Improving the Power, Performance and Usability of Datalog by Pushing Constraints into Recursion

Carlo Zaniolo, Mohan Yang, Ariyam Das and Matteo Interlandi

University of California, Los Angeles
{zaniolo,yang,ariyam}@cs.ucla.edu, m.interlandi@gmail.com

UCLA CSD Technical Report Number: 160003
March 7, 2016

Abstract. We introduce a novel query optimization method based on pushing extrema and other integrity constraints into recursion. This optimization produces a more efficient computation that preserves the soundness and completeness for the predicates specified by the integrity constraints. Complex algorithms, including greedy algorithms, can now be specified via simpler programs that are stratified with respect to min and max, and then implemented via optimized rules that use these aggregates in the seminaive fixpoint computation to achieve more efficient execution. This rewriting is not limited to extrema, but in many situations a similar optimization is applicable to other constraints as well. The relationships of this optimization technique with recent works on monotonic aggregates and their combined efficient implementation in the seminaive fixpoint method are also elucidated.

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 compile and execute efficiently over different systems and architectures, including massively parallel ones [15, 17] [21, 19, 1] [2]. 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 [15, 17, 21, 19]. As discussed in more details in the related work section, extending the declarative semantics of Datalog to allow aggregates in recursion represents a difficult problem that had seen much action in the heydays of Datalog, twenty years ago: [6, 12, 4, 12]. However, the solutions proposed in those papers were not widely deployed in systems, for a number of reasons, including the problems pointed out in [18], and for many years interest in parallel Datalog remained dormant. However, the recent explosion of interest in Big Data analytics has 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 [1, 17, 15, 19]. Some of those approaches did not tackle open semantic issues, and also ignored the examples illustrating those issues that were presented in [12]1. Fortunately, at UCLA, researchers managed to first provide a rigorous solution to the monotonic aggregate problem, and then proceeded to demonstrate portable and scalable systems that support such aggregates on (i) workstations [16], (ii) multicore systems [21, 20], and (iii) Spark-enabled clusters [2].

However, as our experience with such monotonic aggregates expanded, we came to realize that an alternative and often simpler approach exists when extrema (i.e., min and max) are used to specify recursive predicates: rather than writing programs with monotonic min

---

1 Those examples will be discussed in Section 9
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 (although the system will instead use an equivalent and more efficient computation to construct that result). Indeed, we observed that it is quite simple to optimize the computation of those programs by moving the extrema into the actual rules used in the fixpoint iteration. This optimization by constraint approach (very much in the style of relational optimization) produces a simple declarative semantics, combined with a very efficient operational semantics: we can now express greedy algorithms that could not be easily expressed with monotonic min and max. Many other programs that use constraints other than extrema can also be optimized in a similar way.

The paper is organized as follows. In the next section, we review the formal semantics of programs, and the iterated fixpoint computation for stratified programs, and in the following section we define the notion of constraint pushing for our programs. Then, in Section 4 we illustrate our optimization technique with the help of several examples. Then, in Section 5 we provide formal criteria that assure the correctness of our optimization approach. In section 6 we introduce a notation that let us express several extrema constraints using one rule needed for the expression of the Greedy algorithms discussed in Section 7. Modifications required in the Seminaive Fixpoint method to account for the new optimization are discussed in Section 8, whereas its integration with monotonic count and other monotonic aggregates is given in Section 9. An in-depth discussion of related work and a short conclusion close the paper.

2 Constrained Immediate Consequence Operators

The declarative and constructive semantics of a Datalog program $P$ is defined in terms of the operator by $T_P(I)$ where $I$ denotes an Herbrand interpretation of $P$ and $T_P$ is the Immediate Consequences Operator for $P$. For basic Datalog, without negation or aggregates, $T_P(I)$ is a monotonic continuous mapping in the lattice of set-containment which the interpretation $I$ belongs to, whereby we have the following well-known properties:

1. A unique minimal (w.r.t. set-containment) solution of the fixpoint equation $I = T_P(I)$ always exists and it is known as the least-fixpoint of $T_P$ which defines the declarative semantics of $P$.
2. The fixpoint iteration $T_P^n(\emptyset)$, often called the naive fixpoint of $T_P$, defines the operational semantics of our program. For cases of practical interest, the computation only needs to be performed till the first integer $n + 1$ where, $T_P^{n+1}(\emptyset) = T_P^n(\emptyset)$. For positive programs without negation and aggregates the operational and declarative semantics coincide.

In order to express many practical applications, basic Datalog must be extended to allow the use of negation and aggregates, and in fact most Datalog compilers do so, but require that the programs be stratified w.r.t. negation and aggregates, such as the following program:

Example 1. Shortest path from node $a$

$$(r_1) \quad \text{path}(Y,Dy) \leftarrow \text{arc}(a,Y,Dy),$$

$$(r_2) \quad \text{path}(Y,Dy) \leftarrow \text{path}(X,Dx), \text{arc}(X,Y,Dxy), Dy = Dx + Dxy.$$ 

$$(r_3) \quad \text{spath}(Y,\min(Dy)) \leftarrow \text{path}(Y,Dy).$$
The syntax \texttt{min(Dy)} above illustrates the head notation for aggregates that is used in many Datalog systems, and follows SQL-2 approach of allowing one or more more group-by variables for each aggregate. Thus in our example, \texttt{Dy} is the aggregate variable (specifically the \texttt{min} variable) and \texttt{Y} is the group-by variable. We will often refer to the \texttt{min} and \texttt{max} variables as \textit{cost variables}. Now, unlike other Datalog aggregates, the semantics of extrema (i.e., \texttt{min} and \texttt{max}) can be easily reduced to that of negation, whereby the last rule in Example 1 can be replaced by:

\begin{example}
Min via negation in stratified programs.

\begin{align*}
\texttt{spath(Y,Dy)} & \leftarrow \texttt{path(Y,Dy)}, \neg \texttt{betterpath(Y,Dy)}. \\
\texttt{betterpath(Y,Dy)} & \leftarrow \texttt{path(Y,Dyy)}, \texttt{Dyy} < \texttt{Dy}.
\end{align*}
\end{example}

Re-expressing our \texttt{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 which is relevant to the issues discussed in this paper and is discussed in Section 9.

Stratification can be used avoid the semantic problems that caused by using aggregates in recursion. For instance, in our example \texttt{spath} belongs to a stratum that is above that of \texttt{path}, whereby our program is assured to have a perfect-model semantics [22]. The perfect model of a stratified program is unique and can be computed using an \textit{iterated fixpoint computation} [22], 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 \texttt{r1} and \texttt{r2}, before selecting values that are minimal using \texttt{r3}. 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 3 below, the constraint is imposed by the last rule that, for each point reached, selects the minimal value of its distance from \texttt{a}. This non-monotonic constraint can now be pushed into the recursive rules whereby our new rules become:

\begin{example}
Optimized shortest path from node \texttt{a}

\begin{align*}
\texttt{path(Y,min(Dy))} & \leftarrow \texttt{arc(a,Y,Dy)}. \\
\texttt{path(Y,min(Dy))} & \leftarrow \texttt{path(X,Dx)}, \texttt{arc(X,Y,Dxy)}, \texttt{Dy} = \texttt{Dx} + \texttt{Dxy}. \\
\texttt{spath(Y,Dy)} & \leftarrow \texttt{path(Y,Dy)}.
\end{align*}
\end{example}

The rules so obtained define the optimized ICO that will be used in the fixpoint iteration to construct \texttt{path}, applicable to a large class of programs having the semantic and syntactic properties described next.

3 Problem Definition

As illustrated by Example 1, we consider programs (or segments of larger programs) consisting of (i) rules defining one or more recursive predicate(s) and (ii) one constraint rule having one or more of those recursive predicates as its goals and constraints on the variables of this goal. Thus, we have stratified programs where the iterated fixpoint computation [22] generates for the head of the constraint rule, a set of atoms that we denote as \textit{Gc}. For instance,
in Example 1, the recursive predicate is defined by its exit rule \( r_1 \) and the recursive rule \( r_2 \). In the constraint rule \( r_3 \), the constraint is specified using the head notation for aggregates; alternatively, the min constraint could be expressed using the negated goal and an additional rule as in Example 2. Under either notation, the program is stratified, with the head of the constraint rule, \( \text{spath} \), occupying a stratum higher than the recursive predicate \( \text{path} \), and with \( \text{arc} \) belonging to a stratum lower than \( \text{path} \).

**Constraint Pushing into Recursion (CPR):** Our CPR optimization technique consists in removing the constraint goals from the constraint rule to add them to the rules defining the recursive predicates. We obtain an optimized program in which the constraints are applied to the rules or facts defining the recursive predicate, and the old constraint rule has now become a *copy rule* that returns the values produced by the recursive predicate. Thus our transformed program in Example 2 shows that the min constraint now implanted into the heads of our rules \( r_1 \) and \( r_2 \), whereas \( r_3 \) has become a copy rule from \( \text{path} \) to \( \text{spath} \).

Therefore, we have given simple syntactic rules to identify CPR-structured programs and how to perform the CPR optimization on these programs. Next, we must move to the semantic level, and provide simple rules that assure the semantic correctness of our optimization for the predicate \( G \) in the head of the constraint rule—i.e., conditions under which the fixpoint iteration for the transformed rules *produces the same \( G \) atoms* as those produced by iterated fixpoint on the original rules. We will only consider here deterministic constraints where a constraint \( \gamma \) on an interpretation \( I \) is said to be deterministic whenever there is a unique maximal subset of \( I \) satisfying \( \gamma \). Now, given a CPR-structured program, let (i) \( \gamma \) be the constraints in the constraint rule; then if \( T_\rho \) is the ICO for the rules and facts defining the recursive predicate, then the ICO for those rules and facts after they undergo the CPR transformation is denoted \( T_\gamma \). By this definition we have that \( T_\gamma(I) = \gamma(T_\rho(I)) \). The program produced by the CPR optimization on \( P \) will be called the \( \gamma \)-optimized version of \( P \).

Obviously, the CPR optimization does not bring any change in the iterated fixpoint for the strata below that of our recursive predicate, where \( T_\gamma \) is now used instead of \( T_\rho \) in the fixpoint iteration. Observe that the results produced by the fixpoint iteration on \( T_\gamma \) are *sound* since they are a subset of those produced by the iterated fixpoint on the original program. Once the fixpoint is computed, the derivation of the \( G \)-atoms takes place, where the simple copy rule of the CPR-optimized program has replaced the application of the \( \gamma \)-constraints in the original program.

Since both fixpoint iterations start from the empty set, reasoning by induction we conclude that (i) \( T_\gamma^n(\emptyset) \supseteq T_\rho^n(\emptyset) \), and, thus, (ii) \( \gamma(T_\gamma^n(\emptyset)) \supseteq \gamma(T_\rho^n(\emptyset)) = T_\gamma^n(\emptyset) \). Thus the results produced by iterating over \( T_\gamma \) are all *sound* inasmuch as they are a subset of the results produced by the least fixpoint on the recursive rules. However, completeness, besides soundness is needed to assure the correctness of the CPR-transformation.

**Completeness:** The program produced by the CPR-transformation is *complete* when the following property holds for every positive integer \( n \):

\[ \text{Although in terms of declarative semantics, \text{path} could be placed in the same stratum as \text{arc}, in terms of actual computation is simpler to assume that the former is computed before the latter.} \]

\[ \text{Notably, functional dependencies are non-deterministic constraints.} \]
where, $G_\gamma$ denotes the $G$ atoms produced by the original constraint rule, and $Gc$ are the $G$ atoms produced by the copy rule of the optimized program. The CPR-optimized program shown in Example 3 is complete w.r.t. the original program of Example 1. This, and the several examples of complete programs discussed in the next sections will help us derive the general rules for completeness presented in Section 5.

4 Programs Using Extrema and Other Constraints

Another optimization for the program of Example 1 is possible if we constrain the results of the fixpoint to satisfy the upper-bound constraint $Dy < K$, where $K$ is a constant stored in a base predicate $\text{isconstant}(K)$. For example we use the following rules:

Example 4. Only the paths with a length less than a given value $K$.

\[
\text{apath}(Y, Dy) \leftarrow \text{arc}(a, Y, Dy).
\]

\[
\text{apath}(Y, Dy) \leftarrow \text{apath}(X, Dx), \text{arc}(X, Y, Dxy), Dy=Dx+Dxy, Dy > Dx.
\]

\[
\text{boundpath}(Y, Dy) \leftarrow \text{apath}(Y, Dy), \text{isconstant}(K), Dy < K.
\]

Can be optimized into the following program:

Example 5. Optimized version of the previous example.

\[
\text{apath}(Y, Dy) \leftarrow \text{arc}(a, Y, Dy).
\]

\[
\text{apath}(Y, Dy) \leftarrow \text{apath}(X, Dx), \text{arc}(X, Y, Dxy), Dy=Dx+Dxy, Dy > Dx, \text{isconstant}(K), Dy < K.
\]

\[
\text{bounpath}(Y, Dy) \leftarrow \text{path}(Y, Dy).
\]

The condition $Dy > Dx$ implies that, if $Dx$ does not satisfy the constraint of being less that a constant $K$, then $Dy$ does not satisfy it either. Thus, the transformed program of Example 5 is complete w.r.t. that constraint. In this particular case, the transformed program is free of non-monotonic constructs and thus, along with its operational semantics, also has a declarative least fixpoint semantics. In general though, only the operational semantics of the transformed programs is of interest as it provides a significantly optimized version of the operational semantics of the original program. A large class of important programs that are stratified w.r.t. extrema aggregates can be optimized in this way. Now, pushing min and max aggregates into recursion delivers optimization technique of many uses. Consider for instance the following program that counts the number of edges needed to reach the vertices in our graph starting from vertex $a$:

Example 6. Counting the arcs needed to reach a node.

\[
\text{hops}(a, 0).
\]

\[
\text{hops}(Y, J1) \leftarrow \text{hops}(X, J), \text{arc}(X, Y, _), J1 = J + 1.
\]

The problem with this program is that it never terminates in the presence of loops. To address and solve this problem we can use the following postcondition to find the least number of hops (i.e., arcs) used to reach the node:

\[
\text{leasthops}(X, \text{min}(J)) \leftarrow \text{hops}(X, J).
\]

The pushing of the min constraint produces the following optimized rules:
Example 7. Counting the arcs needed to reach a node.

\[
\text{hops}(a, \min(0)).
\text{hops}(Y, \min(J1)) \leftarrow \text{hops}(X, J), \text{arc}(X, Y, _), J1 = J + 1.
\text{leasthops}(X, J1) \leftarrow \text{hops}(X, J1).
\]

The application of \text{min} to the original fact \text{hops}(a, 0) looks somewhat strange, but this reminds us that, as per its definition via negation, a \text{min} in the head of a rule has as scope all the rules defining the same predicate (and this also holds for \text{max}). Therefore, in our example, the cost \text{C} of \text{hops}(a, \text{K}) produced by the recursive rule will then be compared with 0, and if \text{C} < 0 then this original \text{hops}(a, 0) fact could be discarded (though this is not the case for the example at hand.)

Example 8. Connected components of an undirected graph via label propagation. The final label of a node \text{Z} is the minimum node id among all the nodes that can reach \text{Z}. All nodes with the same label belong to the same connected component.

\[
\text{reach}(X, X) \leftarrow \text{arc}(X, _).
\text{reach}(X, Z) \leftarrow \text{reach}(X, Y), \text{arc}(Y, Z).
\text{cc}(Z, \min(X)) \leftarrow \text{reach}(X, Z).
\]

The pushing of the \text{min} constraint produces the following optimized program.

Example 9. HCC algorithm [5] in Datalog. This formulation was the one used [14], although formal semantics was not discussed in the referenced paper.

\[
\text{reach}(\min(X), X) \leftarrow \text{arc}(X, _).
\text{reach}(\min(X), Z) \leftarrow \text{reach}(X, Y), \text{arc}(Y, Z).
\text{cc}(Z, X) \leftarrow \text{reach}(X, Z).
\]

This program is complete since the \text{min} constraint is inplantable into recursion in the original program.

Conjunction of inequalities and extrema constraints. Let us now return to Example 1 and assume a conjunction of an inequality constraint with a \text{min} constraint:

\[
\text{minpath}(Y, \min(Dy)) \leftarrow \text{apath}(Y, Dy), \text{aconstant}(K), Dy < K.
\]

Then we have the following sound and complete rewriting w.r.t. \text{minpath}:

Example 10. Example 1 optimized by both inequality and \text{min}.

\[
\text{apath}(Y, \min(Dy)) \leftarrow \text{arc}(a, Y, Dy).
\text{apath}(Y, \min(Dy)) \leftarrow \text{apath}(X, Dx), \text{arc}(X, Y, Dxy), Dy = Dx + Dxy,
Dy > Dx, \text{aconstant}(K), Dy < K.
\text{minboundpath}(Y, Dy) \leftarrow \text{path}(Y, Dy).
\]

In the next example we use nonlinear rules to compute the distance between pairs of points:

Example 11. Distance between nodes expressed by non-linear rules.

\[
\text{nlp}(X, Z, Dxz) \leftarrow \text{arc}(X, Z, Dxz).
\text{nlp}(X, Z, Dxz) \leftarrow \text{nlp}(X, Y, Dxy), \text{nlp}(Y, Z, Dyz), Dxz = Dxy + Dyz.
\text{shpat}(X, Z, \min(Dxz)) \leftarrow \text{nlp}(X, Z, Dxz)
\]
We can now push the min into the recursive rules and obtain:

**Example 12.** The previous example after min pushing

\[
\text{nlp}(X, Z, \min(Dxz)) \leftarrow \text{arc}(X, Z, Dxz).
\]
\[
\text{nlp}(X, Z, \min(Dxz)) \leftarrow \text{nlp}(X, Y, Dxy), \text{nlp}(Y, Z, Dyz), \text{Dxz} = \text{Dxy} + \text{Dyz}.
\]
\[
\text{shpat}(X, Z, Dxz) \leftarrow \text{nlp}(X, Z, Dxz).
\]

The soundness and completeness of this rewriting will be proven in Section 5. For now, we would like to point out that the resulting algorithm captures properties that have been used by other well-known algorithms, and in particular by Floyd’s algorithm: for a combination of a pair of paths to have minimal length both paths must be of minimal length. The optimization technique based on pushing constraints into recursion is not limited to extrema constraints. For instance, we have pushing of inequality constraints pushing previously discussed, and the two examples described next.

In Example 13, we assume that \text{myints} contains several positive integers, and we want to find all positive integers, up to 100,000, that (i) can be factorized into the integers in \text{myints}, but (ii) they are not multiple of 5 or 8. For instance say that \text{myints} contains the following numbers \{2, 3, 10\}: then numbers such as 6, 12, and 18 are included but 20 is not. Here we use \text{div}(Z,I) that is true whenever Z is divisible by I.

**Example 13.** Products of some integers not divisible by others.

\[
\text{mult}(Y) \leftarrow \text{myints}(Y), Y > 0.
\]
\[
\text{mult}(Z) \leftarrow \text{mult}(X), \text{mult}(Y), Z = X * Y.
\]
\[
\text{somemult}(Z) \leftarrow \text{mult}(Z), Z < 100000, \neg\text{div}(Z, 5), \neg\text{div}(Z, 8).
\]

**Coalescing** The problem of coalescing temporal periods is of significant practical interest. \text{prd}(S,E) contains the initial closed periods, each represented by its start point and end point. Then in our recursive rule, we check if two periods overlap, and if so we merge them into a new period containing all their points. (The predicate \text{lovrl} determines if the period at left overlaps with one at its right.) At the end, we only keep (i.e., \text{maxprd}) periods, i.e. periods in \text{coal} that are not contained in other periods in \text{coal}.

**Example 14.** Temporal coalescing of closed periods \text{prd}(\text{Start}, \text{End}).

\[
\text{coal}(S, E) \leftarrow \text{prd}(S, E).
\]
\[
\text{coal}(S1, E2) \leftarrow \text{coal}(S1, E1), \text{coal}(S2, E2), \text{lovrl}(S1, E1, S2, E2).
\]
\[
\text{maxprd}(E, S) \leftarrow \text{coal}(E, S), \neg\text{coal}(S1, E1), \text{contain}(S1, E1, S, E)).
\]
\[
\text{lovrl}(S1, E1, S2, E2) \leftarrow S1 < S2, S2 \leq E1, E1 < E2.
\]
\[
\text{contain}(S1, E1, S2, E2) \leftarrow S1 \leq S2, E1 \geq E2.
\]
5 Completeness of Constrained Programs

While soundness of constraint pushing is always guaranteed, completeness must be proven on the basis of the specific properties of the (i) constraint used and (ii) the nature of the rules. A simple example of a program that cannot be optimized by pushing extrema constraints is given below:

Example 15. Max constraint that cannot be pushed into recursion

\begin{align*}
p(2). \\
p(5). \\
p(J_1) & \leftarrow p(J), J \leq 10, J \neq 5, J_1 = J + 2. \\
topp(max(J_1)) & \leftarrow p(J_1).
\end{align*}

In the iterated fixpoint computation of this program we obtain the following values for the argument of \( p \): 2 and 5 at the first step; then 2, 5, 4, at the second step; then 2, 4, 5, 6, at the third step, then 2, 4, 5, 6, 8, and 2, 4, 5, 6, 8, 10 at the final step, whereby the \( topp \) result is 10.

However if we perform the CPR transformation and apply the \( max \) to the facts and rules defining \( p \), we obtain 5 at the first step and 5 again the second step, since the rule produces no result. Thus the fixpoint iteration over the constrained rules terminates by producing an incorrect result: i.e. the value 5 instead of 10.

We shall now see that the reason for this problem is the inequality predicate \( \neq \), but many other constraints cause similar problems. For instance, if we replace the goal \( J \neq 5 \) with \( \text{even}(J) \), which is only true if \( J \) is a positive even integer, the fixpoint iteration still produces 2, 5, 4, 6, 8, and 10 from which the final rule again derives 10, whereas iterated fixpoint of the constrained rules produces 5.

Therefore, providing very general sufficient conditions that guarantee the completeness of our-constraint pushing techniques is the sine-qua-non for their use in practical applications. This provides the next topic for our discussion.

Implantable Constraints: Consider CPR-structured program \( P \) with constraints \( \gamma \) and recursive ICO \( T_\rho \). If the following equality holds for any interpretation \( I \):

\[
\gamma(T_\rho(I)) = \gamma(T_\rho(\gamma(I)))
\]

then we say that \( \gamma \) is implantable into \( P \)’s recursion. \( I \) is a superset of \( \gamma(I) \) and thus \( T_\rho(I) - T_\rho(\gamma(I)) \) might not be empty and contain some atoms: the implantability property assures that such atoms are eliminated by the application of \( \gamma \).

Now we have that \( T_\rho(\emptyset) = T_\rho(\gamma(\emptyset)) \), and this property provides the base case for a simple inductive argument that proves the following important property:

**Proposition 1.** If \( P \) is a CPR program with constraints \( \gamma \) implantable into recursion, then its \( \gamma \)-optimized version is complete.

We also have the following important property:

**Proposition 2.** If \( P \) is a CPR program with constraints \( \gamma \) implantable into recursion: then its \( \gamma \)-optimized version is complete.
Let \( P_\gamma \) denote the program containing the rules transformed by the implantation of the constraints (thus the extraction rule is excluded). Then we have that \( P_\gamma \) behaves as a positive program since \( T_{P_\gamma} \) has a unique least fixpoint computable as \( T_{P_\gamma}^{\uparrow \omega}(\emptyset) \). The proof follows from the definitions.

We can now provide sufficient implantability conditions for upper/lower-bound constraints and extrema. Our programs are stratified w.r.t. min and max aggregates and thus have have perfect model semantics. Programs such as that in Example 4 which only use monotonic constructs have a minimal model which is a special case of perfect model. So, let \( P \) be our program and and let \( M_P \) be its perfect model. Now, a valid ground atom of \( P \) is an atom contained in \( M_P \), and a valid instance of a rule \( r \) is simply an instantiation of \( r \) where all atoms in the rules are are contained in \( M_P \).

**Upper-bound and Lower-bound Constraints** Upper-bound and lower-bound constraints are those imposed on some arguments of the recursive predicate defined by the rules. Take for instance Example 4, where we require all paths to be shorter than a constant upper bound \( K \). Here, the \( \text{Dy} < K \), constraint is imposed on the second argument of the recursive predicate, which will be called its constrained argument.

**Definition 1.** A rule \( r \) in a program \( P \) will be said to define an ascending mapping (resp. a descending mapping) when, for each valid instance of \( r \), the cost argument in the head of the rule is \( \geq \) (resp. \( \leq \)) than each constrained argument in the body of the rule.

For instance, consider Example 4: while the inequalities shown in the body of the rule might or might not be satisfied in an arbitrary instantiation of this rule, they are all satisfied in its valid instantiations. Thus because of the goal \( \text{Dy} > \text{Dx} \), the rule defines an ascending mapping. whereby we have the following theorem:

**Proposition 3.** Upper-bound constraints are complete for recursive predicates defined by ascending mapping rules. Lower-bound constraints are complete for recursive predicates defined by descending mappings rules.

Consider now Example 10, and assume that a upper bound condition \( \text{Dxz} < 46 \) is added to the exit rule turning its third argument into a constrained one. Moreover, if we assume that all arcs have positive costs, we can conclude that: \( \text{Dxz} > \text{Dxy} \) and \( \text{Dxz} > \text{Dyz} \) and \( \min(\text{Dxz}) > \text{Dxy} \) and \( \min(\text{Dxz}) > \text{Dyz} \). Thus we have an ascending mapping whereby the optimization conditions \( \text{Dxy} < 46 \) and \( \text{Dyz} < 46 \) will be added to the body of the rule.

**Extrema Constraints** Let \( r \) be a rule in \( P \) containing min and max aggregates: the predicate argument that is maximized or minimized, will called a cost argument. Now, \( X \) and \( X' \) be two valid instances of the same predicate in \( P \), where \( X \) and \( X' \) are identical in all the no-cost arguments. Then if the cost argument of \( X \) is \( \geq \) (resp. \( \leq \)) than the cost-argument of \( X' \) we say that \( X \) inflates (resp. deflates) \( X' \).

Let \( r' \) and \( r'' \) be two valid instances of the same rule where \( r'' \) inflates the head and the recursive goals of \( r' \). Then we also say that \( r'' \) inflates \( r' \) and \( r' \) deflates \( r'' \).

We next introduce the notion of inflation (deflation) preserving rules: these are those that remain valid if we transform them by inflating (resp. deflating) their goals.
Definition 2. A recursive rule $r$ is said to be inflation-preserving if for each valid instance $r'$ of $r$, and each valid inflation of the recursive goals of $r'$ there exists a valid instance of $r$ that inflates $r'$ and contains those inflated goals.

Symmetrically, a rule is said to be deflation preserving if for each valid instance $r'$ of $r$, and each valid deflation of the recursive goals of $r'$ there exists a valid instance of $r$ that deflates $r'$ and contains those deflates goals.

Proposition 4. Let $P$ be a CRP-structured program with max constraints (resp. min constraints). If $P$ is inflation-preserving (resp. deflation-preserving) then its CRP-optimized version is complete.

Proof. Directly from the definition, we find that max constraints are implantable into max-monotonic rules. Symmetrically for min constraints.

We have seen Example 1 and several other programs with linear rules that are free of bound constraints. Those programs are both inflation-preserving and deflation-preserving. In Example 11, we can deflate either the first or the second nil goal and obtain a deflated value in the head, due to the properties of the arithmetic computation performed by the rule; this property remains true even if $D_{xy}$ or $D_{yz}$ are negative. However, no inflation/deflation preservation property holds if we replace addition with subtraction.

If we now consider the linear HCC algorithm of Example 8, we see that if we inflate/deflate the first argument of the pair $(X, Y)$ of the body, we obtain an inflated/deflated first argument in the pair $(X, Z)$ in the head. Thus, we can implant both the min and max pushing constraints.

The presence of upper-bound or lower-bound constraints limits extrema-based optimization. For instance, Example 10 establishes an ascending mapping, whereby we were able to implant the the upper-bound constraint $D_y < K$ into the recursive rule which then loose the inflation preservation property, but keep the deflation-preserving property. Thus we were still able to implant the min constraint. Finally, Example 15 shows that rules that contain numeric constraints, other than the upper-bound or lower-bound constraint, often define mappings that are not inflation/deflation preserving, and thus cannot be optimized by pushing extrema. These examples also show that dominance is quite easy to ascertain in most examples of practical interest without having actually to compute the perfect model of the program.

6 Generalized Notation for Extrema Constraints

While all constraints discussed in this paper can be expressed via the stratified negation used in Datalog programs, we will now introduce various shorthand notations of convenience that simplify the rewriting of our programs. For instance, Example 2 will be rewritten as follows using a not construct:

\[
\text{spath}(Y, Dy) \leftarrow \text{path}(Y, Dy), \neg(\text{path}(Y, Dyy), Dyy < Dy).
\]

In general, the semantics of a rule such as this, consisting of one or more positive goals and one not(...) goal is defined by:

1. replacing the not(...) goal with a uniquely named negated goal—let us call it $\neg\text{sng}_j(SV)$, where $SV$ denotes the variables that the goals not(...) share with the regular goals in the body of the rule, and
2. adding a new rule having as head \( \text{sng}_j(SV) \) and having as body the list of goals contained in \( \text{not}(\ldots) \)

We can now define the semantics of the \text{min} aggregate (and symmetrically for the \text{max}). Consider the following rule with head aggregate:

\[
\text{head}(XX, \text{min}(Y)) \leftarrow \text{BGOALS}(XX, Y, \ldots). \quad \text{eq : 1}
\]

where \( Y \) denotes the minimized cost variable, \( XX \) denotes zero or more group-by variables, and \( \text{BGOALS} \) denotes the positive goals that form the body of the original rule, which because of safety considerations, must contain \( Y \) and all the variables in \( XX \), and might contain other variables as well. Then, the above rule with \text{min} aggregate in its head is viewed as a shorthand for:

\[
\text{head}(XX, Y) \leftarrow \text{BGOALS}(X, Y), \text{not}(\text{BGOALS}(XX, Y_1, \ldots), Y_1 < Y).
\]

Thus, in this rule we have that (i) the goals in \( \text{not}(\ldots) \) are the positive goals of the original rule with the minimized variable \( Y \) renamed into \( Y_1 \), and (ii) the addition of the goal \( Y_1 < Y \).

Therefore, once we rewrite this \( \text{not}(\ldots) \) into an equivalent rule as described above, we obtain the head \( \neg \text{sng}_j(XX, Y) \) since \( XX \) and \( Y \) define the variables shared by the positive goals and the goals in \( \text{not}(\ldots) \).

As an alternative to the head-aggregate notation we allow a \text{ismin} meta-goal in the last position of the body of the rule. Thus, an equivalent notation for \text{eq : 1} is:

\[
\text{head}(X, Y) \leftarrow \text{BGOALS}(X, Y, \ldots), \text{ismin}(X, (Y)). \quad \text{eq : 2}
\]

This notation will also be generalized to allow several extrema goals at the end of the rule. For example, assuming that the maximum value of \( Y \) is found at several \( X \) values, then the \text{min} of these values can be determined by the following rule:

\[
p(X, Y) \leftarrow \text{goals}, \text{max}(X, (Y)), \text{ismin}(Y, (X)).
\]

which is simply a short hand notation for the following two rules connected by the newly introduced predicate \text{nnpn}:

\[
\text{nnpn}_1(X, \text{MaxY}) \leftarrow \text{goals}, \text{ismax}(X, (Y)). \quad \text{eq : 3}
\]
\[
p(\text{min}(X), \text{MaxY}) \leftarrow \text{nnpn}_1(X, \text{MaxY}). \quad \text{eq : 4}
\]

By this process, we can define the semantics of rules containing any number and assortment of extrema in their tails. Mixed notation where we have extrema aggregates specified in both the tail of the body and in the head of the rules will also be supported.

Finally, we allow multiple \text{min} (or \text{max}) arguments, as in the following rule where, for each value of \( Y \) we find the \text{max} pairs \( X, Z \), i.e., the pairs having \text{max} \( X \)-values, and then among the pairs that share this \text{max} \( X \) value, find and return those which are also \text{min} in their \( Z \) values. Thus the meaning of the rule shown below as \text{eq : 5} is defined by its equivalence with the rule that follows, numbered as \text{eq : 6}.

\[
\ldots \leftarrow \text{goals}, \text{ismin}(Y, (X, Z)). \quad \text{eq : 5}
\]
\[
\ldots \leftarrow \text{goals}, \text{ismin}(Y, (X)), \text{ismin}(Y, X, (Z)). \quad \text{eq : 6}
\]

11
7 Greedy Algorithms

Greedy algorithms provide a good stress test to assess the power and usefulness of the constraint pushing approach discussed here. We will start with simple algorithms before moving to more complex ones.

Let us first consider the problem of sorting the values in \( p(X) \) in an ascending order. We can begin with a naive approach that starts from arbitrary sequences of increasing values and then select the one that contains all the values. This is the slow-sort approach, of Example 16 below, that select for step \( J + 1 \) the min of the values larger than that selected at step \( J \).

**Example 16.** A slow-sort algorithm

\[
\begin{align*}
\text{sls}(1, X) & \leftarrow p(X). \\
\text{sls}(J_1, Y) & \leftarrow \text{sls}(J, X), p(Y), Y > X, J_1 = J + 1. \\
\text{minsort}(I, \text{min}(W)) & \leftarrow \text{sls}(I, W).
\end{align*}
\]

We now push the min constraint, to obtain the following program:

**Example 17.** Optimized Slow-Sort

\[
\begin{align*}
\text{sls}(1, \text{min}(W)) & \leftarrow p(W). \\
\text{sls}(J_1, \text{min}(Y)) & \leftarrow \text{sls}(J, X), p(Y), Y > X, J_1 = J + 1. \\
\text{minsort}(I, W) & \leftarrow \text{sls}(I, W).
\end{align*}
\]

This revised algorithm selects at each step the least among the values that exceed those selected so far. In the presence of suitable storage structures (e.g., B+ trees) this can achieve optimal levels of asymptotic performance.

Now, say that in the above sort algorithm instead of \( \text{sls}(J, X), p(Y), Y > X \) we have \( \text{sls}(J, X), q(X, Y), Y > X \). Thus, instead of drawing the Ys from a fixed set of \( p(Y) \) values, we use \( Y \)-values derived from the \( X \)-values using \( P(X < Y) \) (we still keep the condition \( X < Y \), to ensure to produce sequences of increasing values). Now if \( p(X, Y) \) describes the arcs and their cost generated during the transitive closure traversal of our directed graph starting from a node \( a \), then we can specify a min-cost path algorithm as follows:

**Example 18.** Min-length paths for nodes reached from node \( a \).

\[
\begin{align*}
\text{reach}(1, a, 0). \\
\text{reach}(J_1, Y, Dy) & \leftarrow \text{reach}(J, X, Dx), \text{arc}(X, Y, Dxy), D_y = Dx + Dxy, \\
& \quad \text{Dy} > Dx, J_1 = J + 1. \\
\text{fsorted}(J_1, Y, Dy) & \leftarrow \text{reach}(J_1, Y, Dy), \text{ismin}(J_1, \langle Dy, Y \rangle).
\end{align*}
\]

Now, if we push the min constraint into recursion we obtain Dijkstra’s algorithm:

**Example 19.** Reachable from point \( a \)

\[
\begin{align*}
\text{reach}(1, a, 0). \\
\text{reach}(J_1, Y, Dy) & \leftarrow \text{reach}(J, X, Dx), \text{arc}(X, Y, Dxy), D_y = Dx + Dxy, \\
& \quad \text{Dy} > Dx, J_1 = J + 1, \text{ismin}(J_1, \langle Dy, Y \rangle). \\
\text{fsorted}(J_1, Y, Dy) & \leftarrow \text{reach}(J_1, Y, Dy).
\end{align*}
\]

The condition \( \text{ismin}(J_1, \langle Dy, Y \rangle) \) asures that only one new node is generated for each new iteration. But actually at each step of Dijkstra’s algorithm we can add all the nodes having minimal distance. This can be achieved by replacing \( \text{ismin}(J_1, \langle Dy, Y \rangle) \) with \( \text{ismin}(J_1, \langle Dy \rangle) \).
**Prim's Algorithm** generates a minimum spanning tree for undirected graphs, starting from an arbitrary node that we here assume to be \( a \).

Thus, from the end node of a solved edge, we go to the next edge, avoiding previously visited nodes. The basic unconstrained algorithm generates sequences of edges by assigning to each edge the minimum of \( \text{cost}(a, i, j) \) for every \( a \in V \) and \( i, j \in V \setminus \{a\} \) adjacent to \( a \). This fast-growing tree can be restricted into a least-cost spanning tree using the constraints described in Example 20 below.

**Example 20.** Defining a minimum-cost spanning tree.

\[
\begin{align*}
\text{aseq}(1, a, a, 0). \\
\text{aseq}(J_1, Y, Z, Dyz) & \leftarrow \text{aseq}(J, _, _, _), \text{aseq}(I, X, Y, Dxy), I \leq J, \\
& \quad \text{edge}(Y, Z, Dyz), J_1 = J + 1. \\
\text{minbranch}(J_1, Y, Z, Dyz) & \leftarrow \text{aseq}(J_1, Y, Z, Dyz), \\
& \quad \text{ismin}(Z, \langle J_1 \rangle), \text{ismin}(J_1, \langle Dyz, Y \rangle).
\end{align*}
\]

Now, \( \text{ismin}(Z, \langle J_1 \rangle) \) ensures that we will never add a second incoming edge to any node since the repeated node will have a higher cost. Then among the remaining candidate edges at level \( J + 1 \), we take one that delivers the min cost. Thus, we will write the transformed rules as follows:

**Example 21.** Prim's minimum-cost spanning tree program.

\[
\begin{align*}
\text{aseq}(1, a, a, 0). \\
\text{aseq}(J_1, Y, Z, Dyz) & \leftarrow \text{aseq}(J, _, _, _), \text{aseq}(I, X, Y, Dxy), I \leq J, \text{edge}(Y, Z, Dyz), \\
& \quad J_1 = J + 1, \text{ismin}(Z, \langle J_1 \rangle), \text{ismin}(J_1, \langle Dyz, Y \rangle). \\
\text{minbranch}(J_1, Y, Z, Dyz) & \leftarrow \text{aseq}(J_1, Y, Z, Dyz).
\end{align*}
\]

An alternative formulation of Prim's algorithm can be expressed against the growing cost of the tree as follows:

**Example 22.** Using the cost of the tree so far as greedy criterion

\[
\begin{align*}
\text{aseq}(a, a, 0). \\
\text{aseq}(Y, Z, NC) & \leftarrow \text{aseq}(_, _, TC), \text{aseq}(X, Y, PC), PC \leq TC, \\
& \quad \text{edge}(Y, Z, Dyz), NC = TC + Dyz. \\
\text{minbranch}(Y, Z, FC) & \leftarrow \text{aseq}(Y, Z, FC), \\
& \quad \text{ismin}(Z, \langle FC \rangle), \text{ismin}(Y, Z, \langle FC \rangle).
\end{align*}
\]

The actual rewriting is similar to that of previous algorithms and is left to the reader as an exercise.

We have seen Example 1 and several other programs that are free of bound constraints. These programs are both dominance preserving both upward and downward. Thus, we can optimize both in the presence of min constraints and max constraints. The presence of upper-bound (resp. lower-bound) constraints cause the recursive rules to loose the upward/downward dominance. Also Example 11 define a domination-preserving mapping, even if some of the edges have negative weights. However, if we subtract rather than adding the costs of the arcs in the recursive rule, then we loose the dominance-preserving property.
For instance, the recursive rule in Example 10 establishes an ascending mapping, whereby we were able to implant the upper-bound constraint \( D_y < K \) into the recursive rule, which then came to define a min-monotonic mapping, whereby we were able to implant the min constraint too. Finally, Example 15 shows that rules that contain numeric constraints, other than the upper bound or lower-bound constraint, often define mappings that are neither max-monotonic nor min-monotonic and thus they are not optimizable by pushing extrema.

Our examples also show that dominance is quite easy to ascertain in most examples of practical interest without having actually to compute the perfect model of the program.

8 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 of tuples obtained in the last step. We can now perform the seminaive improvement which, modulo the important modification discussed next, can be used to improve the computation of the program obtained from the pushing 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 to take into account the fact that atoms produced in older steps cannot produce new atoms, and
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 the \( \gamma \) constraint whereby new atoms produced in B might not be kept and existing atoms might instead be eliminated. For constraints involving comparison with constants, for instance, besides the addition of duplicate atoms, step C will also make sure no atom that violates the comparison 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 \( (b, D_1) \) with \( D_1 \geq D \), and if this new pair is inserted then every \( (b, D_2) \) with \( D_2 < D \) is deleted from the working set. Symmetric properties hold for min aggregates.

Thus, the management of step C is quite obvious for extrema and comparison-with-constant constraints, it might be less obvious in other cases, such as the coalesce program in Example 14, where we coalesce two tuples \( T_1 \) and \( T_2 \) that overlap, and obtain a tuple that contains both \( T_1 \) and \( T_2 \). Thus the tuples that in step B were recognized in the rule of new-tuple generators are exactly those that must be eliminated in our step C, a property that allows us to reduce the cost of our computation.

9 Monotonic Aggregates

The need to use aggregates in Datalog applications has motivated much research work seeking support extrema and count aggregates in Datalog. It is important therefore that we understand the relationship between the different approaches and how they dovetail at the theoretical level and practical level. Ross and Sagiv proposed the following example to illustrate the problems caused by traditional aggregates in recursion [12]:
Example 23. A program with multiple minimal models.

\[ p(b). \]
\[ q(b). \]
\[ cq(\text{count}(X)) \leftarrow q(X). \]
\[ cp(\text{count}(Y)) \leftarrow p(Y). \]
\[ p(a) \leftarrow cq(1). \]
\[ q(a) \leftarrow cp(1). \]

For this program we have two least fixpoints, which are also minimal models. The first consists of (i): \( p(b), cp(1), q(a), q(b), cq(2) \) and the second of (ii): \( p(a), p(b), cp(2), q(a), q(b), cq(1) \).

To make things worse, the seminaive computation that is used in the typical computation of Datalog programs will instead produce a third outcome (iii): \( p(a), p(b), cp(2), q(a), q(b), cq(2) \).

In reality, (iii) is only obtained when all the rules are fired at once upon the results produced by the previous iteration. If that is not the case, because e.g., the work is distributed among multiple processors, the final result could be any of the three outcomes, and the number of possible outcomes is bound to explode once we move to more complex programs and larger databases.

At the core of the solution proposed in [7, 8] is the observation that continuous count \( m\text{count} \) which returns all the integers up to the cardinality of the set, is in fact monotonic w.r.t. set containment, whereby the standard least-fixpoint semantics holds for programs that use the continuous count \( m\text{count} \) in recursive rules. In fact, if in Example 23 we replace the \( \text{count} \) with \( m\text{count} \) we obtain as unique least fixpoint and minimal model: \( p(a), p(b), cp(1), cp(2), q(a), q(b), cq(1), cq(2) \). Naturally, enumerating all integers till the max can be inefficient, but this can be solved by adding a max constraint in the extraction rule, as illustrated by this example taken from [12].

Example 24. Join once you see three friends but then keep counting.

\[ \text{attend}(X) \leftarrow \text{organizer}(X). \]
\[ \text{attend}(Y) \leftarrow \text{cntfriends}(Y, N), N \geq 3. \]
\[ \text{cntfriends}(Y, m\text{count}(X)) \leftarrow \text{attend}(X), \text{friend}(X, Y). \]
\[ f\text{count}(Y, \max(CC)) \leftarrow \text{cntfriends}(Y, CC), \]

The use of \( m\text{count} \) 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 [11].

The recursive rules in the above program are max monotonic. Thus we can push the max aggregate from the head of the \( m\text{count} \) to that of \( \text{cntfriends} \), where observing that the max of \( m\text{count} \) is the traditional count aggregate whereby we obtain the following CPR-optimized program:

Example 25. The CRP optimized version of Example 24.

\[ \text{attend}(X) \leftarrow \text{organizer}(X). \]
\[ \text{attend}(Y) \leftarrow \text{cntfriends}(Y, N), N \geq 3. \]
\[ \text{cntfriends}(Y, \text{count}(X)) \leftarrow \text{attend}(X), \text{friend}(X, Y). \]
\[ f\text{count}(Y, CC) \leftarrow \text{cntfriends}(Y, CC). \]
In this example, we had a max-monotonic program, whereby \texttt{mcount} was replaced with \texttt{count} by pushing the max into recursion. However, consider again Example 23 after replacing \texttt{count} with \texttt{mcount}: we have seen that the resulting monotonic program has a unique least fixpoint. Thus, if we add an additional rule, say \texttt{fcp(X) \leftarrow cp(X), ismax(X)}), we obtain a stratified program that produces \texttt{fcp(2)}. However, the resulting program is not max-monotonic, and thus \texttt{mcount} in the body cannot be replaced by \texttt{count}.

The observation that from traditional non-monotonic aggregates we can derive a monotonic version of theirs by letting them return an interval of integer values rather than a single value led Majurian et al. [7] to define a monotonic max which respectively returns all the positive integers up to a top value (and symmetrically for the min). In fact the max of two numbers so represented as intervals is simply their union which contains all positive integers up to the max of those two numbers. This approach introduces monotonic max and min that can be freely used in recursive programs under the least-fixpoint semantics of set-containment lattices. But in order to get efficient implementation the continuous max and min must be replaced by the traditional max and min, an optimization that is only possible under conditions that are basically equivalent to those that support the CPR optimization supported here. If efficient implementation is a sine-qua-non, then monotonic extrema do not support more applications than the CPR optimization discussed here, which is also preferable for many other aspects, including simpler recursive rules. On the other hand, monotonic count remains indispensable since many important applications cannot be supported without its use in recursive rules.

\textbf{Optimizer-Initiated CPR} Optimization based on pushing extrema is also desirable for programs that originally contain no extrema, and as such it might be initiated by the system. For instance, returning to Example 4, we see that the original constraint rule

\[
\texttt{boundpath(Y) \leftarrow path(Y, Dy), isconstant(K), Dy < K.}
\]

can and should be improved with the addition of the min goal \texttt{ismin(Y, Dy)} whose addition does not change the values returned by \texttt{boundpath(Y)}, but prompt a CPR optimization that expedites the computation.

A similar observation can be made for the party rules in Example 24, where instead of the current \texttt{fcount} rule we can have the following final attend rule:

\[
\texttt{fattend(Y) \leftarrow attend(Y), cntfriends(Y, CC), ismax(Y, \langle CC \rangle).}
\]

Here too the final two goals above can be added or removed without affecting the values produced in \texttt{fattend}, but their inclusion triggers a CPR optimization that expedites the computation.

\section{Related Work}

Supporting aggregates in recursion is an old and difficult problem which has been the topic of much previous research work, which previously had primarily focused on providing a formal semantics that could accommodate the non-monotonic nature of the aggregates. In particular Mumick et al. [10] discussed programs that are stratified w.r.t. aggregates operators and
proved that a perfect model exist for these programs. Then, Kemp and Stuckey defined extensions of the well-founded semantics to programs with aggregation, and later showed that programs with aggregates might have multiple and counter-intuitive stable models [6] (such as the one of Example 23). The notion of cost-monotonic extrema aggregates was introduced by Ganguly et al. [3], using perfect models and well-founded semantics, whereas Greco et al. [4] showed that their use to express greedy algorithms requires the don’t-care non-determinism of the stable-model semantics provided by the choice construct.

A general approach to the problem was due to Ross and Sagiv [12] which proposes a semantics for monotonic aggregates based on using different lattices for different aggregates. Meet and bag-monotonic are two special cases of Ross and Sagiv’s approach [13]. However, Van Gelder in 1993 [18] 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 next twenty years. 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 [1, 17, 15, 19]. Thus UCLA researchers, first proposed the novel solution to the monotonic aggregate problem outlined in Section 9, and then proceeded to demonstrate their many uses in applications, and their scalable implementations for (i) workstations [16], (ii) multicore systems [21, 20], and (iii) Spark-based clusters [2].

The referenced works suggest that the 20-year old challenge of providing a simple semantics for aggregates in recursion has largely been solved, and will be critical in making possible the massively parallel logic-based systems that were the holy grail pursued by the Fifth Generation Computer Systems project, MCC, ECRC and many others research projects [9]. The exultation for this success is moderated by the findings proposed in this paper, that for extrema there is a simpler solution that rather than trying to define a formal semantics for programs defined by the recursive rules with extrema, instead uses the formal semantics of stratification and then optimizes the operational iterated fixpoint by applying the extrema to the rules implementing this operational semantics. This approach leads to programs that are easier for the user to write and also simpler for the system to optimize, as demonstrated by the new release of DeALS.

11 Conclusion

The ability to specify complex algorithms by adding postconditions to simpler recursive predicates is of potential interest in the conceptual design of algorithms and in their concrete implementation using the DeALS system. Indeed, we are currently investigating new applications of the approach proposed in this paper in the design and implementation of graph algorithms and knowledge discovery algorithms.

References


