TLP **17** (5–6): 1048–1065, 2017. © Cambridge University Press 2017
doi:10.1017/S1471068417000436 First published online 23 August 2017

*Fixpoint semantics and optimization of recursive Datalog programs with aggregates**

CARLO ZANIOLO, MOHAN YANG, ARIYAM DAS,
ALEXANDER SHKAPSKY and TYSON CONDIE
University of California, Los Angeles, Los Angeles, CA, USA
(*e-mails:* zaniolo@cs.ucla.edu, yang@cs.ucla.edu, ariyam@cs.ucla.edu,
shkapsky@cs.ucla.edu, tcondie@cs.ucla.edu)

MATTEO INTERLANDI†
Microsoft, Redmond, WA, USA
(*e-mail:* matteo.interlandi@microsoft.com)

submitted 20 July 2017; accepted 27 July 2017

Abstract

A very desirable Datalog extension investigated by many researchers in the last 30 years consists in allowing the use of the basic SQL aggregates **min**, **max**, **count** and **sum** in recursive rules. In this paper, we propose a simple comprehensive solution that extends the declarative least-fixpoint semantics of Horn Clauses, along with the optimization techniques used in the bottom-up implementation approach adopted by many Datalog systems. We start by identifying a large class of programs of great practical interest in which the use of **min** or **max** in recursive rules does not compromise the declarative fixpoint semantics of the programs using those rules. Then, we revisit the monotonic versions of **count** and **sum** aggregates proposed by Mazuran *et al.* (2013b, *The VLDB Journal 22*, 4, 471–493) and named, respectively, **mcount** and **msum**. Since mcount, and also msum on positive numbers, are monotonic in the lattice of set-containment, they preserve the fixpoint semantics of Horn Clauses. However, in many applications of practical interest, their use can lead to inefficiencies, that can be eliminated by combining them with max, whereby mcount and msum become the standard count and sum. Therefore, the semantics and optimization techniques of Datalog are extended to recursive programs with min, max, count and sum, making possible the advanced applications of superior performance and scalability demonstrated by BigDatalog (Shkapsky *et al.* 2016. In *SIGMOD*. ACM, 1135–1149) and Datalog-MC (Yang *et al.* 2017. *The VLDB Journal 26*, 2, 229–248).

KEYWORDS: Datalog, Constraints, Recursion, Aggregates.

* This work was supported in part by NSF grants IIS-1218471, IIS-1302698 and CNS-1351047 and U54EB020404 awarded by NIH Big Data to Knowledge (BD2K).
† Work done while at UCLA.

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

1 Introduction

A growing body of research on scalable data analytics has brought a renaissance of interest in Datalog because of its ability to specify, declaratively, advanced data-intensive applications that execute efficiently over different systems and architectures, including massively parallel ones (Seo *et al.* 2013; Shkapsky *et al.* 2013; Yang and Zaniolo 2014; Aref *et al.* 2015; Wang *et al.* 2015; Yang *et al.* 2015; Shkapsky *et al.* 2016; Yang *et al.* 2017). A common thread in this new generation of Datalog systems is the use of aggregates in recursion, since aggregates enable the concise expression and efficient support of much more powerful algorithms than those expressible by programs that are stratified w.r.t. negation and aggregates. However, the non-monotonic nature of common aggregates creates major issues, and extending the declarative semantics of Datalog to allow aggregates in recursion represents a difficult problem since the early days of Datalog (Kemp and Stuckey 1991; Greco *et al.* 1992; Ross and Sagiv 1992). These approaches were seeking to achieve both (i) a formal declarative semantics for deterministic queries using the basic SQL aggregates, min, max, count and sum, in recursion and (ii) their efficient implementation by extending techniques of the early Datalog systems (Morris *et al.* 1986; Chimenti *et al.* 1987; Ramakrishnan *et al.* 1992; Vaghani *et al.* 1994; Arni *et al.* 2003). As discussed in the related work section, while many of these approaches only dealt with some of the four basic aggregates, the proposal presented by Ross and Sagiv (1992) covered all four. However, it used different lattices for different aggregates and thus incurred in the problems described by Van Gelder (1993). Therefore, the search for general solutions has motivated several approaches that have addressed the problem using powerful semantics, such as answer-set semantics, that however require higher levels of computational complexity (Pelov *et al.* 2007; Son and Pontelli 2007; Swift and Warren 2010; Faber *et al.* 2011; Gelfond and Zhang 2014).

Recently, the explosion of activity on Big Data has produced a revival of interest in Datalog as a parallelizable language for expressing and supporting efficiently Big Data Analytics (Seo *et al.* 2013; Shkapsky *et al.* 2013; Wang *et al.* 2015). The algorithms discussed in those papers are graph algorithms or other algorithms that use aggregates in recursion for Big Data applications where performance is of the essence. Thus, a convergence of declarative semantics and efficient implementation is critically needed and has motivated the research presented in this paper, where we propose a novel approach based on the concept of *pre-mappability of constraints*. Indeed, pre-mappability of constraints provides a simple criterion that (i) the system optimizer can utilize to push constraints into recursion, and (ii) the user can utilize to write programs that use aggregates in recursion, with the guarantee that they have indeed a formal fixpoint semantics.

The rest of this paper is organized as follows. In the next section, we introduce the problem of pushing constraints into recursion; then, in Section 3, we present the rather surprising result that in many programs of practical interest constraints applied to recursive rules do not invalidate the formal fixpoint semantics of those rules. We thus introduce general sufficient conditions under which we can push comparison constraints, extrema constraints and combinations of the two. With the

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

treatment of min and max in recursive rules completed, we move to the problem of supporting count and sum. In Section 5, we present an in-depth treatment of this important subject by combining the monotonic versions of these aggregates proposed by Mazuran *et al.* (2013a) with the treatment of extrema proposed in the previous sections. Seminaive evaluation in the presence of constraint pushing is discussed in Section 6. Related work and conclusions presented in Sections 7 and 8 bring the paper to a closing.

2 Comparison and extrema constraints

We will now propose motivating examples that also illustrate the Datalog syntax and terminology we use, which are widely used in the literature (Zaniolo *et al.* 1997; Shkapsky *et al.* 2015).

Example 1 (*Limited length paths from node a in a directed graph*)

(r_1) $\texttt{path(Y, Dy)} \leftarrow \texttt{arc(a, Y, Dy)}, \texttt{Dy} \geqslant 0.$
(r_2) $\texttt{path(Y, Dy)} \leftarrow \texttt{path(X, Dx)}, \texttt{arc(X, Y, Dxy)}, \texttt{Dxy} \geqslant 0, \texttt{Dy} = \texttt{Dx} + \texttt{Dxy}.$
(r_3) $\texttt{llpath(Y, Dy)} \leftarrow \texttt{path(Y, Dy)}, \texttt{Dy} < 143.$

In the program of Example 1, $\texttt{path(Y, Dy)}$ is the *head* of r_1, while $\texttt{arc(a, Y, Dy)}$ and $\texttt{Dy} \geqslant 0$ are the rule's *goals*, where $\texttt{Dy} \geqslant 0$ specifies a *lower-bound constraint* (symmetrically, $\texttt{Dy} < 143$ in r_3 is an *upper-bound constraint*). In rule r_2, the two-argument path is both the head and a goal: thus r_2 is a *recursive rule* and path is a recursive predicate. On the other hand, non-recursive rule r_1 with path as its head will be called an *exit* rule. The arithmetic predicates used in r_2 can be defined using Horn Clauses on Datalog$_{1s}$ (Zaniolo *et al.* 1997), and therefore our program has formal semantics based on the equivalence of its model-theoretic, proof-theoretic, and least-fixpoint semantics (van Emden and Kowalski 1976). The least-fixpoint of a monotonic program can be computed by iterating over the *Immediate Consequence Operator* (ICO) defined by the rules of the program, and Datalog researchers have introduced techniques such as seminaive-fixpoint and magic-set to optimize this bottom-up iterative computation into efficient implementations (Zaniolo *et al.* 1997). Even more significant optimization opportunities occur in the presence of constraints. For instance, an unoptimized Datalog computation for this program would apply the upper-bound constraint $\texttt{Dy} < 143$ specified by the final rule r_3 to all the path atoms generated by iterating over rules r_1 and r_2. Thus, this constraint is applied as a post-condition after all pairs have been generated, and does not improve the efficiency of the computation; in fact, this might never terminate when our graph contains cycles.

A very simple solution for this problem is to push the upper-bound constraint $\texttt{Dy} < 143$ into the recursive computation. We can hence rewrite the above program into the following, where the constraint has been transferred to the rules defining path.

Example 2 (*Rewritten version of Example 1 where the constraint is transferred*)

(r'_1) $\texttt{path(Y, Dy)} \leftarrow \texttt{arc(a, Y, Dy)}, \texttt{Dy} \geqslant 0, \texttt{Dy} < 143.$
(r'_2) $\texttt{path(Y, Dy)} \leftarrow \texttt{path(X, Dx)}, \texttt{arc(X, Y, Dxy)}, \texttt{Dxy} \geqslant 0, \texttt{Dy} = \texttt{Dx} + \texttt{Dxy}, \texttt{Dy} < 143.$
(r'_3) $\texttt{llpath(Y, Dy)} \leftarrow \texttt{path(Y, Dy)}.$

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

Thus, we obtain a program where the constraint $\texttt{Dy} < 143$ has been *transferred from the final rule* $\texttt{r}_3$ into the recursive program and applied to rules $\texttt{r}_2$ and $\texttt{r}_1$, i.e., *pre-mapped* to those rules. Observe that the rules produced by this transformation have been renamed $\texttt{r}'_3$, $\texttt{r}'_2$ and $\texttt{r}'_1$. The program so obtained provides an optimized version of the original program that is equivalent to it inasmuch as the two programs deliver the same `llpath` results. This equivalence follows from the fact that lengths of `arc` are $\geqslant 0$, whereby $\texttt{Dy} < 143$ in the body of $\texttt{r}'_2$ implies that $\texttt{Dy} < 143$ for `Dy` in the head of the same rule—i.e., pruned values are those that would be eliminated by the final "< 143" constraint in the original $\texttt{r}_3$ rule.

An even more significant optimization is possible when *extrema constraints* (i.e., max and min aggregates) are applied to the results produced by recursive rules. For instance, say that we substitute rule $\texttt{r}_3$ in Example 1 with the following rule that uses the extrema constraint `is_min`.

$$(\texttt{r}_4) \quad \texttt{spath(Y, Dy)} \leftarrow \texttt{path(Y, Dy), is_min((Y), (Dy))}.$$

The construct `is_min` identifies `Dy` as the *cost argument* and `Y` as the *group-by variable(s)*, using SQL terminology. Thus, `is_min((Y), (Dy))` denotes that we want the min values of `Dy` for each unique value of `Y`. The special notation used for min warns the compiler that its specialized internal implementation should be used in its evaluation, along with the query transformation and optimization techniques discussed next. However, the semantics of rules with our `is_min` construct can be reduced to that of rules with standard (closed-world) negation, whereby the semantics of $\texttt{r}_4$ is defined by the following two rules, where `is_min` is replaced by the negation that a lesser value exists[1]:

$$\texttt{spath(Y, Dy)} \leftarrow \texttt{path(Y, Dy)}, \neg\texttt{lesser(Y, Dy)}.$$
$$\texttt{lesser(Y, Dy)} \leftarrow \texttt{path(Y, Dy), path(Y, Dy1)}, \texttt{Dy1} < \texttt{Dy}.$$

Expressing `is_min` via negation reveals the non-monotonic nature of extrema constraints, whereby their usage in the definition of recursive predicates can compromise the declarative fixpoint semantics of the program. *Stratification*, where each predicate in the programs must belong to strata that are above those of their negated goals, can be used to avoid the semantic problems caused by using aggregates in recursion. For instance, in our example `spath` belongs to a stratum that is above that of `path`, whereby our program is assured to have a *perfect-model semantics* (Przymusinski 1988). The perfect model of a stratified program is a minimal model which is computed using an *iterated fixpoint computation*, starting at the bottom stratum and moving up to higher strata. In our example, therefore, all the possible paths will be computed using rules $\texttt{r}_1$ and $\texttt{r}_2$, before selecting values that are minimal using $\texttt{r}_4$. This approach, used by current Datalog compilers, can be very inefficient or even non-terminating when the original graph of Example 1 contains cycles. The naive re-writing similar to the one used to transfer the comparison constraint into our recursive rules yields

[1] This rewriting assumes that there is only one `is_min` in our program. In the presence of multiple occurrences, we will need to add a subscript to keep them distinct.

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

Example 3 (*Shortest path from node* a)

(r_1'') path(Y, Dy) ← arc(a, Y, Dy), Dy ⩾ 0, is_min((Y), (Dy)).
(r_2'') path(Y, Dy) ← path(X, Dx), arc(X, Y, Dxy), Dxy⩾0, Dy=Dx+Dxy, is_min((Y), (Dy)).
(r_4'') spath(Y, Dy) ← path(Y, Dy).

However, while in Example 1 the transferring of comparisons into the recursive program produces the positive program of Example 2 which has a well-defined formal semantics, the program in Example 3 above uses non-monotonic aggregates (alias negation if we replace them with their definitions) in recursive rules. Thus, there is no guarantee that a formal semantics exists for the rewritten program, and the question of whether this is a correct optimization becomes meaningless. Thus in this paper, we provide a formal minimal-fixpoint semantics that holds for Example 3, and for large classes of programs of major practical interest (Condie *et al.* 2017).

3 Fixpoint and constraints

The declarative and constructive semantics of a Datalog program P is defined in terms of the ICO for P, denoted by $T_P(I)$, where I is any Herbrand interpretation of P. For positive Datalog programs, i.e., those without negation or aggregates, $T_P(I)$ is a monotonic continuous mapping in the lattice of set-containment to which the interpretation I belongs, whereby we have the following well-known properties:

(1) A unique minimal (w.r.t. set-containment) solution of the equation $I = T_P(I)$ always exists and it is known as the *least-fixpoint* of T_P, denoted $lfp(T_P)$; $lfp(T_P)$ *defines the formal declarative semantics of* P.
(2) For an ICO T, with ω being the first infinite ordinal, $T^{\uparrow\omega}(\emptyset)$ is defined by letting $T^{\uparrow 0}(\emptyset) = \emptyset$, and $T^{\uparrow n+1}(\emptyset) = T(T^{\uparrow n}(\emptyset))$. Then, $T^{\uparrow\omega}(\emptyset)$ denotes the union of $T^{\uparrow n}(\emptyset)$ for every n. The *fixpoint iteration* $T_P^{\uparrow\omega}(\emptyset)$ defines the *operational semantics* of our program. In most applications of practical interest, the iteration converges to the final value in a finite number of steps, and it can be stopped at the first integer $n+1$ where, $T_P^{\uparrow n+1}(\emptyset) = T_P^{\uparrow n}(\emptyset)$.

For positive Datalog programs, we have that the operational and declarative semantics coincide.

In the previous section, we have discussed extrema and lower/upper-bound constraints which can be expressed and enforced upon rules by the addition of goals to the rules. We will now define conditions under which post-constraints on the results of recursive-rule computation can be pre-mapped and applied to those rules. Thus, we introduce the notion of a constraint γ being *pre-mappable* to one or more rules, in which case we will simply write that γ is $\mathscr{P}reM$ to those rules. The $\mathscr{P}reM$ property will allow us to extend the fixpoint semantics to programs obtained by transferring a *constraint* γ from the final rule to the rules defining a

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

recursive predicate[2]. Notions related to $\mathscr{P}reM$ are already used widely in optimizing distributed computations. For instance, when a set of tuples is partitioned into K subsets $S_1, \ldots, S_k$ we have that

$$\mathtt{OP}(\bigcup_{1 \leqslant j \leqslant K} S_j) = \mathtt{OP}(\bigcup_{1 \leqslant j \leqslant K} \mathtt{OP}(S_j))$$

Since this equality holds for OP denoting the aggregates sum, min and max, we will say that these aggregates are $\mathscr{P}reM$ to union. This property is frequently called pre-aggregation and exploited by the parallel execution of K OP aggregates on different nodes followed by a final execution of OP on the results returned by those K nodes. The above property also holds when OP denotes an upper-bound or a lower-bound constraint; however in that case, the final application of OP can be omitted given that upper/lower bound constraints distribute over union.

Definition 1 (*The $\mathscr{P}reM$ property*)
In a given Datalog program, let P be the set of its rules defining either a recursive predicate or a set of mutually recursive predicates. Also, let T be the ICO defined by P. Then, the constraint γ will be said to be $\mathscr{P}reM$ to T (and to P) when, for every interpretation I of P, we have that $\gamma(T(I)) = \gamma(T(\gamma(I)))$.

The importance of this property follows from the fact that if $I = T(I)$ is a fixpoint for T, then we also have that $\gamma(I) = \gamma(T(I))$, and when γ is $\mathscr{P}reM$ to the rules defining T we also have that

$$\gamma(I) = \gamma(T(I)) = \gamma(T(\gamma(I)))$$

Now, let T_γ denote the application of T followed by γ, i.e., $T_\gamma(I) = \gamma(T(I))$, where T_γ will be called the γ-*constrained* ICO for the P rules. Now, if I is a fixpoint for T and $I' = \gamma(I)$, then the above equality can be rewritten as

$$I' = \gamma(I) = \gamma(T(\gamma(I))) = T_\gamma(I')$$

Thus, when γ is $\mathscr{P}reM$, the fact that I is a fixpoint for T implies that $I' = \gamma(I)$ is a fixpoint for $T_\gamma(I)$. We will next focus on cases of practical interest where the transfer of constraints under $\mathscr{P}reM$ produces optimized programs that are safe and terminating even when the original programs were not, and prove that the transformation is indeed equivalence-preserving. Thus, we focus on situations where $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$, i.e., the fixpoint iteration converges after a finite number of steps n. The rules defining a recursive predicate p are those having as head p or predicates that are mutually recursive with p.

Theorem 1
In a Datalog program, let T be the ICO for the positive rules defining a recursive predicate. If the constraint γ is $\mathscr{P}reM$ to T, and a fixpoint exists such that

[2] Note that here and in the following definition, the γ symbol is overloaded because it is used to express a constraint both as a literal in a final rule, and a pruning condition over instances.

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

$T^{\uparrow n+1}(\emptyset) = T^{\uparrow n}(\emptyset)$, for some integer n then

$$\gamma(T^{\uparrow\omega}(\emptyset)) = T_\gamma^{\uparrow n}(\emptyset)$$

Proof
Let us assume that $T_\gamma^{\uparrow n+1}(\emptyset) = T_\gamma^{\uparrow n}(\emptyset)$ and prove by induction that $\gamma(T^{\uparrow k}(\emptyset)) = T_\gamma^{\uparrow k}(\emptyset)$ holds for every non-negative integer k. Indeed, the property holds for $k = 0$ since $\gamma(T^{\uparrow 0}(\emptyset)) = T_\gamma^{\uparrow 0}(\emptyset) = \emptyset$. Moreover, since γ is $\mathscr{P}reM$ to T, we have that the following equalities hold: $\gamma(T^{\uparrow k+1}(\emptyset)) = \gamma(T(T^{\uparrow k}(\emptyset))) = \gamma(T(\gamma(T^{\uparrow k}(\emptyset)))) = \gamma(T(T_\gamma^{\uparrow k}(\emptyset))) = T_\gamma^{\uparrow k+1}(\emptyset)$.

Now we have that $\gamma(T^{\uparrow\omega}(\emptyset)) = \gamma(\bigcup_{k\geqslant 0} T^{\uparrow k}(\emptyset))$. But $\bigcup_{k\geqslant 0} T^{\uparrow k}(\emptyset) = \bigcup_{m\geqslant n}(T^{\uparrow m}(\emptyset))$. In fact, the monotonicity of T implies that for each $T^{\uparrow m}(\emptyset)$ with $m \leqslant k$ there is a $T^{\uparrow m'}(\emptyset)$ with $m' > k$ containing it. Now, our γ constraints are also $\mathscr{P}reM$ to union, and thus we have that $\gamma(\bigcup_{m\geqslant n} T^{\uparrow m}(\emptyset)) = \gamma(\bigcup_{m\geqslant n} \gamma(T^{\uparrow m}(\emptyset))) = \gamma(\bigcup_{m\geqslant n}(T_\gamma^{\uparrow m}(\emptyset))) = \gamma(T_\gamma^{\uparrow n}(\emptyset))$ since, for $m \geqslant n$ all the $T_\gamma^{\uparrow m}(\emptyset)$ are identical. Finally, we have that $\gamma(T_\gamma^{\uparrow n}(\emptyset)) = \gamma(\gamma(T^{\uparrow n}(\emptyset))) = T_\gamma^{\uparrow n}(\emptyset)$ because applying twice the same constraint is equivalent to a single application. Thus, we conclude that $\gamma(T^{\uparrow\omega}(\emptyset)) = \gamma(T_\gamma^{\uparrow n}(\emptyset)) = T_\gamma^{\uparrow n}(\emptyset)$. □

We can next prove that the fixpoint $T_\gamma^{\uparrow n}(\emptyset)$ so obtained is in fact a minimal fixpoint for T_γ.

Theorem 2
If γ is $\mathscr{P}reM$ to T and $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset) = I$, then I is a minimal fixpoint for T_γ.

Proof
If $I' \subseteq T_\gamma^{\uparrow n}(\emptyset)$ is a fixpoint, then $I' = T_\gamma^{\uparrow k}(I')$ for every integer k. Now, starting with $T_\gamma^{\uparrow 0}(I') = I' = \gamma(T^{\uparrow 0}(I'))$, we can apply the inductive reasoning process used in Theorem 1 to conclude that the equality $T_\gamma^{\uparrow k}(I') = \gamma(T^{\uparrow k}(I'))$ holds for each k. Therefore the union, for all k, of the sets on the left side of the equation is equal to the union of all the sets on the right side. But the unioned sets on the left are equal to I', whereas all those on the right side define $\gamma(T^{\uparrow\omega}(\emptyset))$, since $\emptyset \subseteq I'$. But Theorem 1 has shown that $\gamma(T^{\uparrow\omega}(\emptyset)) = T_\gamma^{\uparrow n}(\emptyset) = I$. Thus, no proper subset of $\gamma(T^{\uparrow\omega}(\emptyset))$ can be a fixpoint for T_γ. □

Therefore, constraints which are $\mathscr{P}reM$ can be transferred into recursive rules and this represents a significant optimization which, in many cases, also produces programs where the fixpoint iteration terminates even when the iterated fixpoint computation on the original program did not. Moreover, the termination of the fixpoint iteration for the rewritten program ensures that we obtain a minimal fixpoint that is equivalent to the original program with respect to the predicates that were constrained by the $\mathscr{P}reM$ constraint γ. Thus, the well-known compilation techniques of positive programs, including magic set and seminaive computation, can be applied to programs optimized by the transfer of $\mathscr{P}reM$ constraints.

While the problem of determining general classes of $\mathscr{P}reM$ constraints leaves much room for further research, we have already determined classes of $\mathscr{P}reM$ constraints that hold for programs of great practical interest, such as those discussed next.

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

4 Transferring $\mathscr{P}reM$ constraints

We will next establish basic sufficient conditions under which transferability of extrema and comparison constraints hold for programs such as those in Examples 1 and 2. In those examples, we have a final rule (in the next stratum) applying a constraint to some recursive predicate, which we call the *cost predicate*, defined by a set of rules which will be called the *cost rules*. The key intuition behind $\mathscr{P}reM$ constraints is that some form of monotonicity must be preserved while constraints are applied in recursion. A form of monotonicity preserved under extrema is one where the $\geqslant$ or $\leqslant$ relationships holding for cost variables in the bodies of two instances of a rule implies that the same relationship holds for the cost variables in their heads. A stricter condition is the one in which the cost argument in the head is required to be $\geqslant$ (resp. $\leqslant$) than the related cost argument in the body. When this condition is satisfied, upper (lower) bound constraints are $\mathscr{P}reM$ (Proposition 2), because once the constraint is not satisfied for a certain tuple combination and a cost argument, it cannot be satisfied for increased (decreased) costs. Next, we will make the above intuitive explanation more formal. For simplicity, we assume that there is no predicate that is mutually recursive with our cost predicate. The cost rules have two types of goals. The goals of the first type use *interpreted* predicates, such as arithmetic comparisons and other built-ins whose semantics and implementation are pre-defined in the system. The remaining goals will be called *regular goals* which refer to regular predicates, i.e., predicates whose meaning is being defined by the program at hand. The cost predicates appearing in the head of our rule are also regular predicates. We will use the notation $rg(r)$ to denote the conjunction of the regular goals in the body of a rule r. Thus, in our Example 1, $rg(\mathtt{r_2})$ = "$\mathtt{path(X, Dx), arc(X, Y, Dxy)}$".

Each constraint in the final rule defines a cost argument for a particular regular predicate: e.g., in our previous examples, both the constraints >143 and `is_min` established the second argument of `path` to be the cost argument. For simplicity of discussion, we will only consider cases where for each regular predicate p there is only one cost argument denoted $ca(p)$. Finally, we assume that rules (and predicates) are *instantiated* as usual by assigning to their variables values taken from a suitable universe (e.g., a Herbrand universe that also includes numbers). Now, we will define *valid instances of a rule* to be those where variables are instantiated to values that make all the interpreted goals true. Thus, for our rule $\mathtt{r_2}$ in Example 1, arbitrary values can be assigned to the variables `X` and `Y`. However, the values assigned to `Dx`, `Dxy` and `Dy` must satisfy the conditions $\mathtt{Dxy} \geqslant 0 \wedge \mathtt{Dy} = \mathtt{Dx} + \mathtt{Dxy}$. In order to determine if extrema are $\mathscr{P}reM$, we have to provide a sufficient condition whereby the validity of the $\geqslant$ or $\leqslant$ relationships for cost variables in the bodies of two instances of the given rule implies that the same relationship holds for the cost variables in their heads. Therefore, we will introduce the notion that *inflated* cost-predicates in the body of the rule also *inflate* the and cost-predicates of the heads of the rules.

Definition 2 (*Inflating/Deflating cost predicates*)
Let $\mathbf{p}'$ and $\mathbf{p}''$ be two instances of a cost predicate p where non-cost arguments of $\mathbf{p}'$ are identical to those of $\mathbf{p}''$. Then when $ca(\mathbf{p}') \geqslant ca(\mathbf{p}'')$, we say

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

that $\mathbf{p}'$ *inflates* $\mathbf{p}''$. Conversely, when $ca(\mathbf{p}') \leqslant ca(\mathbf{p}'')$, we say that $\mathbf{p}'$ *deflates* $\mathbf{p}''$.

Definition 3 (*Inflating/Deflating conjuncts of regular goals*)
Let r be a rule and $\mathbf{r}'$ and $\mathbf{r}''$ be two of its instantiations where all the regular non-cost goals are identical and each regular cost goal of $\mathbf{r}'$ in $rg(r)$ inflates (resp. deflates) the corresponding goal in $\mathbf{r}''$. Then, we say that the regular goals of $\mathbf{r}'$ *inflate* (resp. *deflate*) the regular goals of $\mathbf{r}''$.

For example, consider the following two instances of a rule r which are identical everywhere but in their cost arguments; the interpreted goals use the square-root function `sqrt`:

$$\mathbf{r}' : \mathtt{path(y, dy')} \leftarrow \mathtt{path(x, dx'), arc(x, y, dxy), dxy} \geqslant 0, \mathtt{dy'} = \mathtt{dx'} + \mathtt{sqrt(dxy)}.$$
$$\mathbf{r}'' : \mathtt{path(y, dy'')} \leftarrow \mathtt{path(x, dx''), arc(x, y, dxy), dxy} \geqslant 0, \mathtt{dy''} = \mathtt{dx''} + \mathtt{sqrt(dxy)}.$$

If we assume that $\mathtt{dx''} > \mathtt{dx'}$, then $\mathtt{path(x, dx'')}$ represents an arbitrary inflation of $\mathtt{path(x, dx')}$. Likewise, the regular goal conjunction of the second rule, i.e., $\mathtt{path(x, dx''), arc(x, y, dxy)}$, represents an arbitrary inflation of regular goal conjunction of the first rule. We now have the following condition under which extrema constraints become $\mathscr{P}reM$.

Definition 4 (*Inflation / Deflation preserving rules*)
A rule r will be said to be *inflation-preserving* (resp. *deflation-preserving*), when for every valid instance $\mathbf{r}'$ of r, and for every V that is an inflation-preserving (resp. deflation-preserving) instance of the regular goals of $\mathbf{r}'$, there exists a valid instance $\mathbf{r}''$ of r which (i) has V as its regular goals, and (ii) its head that inflates (resp. deflates) the head of $\mathbf{r}'$.

To apply this definition to our example, we must check that for each $\mathtt{path(x, dx''), arc(x, y, dxy)}$ that inflates $\mathtt{path(x, dx'), arc(x, y, dxy)}$ there exists a valid instance of the interpreted goals of $\mathbf{r}''$ that delivers a head that inflates that of $\mathbf{r}'$. Now, `sqrt` has two valid interpretations, one that returns a positive square-root value and the other that returns its opposite negative value. Now, if we only allow the negative interpretation of `sqrt`, we see that for $\mathtt{dy''} \geqslant \mathtt{dy'}$, $\mathbf{r}''$ does not inflate $\mathbf{r}'$, and thus r is not inflation-preserving; however, if we only allow the positive interpretation *or we allow both*, then our rule becomes inflation preserving.

Next, let us call *procedure* the set of rules in a program defining the same predicate, while those defining a cost predicate will be called a cost procedure. A cost procedure will be called inflation-preserving (resp. deflation-preserving) when all its rules are inflation-preserving (resp. deflation-preserving). Moreover, by applying a $\mathscr{P}reM$ constraint γ to a cost procedure we mean that the constraint is applied to each rule in the procedure. Then, we have

Proposition 1
Min constraints are $\mathscr{P}reM$ to deflation-preserving cost procedures, and max constraints are $\mathscr{P}reM$ to inflation-preserving cost procedures.

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

Proof
To prove that γ is $\mathscr{P}reM$, we need to prove that for each rule r of our cost procedure $\gamma(T(I)) = \gamma(T(\gamma(I)))$. If r is non-recursive (i.e., an exit rule), the result is immediate since $\gamma(T(\emptyset)) = \gamma(T(\gamma(\emptyset)))$. If r is a recursive rule and $\mathbf{r}'$ is an instance of r, then the application of $\gamma = \texttt{max}$ to I invalidates $\mathbf{r}'$ iff there is another instance of r say $\mathbf{r}''$ which inflates the regular goal conjunct of $\mathbf{r}'$. But since this rule is inflation-preserving, the head of $\mathbf{r}''$ inflates that of $\mathbf{r}'$ and it will thus be eliminated by the application of the $\gamma = \texttt{max}$ outside the parentheses. Therefore, we conclude that `max` is $\mathscr{P}reM$ to inflation-preserving cost procedures, and symmetrically `min` is $\mathscr{P}reM$ to deflation-preserving cost procedures. □

For instance, returning to rules $\texttt{r}_1$ and $\texttt{r}_2$ in Example 1, we see that they are both inflation-preserving and deflation-preserving. Thus, we can safely transfer either the `is_min` or `is_max` constraint from $\texttt{r}_4$, as shown by Example 3 that was produced by transferring `is_min`.

We next provide transferability conditions for upper/lower-bound constraints.

Definition 5 (*Ascending/Descending mappings from rule bodies to their heads*)
A rule r defines an *ascending* (resp. *descending*) *mapping* when in each of its valid instances $\mathbf{r}$, the cost argument in the head is $\geqslant$ (resp. $\leqslant$) the cost argument in each of the cost goals of $\mathbf{r}$.

Thus, for our previous rule using `sqrt`, if we only use the positive interpretation of `sqrt`, then our rule defines an ascending mapping, but under negative interpretation of `sqrt` we have a descending mapping since $\texttt{dy}' < \texttt{dy}$. Thus, if we allow both interpretations the rule neither defines an ascending mapping nor a descending one. Likewise, rules $\texttt{r}_1$ and $\texttt{r}_2$ in Example 1 define an ascending mapping, whereas they define a descending mapping if we assume that the arc lengths in our graphs are $\leqslant 0$. Note that inflation-/deflation-preserving differ from ascending/descending mappings in that the former does not require the cost in the head to be $\geqslant$ (resp. $\leqslant$) each cost argument in the body. For example, if in $\texttt{r}_2$ we replace the goal $\texttt{Dy} = \texttt{Dx} + \texttt{Dxy}$ with $\texttt{Dy} = (\texttt{Dx} + \texttt{Dxy}) - 99$, the rule remains inflation-preserving, while it no longer defines an ascending mapping.

Now, with conditions such as <143 and $\leqslant 143$ defining *upper-bound constraints* (and dually for *lower-bound constraints*), we have that

Proposition 2
Upper-bound constraints are $\mathscr{P}reM$ to cost procedures defining ascending mappings, whereas lower-bound constraints are $\mathscr{P}reM$ to cost procedures defining descending mappings.

Proof
Say that an upper-bound constraint γ has been applied to a rule r in our cost procedure defining an ascending mapping. The proof that $\gamma(T(I)) = \gamma(T(\gamma(I)))$ is trivial for each exit rule since they are applied upon $I = \emptyset$. Say now that T is the ICO of a recursive rule r. Then the application of our $\mathscr{P}reM$ upper-bound constraint eliminates each instance $\mathbf{r}'$ which fails the constraint; but since r defines an

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

ascending mapping, the head of $\mathbf{r}'$ will then fail the same upper-bound constraint. Thus, the upper-bound constraints are $\mathscr{P}reM$ to all rules of a cost procedure defining an ascending mapping, and symmetrically for lower-bounds and descending mappings. □

We see that the cost procedure in Example 1 defines an ascending mapping and thus the transfer of upper-bound $\texttt{Dy} < 143$ done in Example 2 is correct. The rules composing the procedure also define descending mappings and would also support the transfer of lower-bound constraints. However, it is also very important to observe how upper-bound and lower-bound constraints interact and often interfere with extrema. For instance, say that rule $\texttt{r}_4$ applied to our Example 2 specified both the constraints $\texttt{Dy} < 143$ and $\texttt{is_min}((\texttt{Y}),(\texttt{Dy}))$. Then, we see that each of the two and their conjunct $\mathscr{P}reM$ and can be pushed into the procedure. But in general, the fact that each constraint is $\mathscr{P}reM$ does not assure that its conjunct is too. For instance, after the transfer of $\texttt{Dy} < 143$, rule $\texttt{r}'_2$ is still deflation-preserving but no longer inflation-preserving. Thus, while $\texttt{is_min}((\texttt{Y}),(\texttt{Dy}))$ is still transferable after we transfer $\texttt{Dy} < 143$, the constraint $\texttt{is_max}((\texttt{Y}),(\texttt{Dy}))$ is not. Symmetrically, if we transfer the constraint $\texttt{Dy} \geqslant 143$ instead of $\texttt{Dy} < 143$, our rules become inflation-preserving, whereby max constraints can be transferred, but min constraints cannot since the rules are no longer deflation-preserving.

5 Monotonic aggregates

At the core of the approach proposed by Mazuran *et al.* (2013b), there is the observation that the cumulative version of standard count is monotonic in the lattice of set containment. Thus, Mazuran *et al.* (2013b) introduce `mcount` as the aggregate function that returns all natural numbers up to the cardinality of the set. The use of `mcount` in actual applications is illustrated by the following example that is similar to one proposed by Ross and Sagiv (1992).

Example 4 (*Join the party once you see that three of your friends have joined*)
The organizer of the party will attend, while other people will attend if the number of their friends attending is greater or equal to 3, i.e., $\texttt{Nfx} \geqslant 3$.

$$\begin{aligned}
&\texttt{attend(X)} \leftarrow && \texttt{organizer(X)}.\\
&\texttt{attend(X)} \leftarrow && \texttt{cntfriends(X, Nfx)}, \texttt{Nfx} \geqslant 3.\\
&\texttt{cntfriends(Y, N)} \leftarrow && \texttt{attend(X)}, \texttt{friend(Y, X)}, \texttt{mcount((Y),(X), N)}.
\end{aligned}$$

Here too, we use the body notation for our monotonic count-distinct aggregate, $\texttt{mcount}((\texttt{Y}),(\texttt{X}),\texttt{N})$, where its first argument (`Y`) is the group-by argument, its second (`X`) is the *count argument* and the third is the result of the count aggregate. The count argument consists of one or more sub-arguments, whereas the group-by can consist of zero or more sub-arguments. As proven by Mazuran *et al.* (2013b), the semantics of $\texttt{mcount}((\texttt{Y}),(\texttt{X}),\texttt{N})$ can be reduced to that of Horn Clauses. The use of `mcount` in recursion allows us to express programs, such as counting paths between two nodes, that are not expressible using the standard aggregates in stratified programs

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

(Mumick and Shmueli 1995). Moreover, since mcount is monotonic, the fixpoint semantics and optimization techniques of Datalog remain valid. In particular, we can optimize our program by pushing max into the recursion.

By the query goal attend(N), the program in Example 4 returns the names of the people who attend the party. To find out the number of friends each attendee will have at the party, we can use the query goal cntfriends(Y, X), which, however, is likely to return many answers, since for an attendee with k friends it will return his/her name k times. A solution to this problem is to use the following rule to return the final results[3]:

$$\texttt{fcount(Y, N)} \leftarrow \texttt{cntfriends(Y, N), is_max((Y), (N))}.$$

The obvious question now is whether is_max can be transferred into the recursive rules defining cntfriends(Y, N), which is true if these rules are inflation-preserving. We have previously examined the case where there is only one recursive rule, and here we can reduce our problem to that situation by taking the composition of the two recursive rules in Example 4, obtaining the following rule:

$$\texttt{cntfriends(Y, N)} \leftarrow \texttt{cntfriends(X, Nfx), Nfx} \geqslant 3, \texttt{friend(Y, X), mcount((Y), (X), N)}.$$

Now, we must determine if the condition of inflation preservation is satisfied so that we can transfer the max into our recursive rules. Since there is no valid instance of this rule for Nfx < 3, we only need to consider instantiations in which Nfx $\geqslant 3$. Note that the goal mcount((Y), (X), N) is independent from Nfx, and the count cannot decrease: thus, the rule is inflation-preserving.

Example 5 (*Join the party once three of your friends have joined using standard count*)

```
attend(X) ←          organizer(X).
attend(X) ←          cntfriends(X, Nfx), Nfx ⩾ 3.
cntfriends(Y, N) ←   attend(X), friend(Y, X), count((Y), (X), N).
fcount(Y, N) ←       cntfriends(Y, N).
```

This formulation using count is more efficient than the one using mcount since it implies that condition Nfx $\geqslant 3$ is no longer checked for each $+1$ increment: it is instead checked for each new count value delivered by the previous step in the iteration. At each iteration, the old count value is incremented to include all the newly found friends. Once the fixpoint is reached, fcount(Y, N) returns, for each person Y attending the party, the actual count N of Y's friends at the party.

We might also allow users to write programs such as that in Example 5 where count is used instead of mcount in recursion. The compiler will process these programs by interpreting count as a max mcount and transferring max out of recursion before checking that the resulting program is indeed inflation preserving. If this check succeeds, then the original program is executed; otherwise, an error message, or a stern warning, is returned to the user.

[3] Along with the people who count three friends at the party, this will also return those who count only one or two friends among those, who can be removed by an additional final condition N $\geqslant 3$.

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

5.1 From monotonic SUM to regular SUM

The notion of monotonic sum msum for positive numbers was also introduced by Mazuran *et al.* (2013b) as an extension of mcount. For instance, say that we have a set of facts following the template part(Pno, Cost), where for each part number we have its cost expressed as a positive integer. To compute the total value of the parts on store, we can add up their costs using the sum aggregate, as follows:

$$\texttt{total(T)} \leftarrow \texttt{part(Pno, C)}, \texttt{sum(( ), (Pno, C), T)}.$$

Here, (Pno, C) states that the sum is taken over every C appearing under a distinct Pno. As such, the semantics of sum is easy to reduce to that of count, by first expanding each cost C into C pairs (C, J), with $1 \leqslant \texttt{J} \leqslant \texttt{C}$ and then computing the sum by counting these pairs. For instance, the pair (bolt, 4) is represented by the four tuples: (bolt, 4, 1), (bolt, 4, 2), (bolt, 4, 3), (bolt, 4, 4). The meaning of sum(( ), (Pno, C), T) can now be defined by the pair of goals int_up2(C, Int), count(( ), (Pno, C, Int), T), where int_up2(C, Int) produces all the integers up to C. Thus, our previous rule can be rewritten as

$$\begin{aligned}
&\texttt{total(T)} \leftarrow && \texttt{part(Pno, C)}, \texttt{int_upt2(C, Int)}, \texttt{count((), (Pno, C, Int), T)}.\\
&\texttt{int_up2(C, 1)} \leftarrow && \texttt{C} > 0.\\
&\texttt{int_up2(C, J1)} \leftarrow && \texttt{int_up2(C, J)}, \texttt{J1} = \texttt{J} + 1, \texttt{J1} \leqslant \texttt{C}.
\end{aligned}$$

In a totally analogous way, we can define msum via int_up2 and mcount, whereby, e.g., msum((), (C, Int), T) is defined as equivalent to int_upt2(C, Int), mcount((), (C, Int), T). Furthermore, the regular sum is simply the result of applying the max aggregate to msum, and this leads to many useful applications where sum is used directly in inflation-preserving recursive rules. For instance, say that we have the classical part-explosion problem where we have a set of basic parts with their cost basic(Part, Cost), and then we have information about assembled parts, describing for each assembled parts their subparts and quantities in the assemblage assb(Part, SubPart, Qty). To compute the cost of all parts, we can write the following program:

Example 6 (*Part explosion problem: the cost of parts as sum of the cost of its subparts*)

$$\begin{aligned}
&\texttt{cost(Part, Cost)} \leftarrow && \texttt{basic(Part, Cost)}.\\
&\texttt{cost(Part, Ncost)} \leftarrow && \texttt{assb(Part, SP, Qty)}, \texttt{cost(SP, Cost)},\\
& && \texttt{CQ} = \texttt{Cost} * \texttt{Qty}, \texttt{sum((Part), (SP, CQ), Ncost)}.\\
&\texttt{finalcost(Part, Cost)} \leftarrow && \texttt{cost(Part, Cost)}.
\end{aligned}$$

Presented with this program, the compiler must make sure it can be viewed as the result of transferring the max to the program obtained by replacing sum with msum in the second rule. To achieve this, the compiler will check that (i) the arguments CQ of msum are all positive, and (ii) that the rules are inflation-preserving.

To enable compile-time checking of (i), the user must add the goal $\texttt{CQ} > 0$ to the recursive rule in our example. In the absence of this, or equivalent conditions, the compiler returns a stern warning message, but the user will still be able to proceed

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

with execution under the optional trust-but-verify policy that reports this violation as a run-time error. This trust-but-verify option represents a very useful debugging tool, and is also available if, for any reason, the compiler cannot verify (ii).

This example illustrates the great benefits that should be expected from allowing sum in recursion, for those situations where its formal semantics can be reduced to the transfer of a final max into `msum`. However, there seems to be no obvious application that requires the use of `msum` instead of `sum`. Moreover, say that the regular sum of costs of our parts amounts to 50,000: the msum will return the first 50,000 integers, thus causing serious inefficiencies. Thus, while the compiler might use the notion of `msum` to verify that the given program using `sum` in its recursive rules has a fixpoint semantics, `msum` should not be part of the actual language. Since it is implemented directly without calling upon its equivalent count representation, sum can also be applied to arbitrary positive numbers represented in scientific (mantissa and exponent) notation, given that they can be viewed as large integers (Mazuran *et al.* 2013b).

6 Seminaive optimization

So far, we have focused on optimizing the naive fixpoint computation, which basically generates a new set of tuples from the whole set obtained in the last step. We can now perform the seminaive improvement which can be used to improve the computation of the program obtained from the transferring of constraints. For standard Datalog, the seminaive computation operates as follows:

(A) keep track of the step at which each new atom was created;
(B) modify the rules into their differential version, which exploits the fact that rule instances whose bodies only contain atoms produced in older steps cannot produce new atoms;
(C) use the modified rules to produce atoms, enforcing the no-duplicate constraints as the atoms are produced.

For programs obtained by the pushing of constraints, (A) and (B) above remain the same, but (C) is significantly extended beyond deduplication to enforce constraints, whereby new atoms produced in (B) might not be kept and existing atoms might instead be eliminated. For constraints involving extrema, for instance, besides the removal of duplicate atoms, step (C) will also make sure no atom that violates the extremum constraint is added to the working set. For instance, if a program with max constraints produces a new (`b`, `D`), then this new pair is added to the working set only if this does not already contain a pair (`b`, `D1`) with `D1` $\geqslant$ `D`, and if this new pair is inserted then each pair (`b`, `D2`) with `D2` $<$ `D` is deleted from the working set. Symmetric properties hold for min aggregates. The basic (A), (B) and (C) steps are also used to compute `mcount` with the proviso that every non-duplicate atom produced in (C) will increment the running count by one, and the new count value is added to the output. For count and sum, the running counts and sum are incremented at step (C), but atoms are only produced at the end of the fixpoint

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

computation. Further refinements of these strategies support the optimizations of Shkapsky *et al.* (2016).

7 Related work

Several solutions, including Kemp *et al.* (1989) and Srivastava and Ramakrishnan (1992), addressed the problem of pushing comparison constraints into recursive rules, without exploring relationships with the problems of supporting extrema and other aggregates into recursion. Supporting aggregates in recursion is a difficult problem which has been the topic of much research work. Several previous approaches focused primarily on providing a formal semantics that could accommodate the non-monotonic nature of the aggregates. In particular, Mumick *et al.* (1990) discussed programs that are stratified w.r.t. aggregate operators and proved that a perfect model exists for these programs. Then, Kemp and Stuckey (1991) defined extensions of the well-founded semantics to programs with aggregates, and later showed that these might have multiple and counter-intuitive stable models. The notion of cost-monotonic extrema aggregates was introduced by Ganguly *et al.* (1995), using perfect models and well-founded semantics, whereas Greco *et al.* (1992) showed that to express greedy algorithms they require the don't-care non-determinism of the stable-model semantics provided by the choice construct.

The problem of optimizing programs with extrema by early pruning of non-relevant facts was studied by Sudarshan and Ramakrishnan (1991). A more recent work by Furfaro *et al.* (2002) devises an algorithm for pushing max and min constraints into recursion while preserving query equivalence under certain specific monotonicity assumptions. Those approaches provided effective optimization techniques for min and max aggregates in recursion, but did not consider count and sum aggregates. The idea of relating comparison with extrema constraints is also used by Liu *et al.* (2012) to optimize logical quantification.

A general approach to deal with all four aggregates was proposed by Ross and Sagiv (1992), who advocated the use of semantics based on specialized lattices, different from set-containment, whereby each aggregate will then define a monotonic mapping in its specialized lattice. However, Van Gelder (1993) pointed out that automatically determining the correct lattices would be difficult in practice, and this was one of the causes that prevented the deployment of the monotonic-aggregate idea in query languages for the following 20 years. Interestingly, a similar idea (with similar limitations) was more recently proposed by Swift and Warren (2010) and in several other works, e.g., the paper by Zhou *et al.* (2010; 2015) in the context of tabling in Prolog. Also, here is programmer's duty to make sure that a proper partial order/lattice type is used for the program at hand.

A renewed interest in Big Data analytics brought a revival of Datalog as a parallelizable language for expressing more powerful graph and data-intensive algorithms—including many that require aggregates in recursion (Seo *et al.* 2013; Shkapsky *et al.* 2013; Wang *et al.* 2015). The solution proposed here (and previously sketched by Zaniolo *et al.* (2016)) builds on the monotonic count and sum proposed by Mazuran *et al.* (2013a) and provides the foundation of the efficient and scalable

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

systems discussed by Shkapsky *et al.* (2015), Yang *et al.* (2015), Shkapsky *et al.* (2016) and Yang *et al.* (2017).

8 Conclusion

In this paper, we have presented a unifying solution to the problem of allowing the use of aggregates in recursive Datalog programs. The widely recognized practical importance of solving this problem has motivated in-depth formal investigations by many excellent researchers who during the last 25 years studied and proposed specialized solutions for parts of this multi-faceted problem. The main advantage of the unified treatment for the four basic aggregates, min, max, count and sum, proposed in this paper is that it allows a simple specification of algorithms of great practical interest with the guarantee that they have a minimal model semantics that is computed via the standard fixpoint iteration, which can be further refined by the standard optimization techniques of bottom-up Datalog implementations, such as seminaive fixpoint computation, magic-set, left-linear and right-linear recursive rules. The practical significance of these results are demonstrated by the fact that, using these techniques, our BigDatalog system for Apache Spark was able to outperform graph applications written in GraphX and running on the same system (Shkapsky *et al.* 2016), suggesting that Datalog, as a general-purpose high-level language, might be able to outperform high-level languages in the specialized application domains they were designed for.

Practical experience with real-world applications (Condie *et al.* 2017) also taught us important lessons on the dual role of fixpoint semantics and optimization that inspired the title and the research of this paper. Initially, we followed the query optimization approach since it appeals to intuition and was used by many researchers in the past. This led to the equivalent minimal-fixpoint semantics for programs that, via $\mathscr{P}reM$, can exploit extrema and other constraints in recursive rules. However, after developing applications in several domains (Condie *et al.* 2017), we realized that specifying $\mathscr{P}reM$ constraints in recursive rules represents a natural way to write those applications (perhaps because of the similarity with their iterative procedural counterparts we are familiar with). Moreover, this allowed us to overcome major obstacles that proved insurmountable for the optimization approach. For instance, we have seen examples where goals stating that edge lengths or basic part costs are all positive must be added before the constraints are $\mathscr{P}reM$. But most users, knowing that those edges or costs are always positive, (e.g., because of constraints in the database or other rules in the program) prefer to submit their programs for execution without having to add idle conditions to make $\mathscr{P}reM$ manifest. In these situations, the system should produce a stern warning message, but the user should still be allowed to proceed with execution. This policy results in greater convenience and the power of handling situations that cannot be handled by optimizers. The user can also call upon the system assistance by initially executing in "debugging mode" under the trust-but-verify policy outlined in Section 5. This will either produce counter-examples that show that γ is not $\mathscr{P}reM$, or after enough testing, the user

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

can reliably assume that the property holds, and the debugging mode can be turned off.

Acknowledgements

We would like to thank the reviewers for the comments and suggested improvements.

References

Aref, M. *et al.* 2015. Design and implementation of the LogicBlox system. In *Proc. of SIGMOD*. ACM, 1371–1382.

Arni, F., Ong, K., Tsur, S., Wang, H. and Zaniolo, C. 2003. The deductive database system LDL++. *Theory and Practice of Logic Programming 3,* 1, 61–94.

Chimenti, D., O'Hare, A. B., Krishnamurthy, R., Tsur, S., West, C. and Zaniolo, C. 1987. An overview of the LDL system. *IEEE Data Engineering Bulletin 10,* 4, 52–62.

Condie, T., *et al.* 2017. *Advanced Applications by Least-Fixpoint Algorithms Specified using Aggregates in Datalog*. Technical Report 170012, UCLA CSD.

Faber, W., Pfeifer, G. and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence 175,* 1, 278–298.

Furfaro, F., Greco, S., Ganguly, S. and Zaniolo, C. 2002. Pushing extrema aggregates to optimize logic queries. *Information Systems 27,* 5 (July), 321–343.

Ganguly, S., Greco, S. and Zaniolo, C. 1995. Extrema predicates in deductive databases. *Journal of Computer and System Sciences 51,* 2, 244–259.

Gelfond, M. and Zhang, Y. 2014. Vicious circle principle and logic programs with aggregates. *Theory and Practice of Logic Programming 14,* 4–5, 587–601.

Greco, S., Zaniolo, C. and Ganguly, S. 1992. Greedy by choice. In *Proc. of PODS*. ACM, 105–113.

Kemp, D., Meenakshi, K., Balbin, I. and Ramamohanarao, K. 1989. Propagating constraints in recursive deductive databases. In *North American Conference on Logic Programming*, 981–998.

Kemp, D. B. and Stuckey, P. J. 1991. Semantics of logic programs with aggregates. In *Proc. of ISLP*, 387–401.

Liu, Y. A., Stoller, S. D., Lin, B. and Gorbovitski, M. 2012. From clarity to efficiency for distributed algorithms. *SIGPLAN Notices 47,* 10 (October), 395–410.

Mazuran, M., Serra, E. and Zaniolo, C. 2013a. A declarative extension of Horn clauses, and its significance for Datalog and its applications. *Theory and Practice of Logic Programming 13,* 4–5, 609–623.

Mazuran, M., Serra, E. and Zaniolo, C. 2013b. Extending the power of Datalog recursion. *The VLDB Journal 22,* 4, 471–493.

Morris, K. A., Ullman, J. D. and Gelder, A. V. 1986. Design overview of the NAIL! system. In *Proc. of ICLP*, 554–568.

Mumick, I. S., Pirahesh, H. and Ramakrishnan, R. 1990. The magic of duplicates and aggregates. In *VLDB*. Morgan Kaufmann Publishers Inc., 264–277.

Mumick, I. S. and Shmueli, O. 1995. How expressive is stratified aggregation? *Annals of Mathematics and Artificial Intelligence 15,* 3–4, 407–435.

Pelov, N., Denecker, M. and Bruynooghe, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *Theory and Practice of Logic Programming 7,* 3, 301–353.

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press

Przymusinski, T. C. 1988. Perfect model semantics. In *Proc. of ICLP/SLP*, 1081–1096.

Ramakrishnan, R., Srivastava, D. and Sudarshan, S. 1992. CORAL - control, relations and logic. In *Proc. of PVLDB*, 238–250.

Ross, K. A. and Sagiv, Y. 1992. Monotonic aggregation in deductive databases. In *Proc. of PODS*, 114–126.

Seo, J., Park, J., Shin, J. and Lam, M. S. 2013. Distributed socialite: a Datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment 6,* 14, 1906–1917.

Shkapsky, A., Yang, M., Interlandi, M., Chiu, H., Condie, T. and Zaniolo, C. 2016. Big data analytics with Datalog queries on Spark. In *Proc. of SIGMOD*. ACM, 1135–1149.

Shkapsky, A., Yang, M. and Zaniolo, C. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *Proc. of ICDE*. IEEE, 867–878.

Shkapsky, A., Zeng, K. and Zaniolo, C. 2013. Graph queries in a next-generation Datalog system. *Proceedings of the VLDB Endowment 6,* 12, 1258–1261.

Son, T. C. and Pontelli, E. 2007. A constructive semantic characterization of aggregates in answer set programming. *Theory and Practice of Logic Programming 7,* 3, 355–375.

Srivastava, D. and Ramakrishnan, R. 1992. Pushing constraint selections. In *Journal of Logic Programming*, 301–315.

Sudarshan, S. and Ramakrishnan, R. 1991. Aggregation and relevance in deductive databases. In *Proc. of VLDB*, 501–511.

Swift, T. and Warren, D. S. 2010. Tabling with answer subsumption: Implementation, applications and performance. In *Proc. of JELIA*, 300–312.

Vaghani, J., Ramamohanarao, K., Kemp, D. B., Somogyi, Z., Stuckey, P. J., Leask, T. S. and Harland, J. 1994. The Aditi deductive database system. *VLDB Journal 3,* 2, 245–288.

van Emden, M. H. and Kowalski, R. A. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM 23,* 4, 733–742.

Van Gelder, A. 1993. Foundations of aggregation in deductive databases. In *Deductive and Object-Oriented Databases*. Springer, 13–34.

Wang, J., Balazinska, M. and Halperin, D. 2015. Asynchronous and fault-tolerant recursive Datalog evaluation in shared-nothing engines. *Proceedings of the VLDB Endowment 8,* 12, 1542–1553.

Yang, M., Shkapsky, A. and Zaniolo, C. 2015. Parallel bottom-up evaluation of logic programs: DeALS on shared-memory multicore machines. In *Technical Communications of ICLP*.

Yang, M., Shkapsky, A. and Zaniolo, C. 2017. Scaling up the performance of more powerful Datalog systems on multicore machines. *The VLDB Journal 26,* 2, 229–248.

Yang, M. and Zaniolo, C. 2014. Main memory evaluation of recursive queries on multicore machines. In *Proc. of IEEE Big Data*, 251–260.

Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R. T., Subrahmanian, V. S. and Zicari, R. 1997. *Advanced Database Systems*. Morgan Kaufmann.

Zaniolo, C., Yang, M., Das, A. and Interlandi, M. 2016. The magic of pushing extrema into recursion: Simple, powerful Datalog programs. In *Proc. of AMW*.

Zhou, N.-F., Barták, R. and Dovier, A. 2015. Planning as tabled logic programming. *Theory and Practice of Logic Programming 15,* 4–5, 543–558.

Zhou, N.-F., Kameya, Y. and Sato, T. 2010. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *Proc. of ICTAI '10*. Washington, DC, USA, 213–218.

https://doi.org/10.1017/S1471068417000436 Published online by Cambridge University Press