive predicates is of interest in the specification of powerful algorithms and their efficient implementation. In fact, as discussed in [13], the CPR approach tested on the sequential version of DeALS produces executions that are incomparably faster than the iterated fixpoint and two or three times faster than their more complex specification as XY-stratified programs [12]. We are now working on applying these ideas to achieve efficient and scalable support of graph and data mining applications over diverse parallel platforms, as needed to demonstrate the major benefits that our Big Datalog project is achieving through a close integration of theoretical and system advances [7].

Acknowledgments

We thank all the reviewers for their comments on improving the paper. This work was supported in part by NSF grants IIS-1218471 and IIS-1302698.

References

1. Deductive Application Language System. `http://wis.cs.ucla.edu/deals/`.
2. U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *ICDM*, pages 229–238. IEEE, 2009.
3. M. Mazuran, E. Serra, and C. Zaniolo. A declarative extension of horn clauses, and its significance for datalog and its applications. *TPLP*, 13(4-5):609–623, 2013.
4. M. Mazuran, E. Serra, and C. Zaniolo. Extending the power of datalog recursion. *The VLDB Journal*, 22(4):471–493, 2013.
5. T. C. Przymusinski. Perfect model semantics. In *ICLP/SLP*, pages 1081–1096, 1988.
6. J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: a datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013.
7. A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big Data Analytics with Datalog Queries on Spark. In *SIGMOD*. ACM, 2016.
8. A. Shkapsky, K. Zeng, and C. Zaniolo. Graph queries in a next-generation datalog system. *PVLDB*, 6(12):1258–1261, 2013.
9. J. Wang, M. Balazinska, and D. Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12), 2015.
10. M. Yang, A. Shkapsky, and C. Zaniolo. Parallel bottom-up evaluation of logic programs: DeALS on shared-memory multicore machines. In *Technical Communications of ICLP*, 2015.
11. M. Yang and C. Zaniolo. Main memory evaluation of recursive queries on multicore machines. In *IEEE Big Data*, pages 251–260, 2014.
12. C. Zaniolo, S. Ceri, C. Faloutsos, R. T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced database systems*. Morgan Kaufmann Publishers Inc., 1997.
13. C. Zaniolo, M. Yang, A. Das, and M. Interlandi. Improving the power, performance and usability of datalog by pushing constraints into recursion. Submitted for publication, 2016.