Declarative Algorithms in Datalog with Extrema: their formal semantics simplified

Carlo Zaniolo[1], Mohan Yang[1], Matteo Interlandi[2], Ariyam Das[1], Alex Shkapsky[1], Tyson Condie[1]

[1]*University of California, Los Angeles*
(*e-mail:* {zaniolo,yang,ariyam,shkapsky,tcondie}@cs.ucla.edu)

[2]*Microsoft*
(*e-mail:* matteo.interlandi@microsoft.com)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Pre-mappable ($\mathcal{P}$*reM*) extrema constraints in recursive Datalog programs enable concise declarative formulations for classical algorithms (Zaniolo et al. 2017). The programs expressing these algorithms have formal non-monotonic semantics with efficient and scalable support on multiple platforms (Shkapsky et al. 2016; Yang et al. 2017). However, proving $\mathcal{P}$*reM* for different programs can be challenging for programmers. Thus, in this paper, we introduce simple templates that allow users and compilers to verify with ease that programs with extrema have the $\mathcal{P}$*reM* property, along with the rigorous semantics and the efficient and scalable implementation associated with it. We thus obtain simple declarative formulation for classical algorithms under (i) perfect model semantics and (ii) stable model semantics.

1 Introduction

A growing interest in scalable data analytics on BigData 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). These recent developments are significant because a fundamental reason that motivated early Datalog researchers to adopt a bottom-up execution model was its affinity with that of relational DBMS which had demonstrated major gains in performance and scalability via data parallelization as far back as the 80s—albeit only for non-recursive queries (Teradata 1983). But the hope that Datalog could deliver superior levels of performance and scalability for more powerful applications, took a very long time to realize. Initially, massive parallelization of Datalog focused on supporting complex algorithms under answer-set semantics (Alviano and Leone 2016), but more recently several parallel Datalog systems were proposed that support efficiently polynomial-time algorithms by allowing aggregates in recursive queries (Seo et al. 2013; Wang et al. 2015; Shkapsky et al. 2016; Yang et al. 2017). Among these, the UCLA BigDatalog project is the only one that provides a formal semantics for these programs using the notion of Pre-Mappability ($\mathcal{P}$*reM*). This notion, introduced in (Zaniolo et al. 2017), establishes a direct link between specialized fixpoint optimizations yielding superior performance and scalability with powerful declarative semantics that enables the concise expression of a wide range of algorithms. At the system level, a highly optimized fixpoint implementation, including special Resilient Distributed Datasets (RDDs) (Zaharia et al. 2012), allows BigDatalog (Shkapsky et al. 2016) on Apache Spark to surpass the performance and scalability of other Datalog systems, and even outperform GraphX on graph applications—the very domain for which GraphX was designed for (Gonzalez et al. 2014). At the application level, $\mathscr{P}reM$ allows us to express declaratively a wide range of algorithms, including those discussed in this paper and those discussed in (Condie et al. 2018). The validity of the $\mathscr{P}reM$ property for those examples was proven directly using the formal definition of the property (Zaniolo et al. 2017). However as we have now gained experience with more applications, a clear need has emerged for simple criteria and templates that make it easy to verify the $\mathscr{P}reM$ property, particularly since we want a larger number of programmers to benefit from these advances. Providing such criteria presents the first objective of this paper, which also explores the close relationships that exists between the stable model semantics of programs that use extrema inside the recursive definition, and the perfect models of equivalent programs where extrema are applied outside the recursive definition i.e., as post-conditions in stratified programs.

The rest of this paper is organized as follows: Section 2 introduces the concept of pre-mappable constraints, while Section 3 presents simple conditions that can be used to prove $\mathscr{P}reM$. Section 4 formally explains how these conditions can be mapped to relational constraints. Section 5 presents the stable model semantics for $\mathscr{P}reM$ programs. Related work and conclusions presented in Sections 6 and 7 bring the paper to a closing.

2 Pre-Mappable Constraints

This section contains a brief summary of results from (Zaniolo et al. 2017). In the Example 1, below, rule $\mathtt{r}_3$ computes the least distance from node a to the remaining nodes in a directed graph by applying the constraint is_min((Y),D) to the pth(Y,D) atoms produced by rules $\mathtt{r}_1$ and $\mathtt{r}_2$.
Example 1 (*Finding the minimal distance of nodes from* a)

$$\begin{aligned}
\mathtt{r}_1 : \mathtt{pth(Y,D)} \leftarrow \quad & \mathtt{arc(a,Y,D)}.\\
\mathtt{r}_2 : \mathtt{pth(Y,D)} \leftarrow \quad & \mathtt{pth(X,Dx),arc(X,Y,Dxy),D = Dx + Dxy},\\
\mathtt{r}_3 : \mathtt{qpth(Y,D)} \leftarrow \quad & \mathtt{pth(Y,D),is\_min((Y),D)}.
\end{aligned}$$

In our goal is_min((Y),D) , we will refer to (Y) as the *group-by* argument (group-by arguments can consist of zero or more variables), and D is the *cost* argument variable (cost argument consist of a single variable). This goal is expressing the *constraint* that a pair (Y,D) is acceptable only if no other pair exists having the same (Y) and a smaller value of D. Thus the formal meaning of our constraint is defined by replacing $\mathtt{r}_3$ with $\mathtt{r}_4$:

$$\mathtt{r}_4 : \mathtt{qpth(Y,D)} \leftarrow \quad \mathtt{pth(Y,D)}, \neg \mathtt{smlr\_pth(Y,D)}.$$

where the goal is_min((Y),D) has been revised into smlr_pth(Y,D), and the latter is defined as:

$$\mathtt{r}_5 : \mathtt{smlr\_pth(Y,D)} \leftarrow \quad \mathtt{pth(Y,D),smlr\_pth(Y,D1),D1 < D}.$$

The program consisting of rules $\mathtt{r}_1$, $\mathtt{r}_2$, $\mathtt{r}_4$, and $\mathtt{r}_5$ is stratified w.r.t. negation, with pth occupying the lower strata and qpth occupying the higher strata. Thus, the program has a perfect model semantics, but its computation using the iterated fixpoint procedure can be quite inefficient and actually unsafe when the graph contains cycles. To solve these problems, (Zaniolo et al. 2017) introduces the $\mathscr{P}reM$ condition, where qpth can be computed safely and efficiently by pre-mapping the min goal into the rules defining pth, whereby the following program is obtained:

Example 2 (*The endo-min version of Example 1*)

$$\begin{aligned}
\mathtt{r}'_1 : \mathtt{pth(Y,D)} \leftarrow \quad & \mathtt{arc(a,Y,D),is\_min((Y),D)}.\\
\mathtt{r}'_2 : \mathtt{pth(Y,D)} \leftarrow \quad & \mathtt{pth(X,Dx),arc(X,Y,Dxy),D = Dx + Dxy,is\_min((Y),D)}.\\
\mathtt{r}'_3 : \mathtt{qpth(Y,D)} \leftarrow \quad & \mathtt{pth(Y,D)}.
\end{aligned}$$

Thus we have seen two versions of this program: the program in Example 2, with min in recursion will be called the *endo-min version*, whereas the original program shown in Example 1 will be called its *exo-min version*. The $\mathscr{P}reM$ condition defined next establishes a clear semantics relationship between the two versions: the exo-min version defines its abstract perfect-model semantics, whereas the endo-min version defines its optimized concrete semantics that assures more efficient computation and termination in situations, e.g. cycles in the underlying graph, where the iterated fixpoint computation would not terminate. However, in the course of our discussion it will become clear that, often, we might want to let users interface directly with the endo-min version because (i) it is more natural for programmers, and (ii) it has a well-defined total stable model semantics.

Fixpoint and $\mathscr{P}reM$ Constraints. Now we will consider stratified programs, such as those of Example 1, having a perfect model semantics computed by strata. At the lower stratum we find the minimal model of the rules defined by (i) interpreted predicates, such as comparison and arithmetic predicates, and (ii) positive rules such as `r₁` and `r₂` defining the `pth` predicate. If T is the Immediate Consequence Operator (ICO) for this positive program, then its minimal model is defined by the least-fixpoint of T which can be computed by $T^{\uparrow\omega}(\emptyset)$ (a.k.a the naive fixpoint computation). The subset of this minimal model obtained by removing from $T^{\uparrow\omega}(\emptyset)$ all the `pth(Y,D)` that do not satisfy the constraint $\gamma =$ `is_min((Y),D)` will be called the *extreme subset* of $T^{\uparrow\omega}(\emptyset)$ defined by γ. Then, the perfect model of the program in our example, can be obtained by adding to $T^{\uparrow\omega}(\emptyset)$ the `pth` atoms obtained by simply copying `pth` atoms under the name `qpth`. We next formally define the notion of $\mathscr{P}reM$ (Zaniolo et al. 2017) which allows us to transform programs such as those of Example 1 into that of Example 2 where the min or max constraints have been pushed (or more precisely transferred) into the recursive rules.

Definition 1 (*The $\mathscr{P}reM$ Property*)
In a given Datalog program, let P be the set of its rules defining a (set of mutually) recursive predicate(s). 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 T then: $\gamma(I) = \gamma(T(I)) = \gamma(T(\gamma(I)))$. Now, let T_γ denote the application of T followed by γ, i.e., $T_\gamma(I) = \gamma(T(I))$. If I is a fixpoint for T and $I' = \gamma(I)$, then the above equality can be rewritten as: $I' = \gamma(I) = \gamma(T(\gamma(I))) = T_\gamma(I')$.

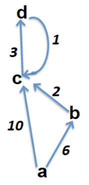
Thus, when γ is $\mathscr{P}reM$, the fact that I is a fixpoint for T implies that $I' = \gamma(I)$ is a fixpoint for $T_\gamma(I)$. We will next describe many programs of practical interest where the transfer of constraints under $\mathscr{P}reM$ produces optimized programs that are safe and terminating even when the original programs were not. 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. As proven in (Zaniolo et al. 2017), the fixpoint $T_\gamma^{\uparrow n}(\emptyset)$ so obtained is in fact a minimal fixpoint for T_γ:

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

Therefore, when the $\mathscr{P}reM$ property holds, declarative exo-min (or exo-max) programs are transformed into endo-min (or endo-max) programs having highly optimized seminaive-fixpoint based operational semantics. For instance, consider Example 1 on the following facts:

Example 3 (`arc` *facts for Example 1*)

```
arc(a,b,6).   arc(a,c,10).
arc(b,c,2).   arc(c,d,3).
arc(d,c,1).
```



In this example, while the computation of $T^{\uparrow\omega}(\emptyset)$ will never terminate, the computation of $T_{\gamma}^{\uparrow n}(\emptyset)$ produces the following `pth` atoms at each step of the computation (underlined atoms denote atoms that are eliminated at the next step of the fixpoint computation since they are no longer competitive).

Example 4 (*Computing* $T_{\gamma}^{\uparrow n}(\emptyset)$ *for Example 1 on facts in Example 3*)
Step 1: `pth(b,6)`, $\underline{\texttt{pth(c,10)}}$
Step 2: `pth(b,6)`, `pth(c, 8)`, $\underline{\texttt{pth(d,13).}}$
Step 3: `pth(b,6)`, `pth(c, 8)`, `pth(d,11)`
Step 4: `pth(b,6)`, `pth(c, 8)`, `pth(d,11)`,

Thus, we have derived the `pth` atoms `pth(b,6)`, `pth(c,8)`, `pth(d,11)` which, along with the given `arc` atom above, constitute the extreme subset of $T^{\uparrow\omega}(\emptyset)$ since the $\mathscr{P}$*reM* property holds for the recursive rules in Example 1. These atoms, renamed `qpth` plus the atoms in $T^{\uparrow\omega}(\emptyset)$ then constitute the perfect model for the original program. We now leave the summary of (Zaniolo et al. 2017) and present new results.

Determining that $\mathscr{P}$*reM holds.* For all its obvious advantages, detecting and proving that $\mathscr{P}$*reM* holds in the program at hand might not be simple even in simple applications. Thus, we now provide methods and tools that greatly simplify this task. We need to focus on the recursive rules, since $\mathscr{P}$*reM* always holds for exit rules (since these are used at the first step of $T^{\uparrow\omega}(\emptyset)$, i.e. they are applied to the empty set). For recursive rules the problem can be illustrated by adding an additional goal to the rule to express the pushing of the extrema constraint into the argument of the ICO. For instance, the recursive rule of our Example 2 should be re-written as:

$$\mathtt{r}_2' : \mathtt{pth(Y,D)} \leftarrow \quad \mathtt{pth(X,Dx)},^{\backslash \mathtt{is\_min((X),Dx)}/}\mathtt{arc(X,Y,Dxy)}, \mathtt{D} = \mathtt{Dx} + \mathtt{Dxy}, \mathtt{is\_min((Y),D)}.$$

Thus we observe that the final `is_min` goal minimizes the second argument of `pth` w.r.t. the first one. The additional goal that should be added after `pth(X,Dx)` is$^{\backslash \mathtt{is\_min((X),Dx)}/}$. Basically, $\mathscr{P}$*reM* is satisfied when this new goal dropped into the rule does not change the mapping from the recursive predicates in the body of the rules to the head of the rule, defined by the original exo-min rule. This “drop-in" rewriting helps understanding $\mathscr{P}$*reM* at the intuitive level, and illustrates the rigorous treatment provided in the next section.

3 Declarative Algorithms by Simple Pre-mappability Patterns

A variety of algorithms can be expressed by recursive programs that use extrema aggregates, which satisfy the $\mathscr{P}$*reM* conditions, and in this section, we introduce simple criteria and formal conditions that will make it easy for programmers to prove that $\mathscr{P}$*reM* holds in their programs. As we will see, there are basically two general techniques for detecting and proving $\mathscr{P}$*reM*. One is based on *existential cost-comparisons* in the rule, the other on extrema goals and the *multivalued dependencies* established by the rule. Since it is trivial to map between the endo-min or endo-max versions of programs and their exo-min or exo-max counterparts, we will normally use the former versions that are often more concise and closer to procedural algorithms.

Naive Sorting. Here our fact base contains atoms such as `item(PartNo,Cost)` that state the `Cost` of a part identified by its `PartNo`. Now the following endo-min program orders those parts by assigning a positive sequence number to them, as follows:

Example 5 (*Ordering parts by their price in an ascending sequence.*)

```
asc(0,nil,0).
asc(J1,Part,Val1) ←  asc(J,_,Val),item(Part,Val1),J1 = J+1,
                     Val1 > Val,is_min((J1),Val1).
```

To prove $\mathscr{P}$*reM* we drop-in the goal `is_min((J),Val)` and determine its effects:

```
asc(J1,Part,Val1) ←  asc(J,_,Val)\is_min(J),Val)/,item(Part,Val1),
                     Val1 > Val,J1 = J+1,is_min((J1),(Val1)).
```

Observe that the conjunct `asc(J,_,Val),item(Part,Val1),Val1 > Val` simply states the following: take each `item(Part,Val1)` if there exists a `Val` in `asc(J,_,Val)` such that `Val1 > Val`. But this statement can be re-expressed as: take each `item(Part,Val1)` whose `Val1` is larger than the least of the values `Val` appearing in `asc(J,_,Val)`. Thus the dropping of `\is_min(J),Val)/` does not change the mapping defined by our rule, and $\mathscr{P}$*reM* is thus proven. This reasoning based on existential comparisons allows us to determine $\mathscr{P}$*reM* for many interesting examples, including several greedy algorithms.

Huffman Encoding. We generate nodes of the Huffman tree, where nodes are represented by a three-argument predicate `huff`, where its first argument is the frequency value, its second argument is either a symbol or a reference to the left subtree, and its third argument is either `nil` or a reference to the right subtree.

Example 6 (*Huffman Encoding*)

```
huff(Sym,Val,nil) ←  freq(Sym,Val).
np(0,0,0).
np(PR,L,R) ←  np(OR,_,PR),huff(L,_,_),L > PR,huff(R,_,_),R > L,is_min((OR),R).
huff(V,NL,NR) ←  np(_,NL,NR),huff(Lval,NL,_),huff(Rval,NR,_),V = Lval+Rval.
```

The first recursive rule, i.e. the one with head `np(PR,L,R)`, states that, following a `np` goal with right value `PR`, we now have a pair `(L,R)`. Then the first exit rule in our program transforms the given symbol+frequency information into the bottom nodes of the tree. For example, from the facts "`freq(a,5). freq(b,9). freq(c,12). freq(d,13). freq(e,16). freq(f,45).`" the first exit rule produces: `huff(5,a,nil),huff(9,b,nil),huff(12,c,nil), huff(13,d,nil), huff(16,e,nil),huff(45,f,nil)`.

Now, the proper computation begins with the second exit rule, i.e., `np(0,0,0)`. From that the first recursive rule produces `np(0,5,9)`, and the second rule produces atom `huff(14,5,9)` (since 5+9 = 14). Next, from the atom `np(0,5,9)` the first recursive rule produces `np(9,12,13)`, whereby we obtain `huff(25,12,13)`. We next obtain the triple `np(13,14,16)` and the new node `huff(30,14,16)` and so on until the complete tree is constructed.

The proof of the $\mathscr{P}$*reM* property for the first recursive rule is a variation of the argument used for the previous example. In fact, say that we check $\mathscr{P}$*reM* by dropping-in the additional goal `is_min((OR),PR)` after the `np(OR,_,PR)` goal in the first rule. Now `R` must satisfy the constraint that it is the least value that follows some `L` that follows some `PR`, i.e. that follows the least of such `PR` values. Thus the goal dropped into the recursive rule defining `np` does not change its meaning and $\mathscr{P}$*reM* is proven.

4 Inferring $\mathscr{P}reM$ from Relational Dependencies and Extrema

It is often the case that the $\mathscr{P}reM$ property can be inferred by considering the properties of extrema goals and the multivalued dependencies that hold in the equivalent relational DB views of the (function-free) atoms in the rule body. In a given interpretation I, we say that Rq is a relational view of a predicate $\mathtt{q}(\mathtt{x}_1,\ldots,\mathtt{x}_n)$ if defined as $Rq = \{(x_1,\ldots,x_n)|\mathtt{q}(\mathtt{x}_1,\ldots,\mathtt{x}_n) \in I\}$. The next examples are meant to illustrate how the well-developed theory of functional dependencies (FDs) and multivalued dependencies (MVDs) (Maier 1983), taught in the relational schema design chapter of DB textbooks, will simplify the task of determining $\mathscr{P}reM$.

In a nutshell, the MVDs $Y \twoheadrightarrow X$ and $Y \twoheadrightarrow Z$ hold in the relation $R(X,Y,Z)$ obtained as the natural join (i.e. the join on their common attribute Y) of two relations $R1(X,Y)$ and $R2(Y,Z)$. Also the FD $X \rightarrow Z$ holds in R when no two tuples exist in R having the same X-value and a different Z-value. Complex properties of relations can be easily proven by arrow-based inferences that exploit the formal properties of FDs and MVDs including those used in this paper. that are listed below: (Here, X, Y, Z, and W are subsets of the attributes of the given relation; moreover we use the notation X,Y to denote the union of X and Y.)

Replication: if $X \rightarrow Y$ then $X \twoheadrightarrow Y$.
FD Augmentation: if $X \rightarrow Y$ then $X,Z \rightarrow Y$.
MVD Augmentation: If $X \twoheadrightarrow Y$ and $Z \subseteq W$, then $X,W \twoheadrightarrow Y,Z$.
Mixed Transitivity: If $Y \twoheadrightarrow Z$ and $Z \rightarrow X$, then $Y \rightarrow X - Z$.

Bill of Materials. A classical recursive application for traditional databases is Bill of Materials (BOM), where we have the DAG of parts-subparts, `assbl(Part,Subpart,Qty)` describing how a given part is assembled using various subparts, each in a given quantity. Not all subparts are assembled, basic parts are instead supplied by external suppliers in a given number of days, as per the facts `basic(Part,Days)`. Simple assemblies, such as bicycles, can be put together the very same day in which the last basic part arrives. Thus, the time needed to produce the assembly is the maximum number of days required by the basic parts it uses.

$$
\begin{aligned}
\mathtt{r}_0: \mathtt{deliv(Part,Days)} \leftarrow\ & \mathtt{basic(Part,Days), is\_max((Part),Days)}.\\
\mathtt{r}_1: \mathtt{deliv(Part,Days)} \leftarrow\ & \mathtt{deliv(Sub,Days), assbl(Part,Sub), is\_max((Part),Days)}.
\end{aligned}
$$

Now, to determine if $\mathscr{P}reM$ holds, we must study the mapping of our rule transformed by the drop-in goal as follows:

$$
\begin{aligned}
&\mathtt{deliv(Part,Days)} \leftarrow\\
&\qquad \mathtt{deliv(Sub,Days)}, ^{\backslash \mathtt{is\_max((Sub),Days)}/} \mathtt{assbl(Part,Sub), is\_max((Part),Days)}.
\end{aligned}
$$

Thus we want to prove that the dropped in goal does not change the mapping defined by this rule. In our proof we will refer to `is_max((Sub),Days)` as the *drop-in* constraint and to `is_max((Part),Days)` as the original constraint. We can exploit the following properties:

1. min and max aggregates enforce constraints that are special kinds of FDs, and
2. these min/max FDs are applied to the relation instances defined by the bodies of the rules, which are the natural joins of the two tables that are the relational views of `deliv(Sub,Days)` and `assbl(Part,Sub)`.

Therefore, let $R(Sub,Days,Part)$ be the natural join of the binary relations $Rdeliv(Sub,Days)$ and $Rassbl(Part,Sub)$ which are the relational views of `deliv(Sub,Days)` and `assbl(Part,Sub)`. Therefore, the following MVDs hold in R: $Sub \twoheadrightarrow Days$ and $Sub \twoheadrightarrow Part$.

Now, let R' be the relation obtained from R by enforcing the constraint $\texttt{is\_max((Part),Days)}$, i.e. by eliminating from R the tuples have the same *Part* value as another tuple, but a smaller or equal value for *Days*. As a result of enforcing the max constraint upon R, we have that the FD $Part \rightarrow Days$ holds in R', along with the well-known properties of FDs. But in addition to this intra-relational constraint (the FD is defined w.r.t. the other tuples in R'), the following inter-relational max-constraint holds:

$$\text{If } (\texttt{s},\texttt{d},\texttt{p}) \in R', \text{ then } (\texttt{s},\texttt{d},\texttt{p}) \in R \text{ and } \nexists(\texttt{s}',\texttt{d}',\texttt{p}) \in R \text{ s.t. } \texttt{d}' > \texttt{d}.$$

We will use the notation: $Part \xrightarrow{max:R} Days$ to denote that both these constraints hold in R', and refer to this as a *max FD constraint*. The reasons for using this arrow-based notation follow from the following properties that are proven in the Appendix (see Theorem 4 and its corollary).

1. If $Sub \twoheadrightarrow Days$ and $Sub \twoheadrightarrow Part$ hold in R they also hold in R';
2. If $Sub \twoheadrightarrow Part$ and $Part \xrightarrow{max:R} Days$ hold in R' then $Sub \xrightarrow{max:R} Days$ also holds in R'.

The second property generalizes the *mixed-transitivity* property of MVDs and FDs.

Now, $\mathscr{P}reM$ follows directly from $Sub \xrightarrow{max:R} Days$. In fact, while the imposition of the drop-in constraint$^{\backslash\texttt{is_max((Sub),Days)}/}$might in general cause the exclusion from $\texttt{Deliv}$ of pairs $(\texttt{Part},\texttt{Days})$ that violate this constraint, this exclusion will have no effect upon R' which contains no such pairs given that $Sub \xrightarrow{max:R} Days$ holds in R'. Thus we have the simple and general rule to decide $\mathscr{P}reM$: the property holds if the drop-in constraint is implied by the MVD and the original constraint. We will use and refine this simple rule in the remaining examples.

Connected Components in a Graph. Starting from the exo-min program is more natural in some applications. For instance in this application, we have an undirected graph, where an edge connecting, say, $\texttt{a}$ and $\texttt{b}$ is represented by the pairs $\texttt{edge(a,b)}$ and $\texttt{edge(b,a)}$. Then, we can start by computing the transitive closure of this graph, i.e., all pairs of connected nodes.

Example 7 (*Connected Components in an undirected graph.*)

```
cc(X,X) ←  edge(X,_).
cc(X,Z) ←  cc(X,Y),edge(Z,Y).
```

The transitive closure so obtained is quite redundant since a clique of N nodes is represented by $2 \times N \times N$ pairs. A more concise representation consists in selecting as a representative of a clique a particular node and then just store N pairs that represent the arcs going from the selected node to every node of the clique, including itself. For instance, if we represent the nodes by integers, we can select the node with the lowest integer as the representative for its clique. For that, we can use the following rule to find if a node $\texttt{Z}$ belongs to a clique named $\texttt{X}$.

```
inclique(Z,X) ←  cc(X,Z),is_min((Z),X).
```

Then, the next question that arises naturally is whether this computation can be optimized by transferring $\texttt{is\_min((Z),X)}$ into the rules of Example 7, a question that can be answered by checking $\mathscr{P}reM$ using the following rule:

$$\texttt{cc(X,Z)} \leftarrow \quad \texttt{cc(X,Y)},^{\backslash\texttt{is\_min((Y),X)}/}\texttt{edge(Z,Y)},\texttt{is\_min((Z),X)}.$$

Here we can consider $R(\texttt{X},\texttt{Y},\texttt{Z})$ and observe the following MVDs hold: $\texttt{Y} \twoheadrightarrow \texttt{X}$ and $\texttt{Y} \twoheadrightarrow \texttt{Z}$. Thus the imposed constraint $\texttt{Z} \xrightarrow{\texttt{min:R}} \texttt{X}$ together with the second MVD implies that $\texttt{Y} \xrightarrow{\texttt{min:R}} \texttt{X}$ also holds. Thus, once the first constraint is enforced, enforcing the second does not change the mapping. Thus, $\mathscr{P}reM$ holds and the re-written program is as follows (we omit the no-op goal $\texttt{is\_min((X),X)}$ from the exit rule).

$$\begin{array}{ll}\texttt{cc(X,X)} \leftarrow & \texttt{edge(X,\_)}.\\ \texttt{cc(X,Z)} \leftarrow & \texttt{cc(X,Y),edge(Z,Y),is\_min((Z),X)}.\end{array}$$

Minimal Distances in Directed Graph. Let us now return to our Example 2, whose had the following recursive rule and drop-in goal we will rewrite as follows:

$$\texttt{path(Y,Dx+Dxy)} \leftarrow \quad \texttt{path(X,Dx)}^{\backslash\texttt{is\_min((X),Dx)}/}\texttt{,arc(X,Y,Dxy),is\_min((Y),Dx+Dxy)}.$$

In order to prove $\mathscr{P}reM$ we must exploit both the MVD due to the fact that this rule computes the join of the relational views of $\texttt{path(X,Dx)}$ and $\texttt{arc(X,Y,Dxy)}$ and the min-distributive properties of the sum function. We will say that the function $F(X,Y)$ is distributive w.r.t. min when $\min(F(X,Y)) = F(\min(X),\min(Y))$. Thus in a relation $R(X,W,Z)$ selecting the tuple that, for a given value of W, has the least value of $X+Y$ is equivalent to selecting the tuple that contains both the min value of X and the min value of Y, and such a tuple is assured to exist because $W \twoheadrightarrow X$ and $W \twoheadrightarrow Z$ hold.

Theorem 2
Let $R(X,W,Z)$ be a relation where $W \twoheadrightarrow X$ and $W \twoheadrightarrow Z$, and let $F(X,Z)$ be a min-distributive function. Then, the constraints $W \xrightarrow{min:R} X$ and $W \xrightarrow{min:R} Z$ hold in the relation R' constructed by enforcing the constraint $\texttt{is\_min}((W),F(X,Z))$ upon R.
Proof
Because of the MVD, if x and z are minimal X-values and Z-values associated with the same W-value w, R must also contain a tuple (x,w,z). Since F is min-distributive this is the tuple which produces the min value of $F(x,z)$ for the given w, and as such is the only tuple with W-value w that is also present in R', which therefore also satisfies the constraints $W \xrightarrow{min:R} X$ and $W \xrightarrow{min:R} Z$.
□
We will make use of this theorem, and the dual one that holds for max-distributive function, in the next examples where we also exploit the *augmentation* property of min/max FDs, that basically states that min or max FDs remain valid if we augment their left sides by adding additional attributes. For instance since R' in the previous example satisfies $W \xrightarrow{min:R} Z$, it also satisfies $X,W \xrightarrow{min:R} Z$, where the notation X,W denotes the union of X and W. In fact $X,W \xrightarrow{min:R} Z$ is the conjunct of the following two properties (i) the FD $X,W \rightarrow Z$ holds, and (ii) no tuple with the same X and W values and a smaller Z value exists in the original R. Now, (i) is true because of the augmentation property of FDs, and (ii) is true because of the valid constraint that no tuple exists with the same W-value and a smaller Z-value. Therefore, with R the natural join of the relational views of $\texttt{path(X,Dx)}$ and $\texttt{arc(X,Y,Dxy)}$, let us take the following steps:

Step 1 [Find MVDs in R]: $\texttt{X} \twoheadrightarrow \texttt{Dx}$ and $\texttt{X} \twoheadrightarrow \texttt{Y,Dxy}$.
Step 2 [Determine min-FDs in table R' obtained from R by application of $\texttt{is\_min((Y),Dx+Dxy)}$]: R' satisfies: $\texttt{Y} \xrightarrow{\texttt{min:R}} \texttt{Dx+Dxy}$.
Step 3 [Use augmentation to align MVDs and min-FDs to distribute min]: $\texttt{X,Y} \twoheadrightarrow \texttt{Dx}$ and $\texttt{X,Y} \twoheadrightarrow \texttt{Y,Dxy}$ hold in R. Also: $\texttt{X,Y} \xrightarrow{\texttt{min:R}} \texttt{Dx+Dxy}$ holds in R'.
Step 4 [Distribute min-FDs, because of the constraints established in Step3]: $\texttt{X,Y} \xrightarrow{\texttt{min:R}} \texttt{Dx}$ and $\texttt{X,Y} \xrightarrow{\texttt{min:R}} \texttt{Dxy}$ hold in R'.

Step 5 [Use augmentation to align the MVDs in Step 1 and the min-FDs in Step 4 for mixed transitivity and apply it]: From the second MVD in Step 1 and the first min-FD in Step 4 we infer: $\mathtt{X} \twoheadrightarrow \mathtt{X,Y,Dxy}$ and $\mathtt{X,Y,Dxy} \xrightarrow{\text{min:R}} \mathtt{Dx}$. From these we infer: $\mathtt{X} \xrightarrow{\text{min:R}} \mathtt{Dx}$.

While dealing with the properties of FDs and MVDs might represent a challenge for naive programmers, the pattern established by these examples can be mechanically applied to other problems by naive users and compilers that verify $\mathscr{P}$*reM*. The next example illustrates this point.

Max Probable Path. Assume that the cost argument associated with a directed arc represents a probability (e.g., probability that road connecting two nodes is practicable). Then, we can compute the max probability between nodes as follows:

Example 8 (*Max Path Probability*)

```
r0: ppath(X,Y,Vxy) ←    arc(X,Y,Dxy), is_max((X,Y),Vxy).
r1: ppath(X,Z,V) ←      ppath(X,Y,Vxy), ppath(Y,Z,Vyz),
                        V = Vxy * Vyz, is_max((X,Z),V).
r2: maxprop(X,Y,V) ←    ppath(X,Y,V).
```

Since cost arguments represent probabilities, their values range between zero and one, whereby the max product of two costs is equal to the product of their max. Now, the programmer must check that $\mathscr{P}$*reM* holds on the following recursive rule:

$$\mathtt{ppath(X,Z,V)} \leftarrow \quad \mathtt{ppath(X,Y,Vxy)}^{\backslash\mathtt{is\_max((X,Y),Vxy)}/}, \mathtt{ppath(Y,Z,Vyz)}^{\backslash\mathtt{is\_max((Y,Z),Vyz)}/},$$
$$\mathtt{V = Vxy * Vyz, is\_max((X,Z),V)}.$$

So let $R(\mathtt{X,Y,Vxy,Z,Vyz})$ be the natural join (i.e., on the column Y) of the relational views of `ppath(X,Y,Vxy)` and `ppath(Y,Z,Vyz)`. We now have:

Step 1 [Find MVDs in R]: $\mathtt{Y} \twoheadrightarrow \mathtt{X,Vxy}$ and $\mathtt{Y} \twoheadrightarrow \mathtt{Z,Vyz}$.
Step 2 [Find max-FDs from `is_max((X,Z),Vxy * Vyz)`]: $\mathtt{X,Z} \xrightarrow{\text{max:R}} \mathtt{Vxy * Vyz}$ holds in R'.
Step 3 [Align MVDs and max-FDs]: $\mathtt{X,Y,Z} \twoheadrightarrow \mathtt{X,Vxy}$ and $\mathtt{X,Y,Z} \twoheadrightarrow \mathtt{Z,Vyz}$ hold in R.
Also: $\mathtt{X,Y,Z} \xrightarrow{\text{max:R}} \mathtt{Vxy * Vyz}$ holds in R'.
Step 4 [Distribute max-FDs]: $\mathtt{X,Y,Z} \xrightarrow{\text{max:R}} \mathtt{Vxy}$ and $\mathtt{X,Y,Z} \xrightarrow{\text{max:R}} \mathtt{Vyz}$ hold in R'.
Step 5 [Align MVDs from Step 1 and max-FDs from Step 4 and apply mixed transitivity]: From the second MVD in Step 1 and the first max-FD in Step 4, we infer: $\mathtt{X,Y} \twoheadrightarrow \mathtt{X,Y,Z,Vyz}$ and $\mathtt{X,Y,Z,Vyz} \xrightarrow{\text{max:R}} \mathtt{Vxy}$. From these two we infer: $\mathtt{X,Y} \xrightarrow{\text{max:R}} \mathtt{Vxy}$. Symmetrically, from $\mathtt{Y,Z} \twoheadrightarrow \mathtt{X,Y,Z,Vxy}$ and $\mathtt{X,Y,Z,Vxy} \xrightarrow{\text{max:R}} \mathtt{Vyz}$, we infer $\mathtt{Y,Z} \xrightarrow{\text{max:R}} \mathtt{Vyz}$.

Minimum Cost Spanning Tree. We have an undirected graph, whose edges are represented by `cedge(X,Y,C)` where $\mathtt{X} < \mathtt{Y}$, and C is the cost of the edge. We proceed as in Example 7 except that among the edges that connect different connected components, we select one having minimum cost (but larger than those used in previous steps).

Example 9 (*Minimum spanning tree à la Kruskal*)

```
kr(0,X,X,0) ←     cedge(X,_,_).
cc(Z,Y) ←         cc(X,Y), kr(_,X,Z,_), is_min((Z),(Y)).
kr(C,X,Y,NC) ←    kr(OC,_,_,C), cedge(X,Y,NC), NC > C,
                  cc(NX,X), cc(NY,Y), NY<>NX, is_min((C),NC).
```

For the recursive rule defining `cc`, $\mathscr{P}reM$ can be proven in the same way it was proven for Example 7. For the rule defining `kr`, $\mathscr{P}reM$ can be proven by dropping in the goal `is_min((OC),C)` after the first goal. But this will not modify the mapping defined by the rule, since an `NC` value is larger than some value of `C` iff it is larger than the least of those `C`-values.

5 Stable Models of $\mathscr{P}reM$ Programs

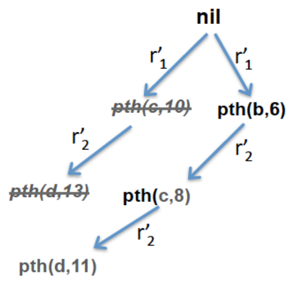
The examples in previous sections illustrate that endo-min/max programs can often appeal to the intuition of the programmer better than their original exo-min/max versions which they optimize. It is only natural therefore that we ask the question whether the endo-min version of programs have their own formal semantics. As we shall see next, the answer is yes, for most $\mathscr{P}reM$ programs of practical interest, including all the programs we have considered so far: these programs have total stable model semantics. We now summarize this important result for which the proof is given in the appendix.

Cost Atom Derivation and Stability. The *Cost-Atom Derivation Graph* (CAD-graph in short) is a labeled graph that represents the derivation of cost atoms in $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$. It is defined as follows:

1. For each instance of a recursive rule r used in $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ with head H and recursive goals G_j, $j \geq 1$, there is an arc labeled r:j from G_j to H.
2. For each instance of exit rule r used in $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ with head H there is an arc labeled r:0 from the distinguished goal atom **nil** to H.

The CAD-graph for the min distance example of Section 2 is shown in Example 10. Since the recursive rule in our example only contains one cost goal, the ":j" part of the arc label is redundant and has been omitted in our graph. Competing atoms are shown vertically aligned. Nodes at distance i from **nil** are those produced at the i^{th} step of the computation. The source of each arc shows the cost atom from step $i-1$ while its label shows the rule used. Thus, the CAD-graph succinctly describes the computation of $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$.

Example 10 (*Cost-atom derivation table&graph for Example 3.*)



`arc(a,b,6).arc(a,c,10).arc(b,c,2).arc(c,d,3).arc(d,c,1).`

Step 1: `pth(b,6)`, $\underline{\texttt{pth(c,10).}}$

Step 2: `pth(b,6)`, `pth(c, 8)`, $\underline{\texttt{pth(d,13).}}$

Step 3: `pth(b,6)`, `pth(c, 8)`, `pth(d,11)`

Step 4: `pth(b,6)`, `pth(c, 8)`, `pth(d,11)`,

To the left of this CAD-graph we show the associated CAD-table, which displays the cost atoms obtained at each step without showing the rules used in the derivation. Competing cost atoms are displayed in the same column, where a horizontal bar between the old value and the new one shows that the value has decreased and has been superseded by the new value.

Stable Models in $\mathscr{P}reM$ Programs. The CAD-graph for a given $\mathscr{P}reM$ program can be used to decide whether the program has a stable model. In particular, the following modification of

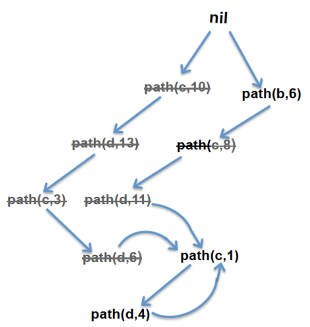
Example 2, illustrates, by its CAD-graph, a $\mathscr{P}reM$ program where its minimal model $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ is not a stable model.

Example 11 (*arcs departing from* `a`)

$$
\begin{array}{lll}
(\mathtt{r}_1) & \mathtt{path(Y,D)} \leftarrow & \mathtt{arc(a,Y,D), is\_min((Y),D).} \\
(\mathtt{r}_2) & \mathtt{path(Y,LD)} \leftarrow & \mathtt{path(X,Dx), arc(X,Y,Dxy), D = Dx + Dxy, lb(D,LD), is\_min((Y),LD).} \\
(\mathtt{r}_3) & \mathtt{lb(X,X)} \leftarrow & \mathtt{X >= 1.} \\
(\mathtt{r}_4) & \mathtt{lb(X,1)} \leftarrow & \mathtt{X < 1.}
\end{array}
$$

$\mathscr{P}reM$ still holds for this program and the $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ computation produces the CAD-graph below[1]. Observe that while the programs of these two examples are basically the same, and both satisfy the $\mathscr{P}reM$ property, their `arc` fact bases are different. Indeed while in both cases the fact base contain a cycle, the arcs in the second fact base have negative lengths, and this causes the resulting CAD-graph, displayed in Example 12, below, to have a directed cycle.

Example 12 (*Derivation table&graph for Example 11*)



`arc(a,b,6). arc(a,c,10). arc(b,c,2).`
`arc(c,d,3). arc(d,c,-10).`

Step1: $\mathtt{path(b,6)}$, $\underline{\mathtt{path(c,10)}}$

Step2: $\mathtt{path(b,6)}$, $\underline{\mathtt{path(c,\ 8)}}$, $\underline{\mathtt{path(d,13)}}$

Step3: $\mathtt{path(b,6)}$, $\underline{\mathtt{path(c,\ 3)}}$, $\underline{\mathtt{path(d,11)}}$

Step5: $\mathtt{path(b,6)}$, $\mathtt{path(c,\ 1)}$, $\underline{\mathtt{path(d,\ 6)}}$

Step6: $\mathtt{path(b,6)}$, $\mathtt{path(c,\ 1)}$, $\mathtt{path(d,\ 4)}$

Step7: $\mathtt{path(b,6)}$, $\mathtt{path(c,\ 1)}$, $\mathtt{path(d,\ 4)}$

As stated by the following theorem (proven in the Appendix), this is an important difference:

Theorem 3
If P is an endo-min (resp. endo-max) $\mathscr{P}reM$ program whose CAD-graph is free of cycles, then $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ is a stable model for P.

Acyclic CAD-graphs. The absence of cycles in CAD-graph is assured in the following two situations that occur frequently in actual $\mathscr{P}reM$ programs:

C1 **Acyclic fact base.** This is the situation where the fact base can be viewed as a graph that is free of cycles. For instance, in our running example, if we remove the arc leading back from `d` to `c` the resulting fact base is free of cycles and thus a stable model exists for both the programs in Example 2 and Example 11.

C2 **Inflationary/deflationary path.** A path in the CAD graph will be called inflationary (resp. deflationary) if the cost of each cost-atom in the path is larger (resp. smaller) than that of each of its competing predecessor. Inflationary (resp. deflationary) paths occur when all arcs in the underlying `arc` fact-base are, respectively, positive (resp. negative). Cycles can occur when we have paths of both kinds, although this is a necessary condition but not a sufficient one. For instance in Example 12 if we replace $\mathtt{arc(d,c,-10)}$ with $\mathtt{arc(d,c,-2)}$ we still have that all paths are inflationary.

[1] For simplicity we do not show the labels of the arcs, since it is clear the arcs departing from **nil** are labeled r_1' and the others are labeled r_2'.

In all examples of practical interest, including those discussed in the previous sections, we found that either condition C1, or condition C2 or both hold, and they are actually easy to verify once $\mathscr{P}reM$ is verified. Therefore, we now have a practical test to assure that stable models exist for Datalog programs expressing declaratively classical algorithms—stable models that can be computed very efficiently via $T_{\gamma}^{\uparrow n}(\emptyset) = T_{\gamma}^{\uparrow n+1}(\emptyset)$ (Shkapsky et al. 2016).

6 Related Work

Supporting aggregates in recursion is a difficult problem which has been the topic of much previous research work. Several 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 problem of optimizing programs with extrema by early pruning of non-relevant facts was studied in (Ganguly et al. 1991) and (Sudarshan and Ramakrishnan 1991) exploiting different optimization conditions, and the notion of cost-monotonic extrema aggregates was introduced by (Ganguly et al. 1995), using perfect models and well-founded semantics. More recently, (Furfaro et al. 2002) devised an algorithm for pushing max and min constraints into recursion while preserving query equivalence under certain specific monotonicity assumptions. 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, whereby each aggregate will then define a monotonic mapping in its specialized lattice—an approach that was not without practical limitations (Van Gelder 1993). A similar idea (with similar limitations) was more recently proposed in (Swift and Warren 2010) and in several other works e.g., (Zhou et al. 2010; Zhou et al. 2015) in the context of tabling in Prolog relying on 3-valued well-founded model semantics.

Researchers working on Answer Set Programming (ASP) have also shown much interest in the benefits that aggregates offer in logic programs (Pelov et al. 2007; Pontelli et al. 2004; Son and Pontelli 2007). A significant semantic analysis of the problem was proposed by (Gelfond and Zhang 2014). In (Alviano and Leone 2016), various aggregates subclasses are identified that are tractable in polynomial-time, which are also supported in DLV system.

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 builds on the monotonic count and sum proposed in (Mazuran et al. 2013) and provides the foundation of the efficient and scalable systems discussed in (Shkapsky et al. 2015; Yang et al. 2015; Shkapsky et al. 2016; Yang et al. 2017).

7 Conclusion and Future Work

The problem of supporting aggregates in recursive logic programs has long been the focus of much research because of the great opportunities and very difficult challenges it presents. Significant opportunities in BigData applications were recently demonstrated by (i) the superior levels of scalability and performance achieved by systems that support Datalog with aggregates (Shkapsky et al. 2016; Yang et al. 2017), and (ii) the broad spectrum of declarative algorithms that can be expressed using aggregates in recursion, including those discussed in (Shkapsky et al. 2016; Yang et al. 2017; Zaniolo et al. 2017) and in this paper. The most difficult challenges posed by aggregates in recursion are caused by their non-monotonic nature but they go beyond the much-studied problem of providing a rigorous formal semantics since they include the requirements that such formal semantics (a) must be conducive to efficient implementation and (b) must be simple enough to be intuitive to and usable by everyday programmers. The $\mathscr{P}reM$ property studied in this paper can answer successfully these challenges by moving the state-of-the-art forward significantly on the three fronts of efficient implementation, formal semantics, and usability. In this paper, we have achieved solid advances on the second front by providing general conditions that assure that $\mathscr{P}reM$ endo-min/max programs have stable models and those models are equivalent to the perfect models of their exo-min/max counterparts. On the third front, the usability of $\mathscr{P}reM$ has been improved significantly by the discovery of simple templates (including the novel ones based on FDs and MVDs) that allow users and compilers to easily verify pre-mappability. This progress brings us closer to the goal of supporting declarative BigData algorithms by Datalog with aggregates, but much work remains to be done in proving the generality of the approach. Algorithms that use count and sum, viewed as maximized version of monotonic progressive count and sum, were discussed in (Zaniolo et al. 2017) and shown to satisfy $\mathscr{P}reM$. These require simple generalizations of our MVD/FD-templates, that were not covered in this paper because of space limitations. A second class of algorithms, such as temporal coalescing, can be expressed under $\mathscr{P}reM$ by using specialized aggregates that will require new templates to validate them. Finally, we have procedural algorithms that cannot be directly mapped into declarative algorithms satisfying $\mathscr{P}reM$—although that might be possible under revised formulations for those algorithms, as this paper has done for greedy algorithms.

References

ALVIANO, M. AND LEONE, N. 2016. On the properties of gz-aggregates in answer set programming. In *IJCAI*. 4105–4109.

AREF, M., TEN CATE, B., GREEN, T. J., KIMELFELD, B., ET AL. 2015. Design and implementation of the LogicBlox system. In *SIGMOD*. ACM, 1371–1382.

CONDIE, T., DAS, A., INTERLANDI, M., SHKAPSKY, A., YANG, M., AND ZAIONO, C. 2018. Scaling-up reasoning and advanced analytics on bigdata. Tech. rep.

FURFARO, F., GRECO, S., GANGULY, S., AND ZANIOLO, C. 2002. Pushing extrema aggregates to optimize logic queries. *Inf. Syst. 27,* 5 (July), 321–343.

GANGULY, S., GRECO, S., AND ZANIOLO, C. 1991. Minimum and maximum predicates in logic programming. In *PODS*. 154–163.

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. *TPLP 14,* 4-5, 587–601.

GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. 2014. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*. 599–613.

KEMP, D. B. AND STUCKEY, P. J. 1991. Semantics of logic programs with aggregates. In *ISLP*. 387–401.

MAIER, D. 1983. *The Theory of Relational Databases*. Computer Science Press.

MAZURAN, M., SERRA, E., AND ZANIOLO, C. 2013. A declarative extension of Horn clauses, and its significance for Datalog and its applications. *TPLP 13,* 4-5, 609–623.

MUMICK, I. S., PIRAHESH, H., AND RAMAKRISHNAN, R. 1990. The magic of duplicates and aggregates. In *VLDB*. Morgan Kaufmann Publishers Inc., 264–277.

PELOV, N., DENECKER, M., AND BRUYNOOGHE, M. 2007. Well-founded and stable semantics of logic programs with aggregates. *TPLP 7,* 3, 301–353.

PONTELLI, E., SON, T. C., AND ELKABANI, I. 2004. Smodels with clp: A treatment of aggregates in ASP. In *LPNMR*. 356–360.

ROSS, K. A. AND SAGIV, Y. 1992. Monotonic aggregation in deductive databases. In *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. *PVLDB 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 *SIGMOD*. ACM, 1135–1149.

SHKAPSKY, A., YANG, M., AND ZANIOLO, C. 2015. Optimizing recursive queries with monotonic aggregates in DeALS. In *ICDE*. IEEE, 867–878.

SHKAPSKY, A., ZENG, K., AND ZANIOLO, C. 2013. Graph queries in a next-generation Datalog system. *PVLDB 6,* 12, 1258–1261.

SON, T. C. AND PONTELLI, E. 2007. A constructive semantic characterization of aggregates in answer set programming. *TPLP 7,* 3, 355–375.

SUDARSHAN, S. AND RAMAKRISHNAN, R. 1991. Aggregation and relevance in deductive databases. In *VLDB*. 501–511.

SWIFT, T. AND WARREN, D. S. 2010. Tabling with answer subsumption: Implementation, applications and performance. In *JELIA*. 300–312.

TERADATA. 1983. *Data Bases Computer Concepts and Facilities*. Teradata: Document Number CO2-0001-00.

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. *PVLDB 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 *IEEE Big Data*. 251–260.

ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX Association, 2–2.

ZANIOLO, C., YANG, M., DAS, A., SHKAPSKY, A., CONDIE, T., AND INTERLANDI, M. 2017. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *TPLP 17,* 5-6, 1048–1065.

ZHOU, N.-F., BARTÁK, R., AND DOVIER, A. 2015. Planning as tabled logic programming. *TPLP 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 *ICTAI '10*. Washington, DC, USA, 213–218.

Appendix A MVDs and Extrema FD Constraints

Assuming that with n and m denoting positive integers let us consider a relation $R(X_1,\ldots,X_n,Y_1,\ldots,Y_m,C)$, and let $R'(X_1,\ldots,X_n,Y_1,\ldots,Y_m,C)$ be the relation obtained from R by enforcing the min-FD $Y \xrightarrow{min:R} C$, i.e., by deleting the tuples in R that violate the constraint $\texttt{is\_min}((Y),C)$. We have the following theorem:

Theorem 4
If $X_1,\ldots,X_n \twoheadrightarrow Y_1,\ldots,Y_m$ holds in $R(X_1,\ldots,X_n,Y_1,\ldots,Y_m,C)$ and $R'(X_1,\ldots,X_n,Y_1,\ldots,Y_m,C)$ is obtained from R by enforcing $Y_1,\ldots,Y_m \xrightarrow{min:R} C$, then $X_1,\ldots,X_n \xrightarrow{min:R'} C$ holds in R'.

Proof
It suffices to prove that R' cannot contain two tuples $ta = (x_1,\ldots,x_n,y_1,\ldots,y_m,c_1)$ and $tb = (x_1,\ldots,x_n,y'_1,\ldots,y'_m,c_2)$ where $c_2 > c_1$. In fact if these two tuples are in R' they must also be in R where the fact that $X_1,\ldots,X_n \twoheadrightarrow Y_1,\ldots,Y_m$ holds implies that tuples $tc = (x_1\ldots x_n,y_1,\ldots y_m,c_2)$ and $td = (x_1\ldots x_n,y'_1,\ldots y'_m,c_1)$ exist in R as well. But the enforcement of the min constraint to derive R' implies that tc and tb will be deleted, and the only remaining tuples are those with the least C value of c_1. $\Box$

Corollary. As a direct consequence of this theorem we have that since $X \to C$ now holds in R' so do the original MVDs $X \twoheadrightarrow C$ and $X \twoheadrightarrow Y$.

Appendix B Stable Models for Min and Max Programs

In Section 2 of this paper, we established a clear relationship between $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ and the perfect model of the exo-min version of the program. Here we seek to elucidate the strong relationship that exists between $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ and the stable model of an endo-min (endo-max) program.

B.1 Stability Definition

We next characterize the semantics of our endo-min and endo-max programs by the stable-model semantics of equivalent programs that use negation, and then provide efficient algorithms to verify such semantics. We begin by recasting and endo-min or endo-max programs into equivalent programs that use negation, called the negation based-equivalents (*n.b.e*) of the original programs. Also, if P the original program its *n.b.e* will be denoted as *nbe*(P). Take for instance the following program consisting of our original Example 2 and a set of five facts.

Example 13 (*arcs departing from* a)

```
r6 : pth(Y,D) ←   arc(a,Y,D),is_min((Y),D).
r7 : pth(Y,D) ←   pth(X,Dx),arc(X,Y,Dxy),D = Dx + Dxy,is_min((Y),D).
arc(a,b,6).       arc(a,c,10)
arc(b,c,2).       arc(c,d,3).
arc(d,c,1).
```

Observe that the rules of this program have the $\mathscr{P}$*reM* property and, for the given set of facts, the computation of $T_\gamma^{\uparrow n}(\emptyset)$ converges after a finite number of step producing the following pth atoms: { pth(b,6), pth(c,8), pth(d,11)}. The *n.b.e* for above program is as follows:

Example 14 (*The n.b.e. of the program in Example 13*)

```
r8 : cpth(Y,D) ←          arc(a,Y,D).
r9 : cpth(Y,D) ←          pth(X,Dx),arc(X,Y,Dxy),D = Dx+Dxy.
r10 : smlr_pth(Y,D) ←     cpth(Y,D),cpth(Y,D1),D1 < D.
r11 : pth(Y,D) ←          cpth(Y,D),¬smlr_pth(Y,D).
arc(a,b,6).               arc(a,c,10)
arc(b,c,2).               arc(c,d,3).
arc(d,c,1).
```

Thus instead of is_min((Y),D), for each cost atom produced in r_8 and r_9 we use in r_{10} to identify those that, for the same Y value have a smaller cost and exclude them from pth in r_{11}. Observe that *nbe*(P) contains all the predicates of P, which will be called *core* predicates, plus new ones that will be called *ancillary* predicates. In the example at hand, pth and arc are core predicates and smlr_pth and cph are ancillary predicates. Now, with I an interpretation of P, *nbe*(I) will denote the interpretation of I obtained by extending I using the rules in *nbe*(P) defining ancillary predicates. Thus, for the example at hand, given {pth(b,6), pth(c,8), pth(d,11)} we use rule r_8 and r_9 to derive cpth(b,6), cpth(c,10), cpth(c,8), cpth(d,11), cpth(c,12).

Then, the interpretation I of an endo-min program P will be said to be a stable model for P if *nbe*(I) is a stable model for *nbe*(P). For instance, given the following core atoms:

$M1$ = {arc(a,b,6), arc(a,c,10), arc(b,c,2), arc(c,d,3), arc(d,c,1),
pth(b,6), pth(c,8), pth(d,11)},

the *n.b.e* expansion of these atoms, i.e. $M2 = nbe(M1)$, is as follows:

$M2$ = {arc(a,b,6), arc(a,c,10), arc(b,c,2), arc(c,d,3), arc(d,c,1),
cpth(b,6), cpth(c,10), cpth(c,8), cpth(d,11), cpth(c,12)
smlr_pth(c,10), smlr_pth(c,12),
pth(b,6), pth(c,8), pth(d,11)}.

Therefore according to our definition, $M1$ will be said to be stable model for the program in Example 13 iff $M2$ is a stable model for the program in Example 14.

We have used here Example 2 for illustrative purposes, but it is clear that the definition of *nbe*(P) and the fact that its stable model defines the stable model of the original P, holds for any endo-min program P (and conversely for endo-max programs). Dual notions and definitions hold for endo-max programs.

B.2 Verifying Stability

We will next address the two related problems of (i) verifying when a given interpretation is a stable model for the program at hand and (ii) identifying endo-min and endo-max programs for which a stable model is guaranteed to exists.

In general, to verify that a given interpretation M is stable for an endo-min program P, we apply the *stability transformation*, which builds the redux of *ground*(P) w.r.t. M, denoted as *ground*$_{\mathbf{M}}(P)$, by (a) first removing all the rules with goal $\neg q$ if $q \in M$, and then (b) removing all negated goals from the remaining rules. If the the least fixpoint of the positive *ground*(P) so constructed is equal to M, then this is a stable model for P. However for endo-min and endo-max programs we do not need to instantiate the rules in P since the same mapping as that defined by *ground*(P) can be obtained by transforming P as follows: (i) leave the positive rules in P

unchanged, (ii) assert a new fact q_inM for each atom $\mathtt{q} \in \mathtt{M}$, and (iii) replace each negated goal $\neg$q in the rules of P by a goal $\neg$q_inM. The program so obtained will be called the *symbolic redux of P*.

For instance, the symbolic redux of for our example can constructed by (i) leaving $\mathtt{r}_{10}$ unchanged, since it has no negated goal, and by (ii) adding a fact smlr_pth_inM(Y,D) for each instance of smlr_pth(Y,D) on the candidate model M, and (iii) replacing the goal $\neg$smlr_pth(Y,D) in $\mathtt{r}_{11}$ with $\neg$smlr_pth_inM(Y,D). Therefore, we obtaining the following negation-stratified program, where we omit listing rule $\mathtt{r}_{10}$ which no longer plays a role in the computation of pth and cpth. For brevity, we also omit listing the arc facts.

Example 15 (*Checking stability*)

$\mathtt{r}_8$: cpth(Y,D) $\leftarrow$	arc(a,Y,D).
$\mathtt{r}_9$: cpth(Y,D) $\leftarrow$	pth(X,Dx),arc(X,Y,Dxy),D = Dx + Dxy.
$\mathtt{r}_{10}$: ... $\leftarrow$	...
$\mathtt{r}_{11}$: pth(Y,D) $\leftarrow$	cpth(Y,D),$\neg$smlr_pth_inM(Y,D).
smlr_pth_inM(c,10).	smlr_pth_inM(c,12).

We can now compute the perfect model for this stratified program, and find that it is equal to $M2$. Thus $M2$ is a stable model for Example 15 and $M1$ is therefore a stable model for Example 14

B.3 Direct Stability Verification

We can optimize the rules of of Example 15 to produce a direct and faster computation for the core pth atoms, and then use the other rules to derive the ancillary predicate from the core atoms. In fact if the set of core atoms so obtained is exactly the same as those in the original $M1$ then we are guaranteed that we will also obtain back the same $M2$ proving stability.

Optimized Stability Verification. A further optimization in the computation of pth can be obtained from the stratified program in Example 14 by merging the computation of $\mathtt{r}_8$ and $\mathtt{r}_9$ with $\mathtt{r}_{11}$ producing the following rules that can deliver a direct verification for core predicates of the program at hand (for brevity, we do not list the arc facts):

Example 16 (*Optimized Verification Program*)

$\mathtt{r}'_8$: pth(X,D) $\leftarrow$	arc(a,Y,D),$\neg$smlr_inM(X,D)
$\mathtt{r}'_9$: pth(X,D) $\leftarrow$	pth(X,Dx),arc(X,Y,Dxy),D = Dx + Dxy,$\neg$smlr_pth_inM(X,D).
smlr_pth_inM(c,10).	smlr_pth_inM(c,12).

The optimized stability verification just discussed can be use to verify that an arbitrary interpretation I is a stable model for the given program. However consider the situation where I is *extreme*, i.e. it that satisfies the condition $\gamma(I) = I$, where were γ stands for min or max. Observe now that in all $\mathscr{P}reM$ programs $M = T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ is extreme since $\gamma(T^{\uparrow n}(\emptyset)) = T^{\uparrow n}(\emptyset)$.

Then we have that the stability check for $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ a $\mathscr{P}reM$ program P can be performed by a positive program P', called *direct verification program*, constructed as follows:

1. make the following *positive assertions*: assert as facts all the atoms in M under a new distinguished name, whereby an atom of M having name q will be renamed q_inM, and
2. in the rules of P, replace every negated goals $\neg$smlr_q_inM(...) with the positive goal q_inM(...).

Thus, the direct verification program for Example 16 is as follows (`arc` facts omitted):

$$\begin{aligned}
\mathtt{r}''_8 : \mathtt{pth(Y,D)} \leftarrow\ & \mathtt{arc(a,Y,D),pth\_in\_M(X,D)}\\
\mathtt{r}''_9 : \mathtt{pth(Y,D)} \leftarrow\ & \mathtt{pth(X,Dx),arc(X,Y,Dxy),D = Dx + Dxy,pth\_in\_M(X,D).}
\end{aligned}$$

$\mathtt{pth\_inM(b,6).\ pth\_inM(c,8).\ pth\_inM(d,11).}$

To prove the equivalence of this program to that of Example 16, observe that $\neg\mathtt{smlr\_pth\_inM(X,D)}$ is satisfied in the following three cases:

(i) there is a positive assertion $(\mathtt{pth\_inM(X,D)}$. or
(ii) there is a positive assertion $\mathtt{pth\_in(X,D')}$ where $\mathtt{D'} > \mathtt{D}$, or
(iii) there is no positive assertion having the same first argument `x` (the group-by argument).

Now, when verifying Now, when verifying $M = T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ which is extreme e have that (ii) and (ii) are always false, and there only remains condition (i) which checks that $\mathtt{pth\_inM(X,D)} \in \mathtt{M}$. This conclude our proof: the direct verification program, which is positive, can be used to check stability of $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ for a $\mathscr{P}reM$ program.

Since the direct stability verification program for a candidate model M uses the original exo-rules modified with the addition of the final goal that discards atoms that are not in M, we will denote by T_M the ICO of the direct stability verification program, which can be defined as $T_M(I) = M \cap T(I)$, where T is the ICO for the original exo-min program. These properties and the transformations we have illustrated with the help of our running example hold for any exo-min or exo-max $\mathscr{P}reM$ program. Thus we can state the following:

Theorem 5
If P is an endo-min or endo-max program with ICO T_γ, then $M = T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ is a stable model for P iff $T_M^{\uparrow\omega}(\emptyset) = M$.

For instance, for our example, the direct stability-verification computation for the candidate stable model $M = \mathtt{pth(b,6),\ \ pth(c,8),\ \ pth(d,11)}$ derives the following atoms:

Step1: $\mathtt{pth(b,6)}$,
Step2: $\mathtt{pth(b,6),\ \ pth(c,8)}$,
Step3: $\mathtt{pth(b,6),\ \ pth(c,8),\ \ pth(d,11)}$
Step4: $\mathtt{pth(b,6),\ \ pth(c,8),\ \ pth(d,11)}$

Thus, we obtain back our original M which therefore is a stable model for Example 2.

The direct verification algorithm derives extreme atoms using extreme rules, where an atom is called *extreme* if it belongs to $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$. Also: an exit rule instance will be called extreme if its head is extreme, and a recursive rule instance will be called extreme if its heads and each of its cost goals are extreme. Now, extreme rule instances play an important role in the notion of extreme derivation discussed next.

B.4 Assuring Model Stability in $\mathscr{P}reM$ Programs

Thus, we would like to give the recursive definition of derivation for a cost atom, and also define the derivation-level at which a cost-atom is produced.

Definition 2 (*Derivations and Extreme Derivations*)

(i) Each instance of an exit rule used in $T_\gamma(\emptyset)$ defines a level-1 derivation for its head cost atom, and

(ii) If there is an instance of a recursive rule r having as head the cost atom H, which belongs to $T_\gamma^{\uparrow j}(\emptyset)$ but not to $T_\gamma^{\uparrow j-1}(\emptyset)$, then this rule instance together with the derivations of each of its coast goals, defines a j^{th} level derivation for H.

(iii) A derivation of cost atom will be called extreme if each rule instance used in the derivation is extreme.

Then we have the following theorem.

Theorem 6
Let T be the ICO of an endo-min or endo-max program P having the $\mathscr{P}reM$ property. Then $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ is stable model for P iff there exists an extreme derivation for each atom in $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$.

Proof
Observe that every extreme derivation for a cost atom is also a valid derivation for the same cost atom in the direct verification algorithm, computing $T_M^{\uparrow\omega}(\emptyset)$. Conversely, every valid derivation of a cost atom in $T_M^{\uparrow\omega}(\emptyset)$ an extreme derivation for $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$. $\square$

Examples 10 and 12 show that the validity of $\mathscr{P}reM$ does not guarantee that the minimal model produced by $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ is stable, inasmuch as different fact bases might or might not produce a stable model, and the direct Verification Algorithm provides simple and efficient tool for verifying stability. However, in most applications we must and can do even better and support programmers who want to know a priori that their declarative algorithm will work correctly for the application and data set it was designed for. Therefore, we will next provide simple conditions that guaranteed that $\mathscr{P}reM$ programs actually have total stable models. A very useful sufficient conditions is that the CAD-graph does not contain any cycle as state in Theorem 3, for which we now provide the proof.

Theorem 7 (Same as Theorem 3)
If P is an endo-min (resp. endo-max) $\mathscr{P}reM$ program whose CAD-graph is free of cycles, then $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ is a stable model for P.

Proof
Let C be a cost atom in $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$. Since there can be several derivations for C, consider a final one, i.e. one that has level j and there is no derivation of C at level larger than j. Thus C has been produced by the instance of a rule r, let us call it $inst(r)$ having C as its head. If some goal of $inst(r)$ is not extreme, let us replace it in $i(r)$ by its final competitor: the final competitor in its class must have a level which is less than j otherwise j cannot be final derivation level for C. Moreover, this final competitor is extreme and, because of $\mathscr{P}reM$, the instance of r obtained by replacing the non-extreme goal with this extreme one is true. By applying this process to each non-extreme goal in $i(r)$ we obtain an extreme instance of r that has C has its head and where each of its goals is extreme. Moreover, these extreme goals have a final level that is less than j. Given that all our derivations are finite and are free of cycles, the repeated application of this inductive reasoning takes us down to the bottom level delivering an extreme derivation for C. Thus there is an extreme derivation for each cost atom in $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$. $\square$

As discussed in Section 5, the acyclicity of the CAD-graph is guaranteed when the underlying fact base is acyclic or all the paths in the CAD-graph are either inflationary or deflationary. Since these conditions hold for all declarative algorithms discussed in this paper, their exo-min or exo-max programs have stable models.