Declarative BigData Algorithms via Aggregates and Relational Database Dependencies

Carlo Zaniolo, Mohan Yang, Matteo Interlandi,
Ariyam Das, Alexander Shkapsky, Tyson Condie
{zaniolo,yang,ariyam,shkapsky,tcondie}@cs.ucla.edu,
matteo.interlandi@microsoft.com

University of California at Los Angeles

Abstract. The ability of using aggregates in recursive Datalog queries is making possible a simple declarative expression for important algorithms and it is conducive to their parallel implementations with scalability and performance that often surpass those of their formulations in GraphX and Scala. These recent advances were made possible by the notion of Pre-Mappability ($\mathscr{P}reM$) that, along with a highly optimized seminaive-based operational semantics, guarantees their formal non-monotonic semantics for the programs expressing these declarative algorithms. However, proving that these programs have the $\mathscr{P}reM$ property can be too difficult for everyday programmers. Therefore in this paper, we introduce basic templates that facilitate and automate this formal task. These templates are based on simple extensions of Functional and Multivalued Dependencies (FDs and MVDs) whereby properties such as the mixed transitivity of MVDs and FDs are used to prove the validity of these powerful declarative algorithms.

1 Introduction

The surge of interest in BigData has brought a renaissance of interest in Datalog and its ability to specify, declaratively, advanced data-intensive applications that execute efficiently over different systems and parallel architectures [1, 6, 7, 9, 10]. While efficient and scalable support for non-recursive SQL queries was realized as far back as the 1980s [8], the much more powerful queries that use negation or aggregates in recursion proved much more difficult, as the challenge of providing formal non-monotonic semantics and efficient implementations remained unanswered for many years. Recently however, significant progress on both the implementation and semantic fronts has been achieved by the UCLA BigDatalog project that, using the notion of Pre-Mappability ($\mathscr{P}reM$), can support a wide spectrum of declarative algorithms, with scalability and performance levels that often surpass those of other Datalog implementations, and those of Apache Spark application packages such as GraphX [7, 10]. The notion of $\mathscr{P}reM$ was introduced in [11], and in [12] it is shown that $\mathscr{P}reM$ enables the formulation of a wide range of applications under stable model semantics. In this paper, we summarize those findings and delve deeper on how to verify $\mathscr{P}reM$ by using simple templates based on extensions of functional and multivalued dependencies.

2 Pre-Mappable Constraints

This section contains a brief summary of results from [11] and [12]. In 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*).*

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

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 to D as the *cost* argument (cost argument consists of a single variable). This goal states the *constraint* that for a pair (Y,D) no other pair exists having the same Y-value and a smaller D-value. Thus the formal meaning of our constraint is defined by replacing $\mathtt{r}_3$ with $\mathtt{r}_4$:

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

where the goal is_min((Y),D) has been replaced by the negated goal ¬smlr_pth(Y,D), where smlr_pth(Y,D) is defined as:

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

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 stratum and qpth occupying the higher stratum. Thus, the program has a perfect model semantics [5]. However, the iterated fixpoint procedure proposed in [5] is transfinite and very inefficient and even unsafe in practice—for the example at hand it does not terminate when the graph has cycles. To solve these problems, [11] introduces the $\mathscr{P}$*reM* condition, under which 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).

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

Thus we have seen two formulations for this algorithm: the program in Example 2, with min in recursion will be called the *endo-min version*, and the original program of Example 1 that 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 where the iterated fixpoint of the exo-min version would be very inefficient or even non-terminating, as it is in fact the case when the graph defined by arc contains directed cycles. $\mathscr{P}$*reM* is indeed a very powerful condition that allows us to express declaratively a large number of basic algorithms, while assuring that (i) they have a rigorous non-monotonic semantics [11, 12], and (ii) they are amenable to very efficient implementations of superior scalability [7, 10]. In the rest of this section, we define $\mathscr{P}reM$ and its formal semantics, following [11, 12]. Then in the rest of the paper we focus on defining simple templates that greatly simplify the task of proving that the Datalog program at hand has the $\mathscr{P}reM$ property—as needed to allow everyday programmers to take full advantage of this powerful new formal tool.

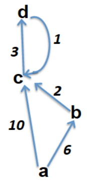
Fixpoint and $\mathscr{P}reM$ Constraints. We will consider stratified programs, such as those of Example 1, having a perfect model semantics computed by strata. At the lower stratum we find the minimal model of the rules defined by (i) interpreted predicates, such as comparison and arithmetic predicates, and (ii) positive rules such as $\mathtt{r}_1$ and $\mathtt{r}_2$ defining the `pth` predicate. If T is the Immediate Consequence Operator (ICO) for the program defined by these positive rules, then its unique minimal model is defined by the least-fixpoint of T, which can be computed as $T^{\uparrow\omega}(\emptyset)$ (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 $\gamma = \mathtt{is\_min((Y),D)}$ 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 $\mathscr{P}reM$ [11] 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 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 $T_\gamma(I)$. In many programs of practical interest, the transfer of constraints under $\mathscr{P}reM$ produces optimized programs for the naive fixpoint computation that are safe and terminating even when the original programs were not. Thus we focus on programs where, for some integer n, $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$, i.e., the fixpoint iteration converges after a finite number of steps n. As proven in [11], the fixpoint $T_\gamma^{\uparrow n}(\emptyset)$ so obtained is in fact a minimal fixpoint for T_γ, where γ denotes a min or max constraint:

Theorem 1. *If γ is $\mathscr{P}reM$ to a positive program P with ICO 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. We refer to these as *cost-atoms*. In the example below, we have aligned in columns the cost atoms sharing the same group-by value, and used a bar to separate cost atoms from their successors that, because of their lower costs, replaced them at the next step.

Example 4 (Computing $T_\gamma^{\uparrow n}(\emptyset)$ for Example 1 on facts in Example 3).

Step 1: $\texttt{pth(b,6)}$, $\underline{\texttt{pth(c,10)}}$.
Step 2: $\texttt{pth(b,6)}$, $\texttt{pth(c, 8)}$, $\underline{\texttt{pth(d,13)}}$.
Step 3: $\texttt{pth(b,6)}$, $\texttt{pth(c, 8)}$, $\texttt{pth(d,11)}$.
Step 4: $\texttt{pth(b,6)}$, $\texttt{pth(c, 8)}$, $\texttt{pth(d,11)}$.

Thus, we have derived the `pth` atoms `pth(b,6)`, `pth(c,8)`, `pth(d,11)` which constitute the extreme subset of the cost atoms in $T^{\uparrow\omega}(\emptyset)$. Since the $\mathscr{P}reM$ property holds for the recursive rules in Example 1, these extreme atoms, renamed `qpth`, along with the atoms in $T^{\uparrow\omega}(\emptyset)$ constitute the perfect model for the original program. Moreover, in [12] we have shown that the program of Example 2, and most of the endo-min or endo-max programs that satisfy $\mathscr{P}reM$ have a stable model semantics [2].[1] Besides its obvious theoretical interest, the existence of a stable model dovetails with our experience that programmers rather than writing exo-min and exo-max versions often write the endo-min or endo-max versions of their algorithms directly, inasmuch as these preserve the spirit and intuition of the procedural formulations of the same algorithms.

The obvious theoretical interest of $\mathscr{P}reM$ notwithstanding, the concept would remain of limited practical interest, until we can provide programmers with simple tools that they can use to verify that their declarative algorithms satisfy $\mathscr{P}reM$. The rest of the paper focuses on providing formal tools that answer this very practical requirement.

3 Declarative Algorithms and the $\mathscr{P}reM$ Property

A wide spectrum of declarative algorithms of practical interest can be expressed by programs that satisfy $\mathscr{P}reM$. For some programs (e.g., those disussed in Section 3.4), the proof that $\mathscr{P}reM$ holds is relatively simple, but for many others, including our

[1] In general it can be shown that endo-min or endo-max programs have a total stable model whenever there is no cyclic derivation in their cost atoms. This is in fact the case for Example 2, when the graph defined by `arc` has no directed cycle, or its arcs have positive length [12].

running example, i.e., Example 2, a direct proof can be quite challenging. To handle these programs we propose simple tools that can be used by the programmer, or the compiler, to prove $\mathscr{P}reM$ by drawing from the classical theory of Functional and Multivalued Dependencies (FDs and MVDs) used in relational DB schema design.

A first observation to be made is that $\mathscr{P}reM$ 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 validity of $\mathscr{P}reM$ can be illustrated by adding an additional goal to the rule to express the pre-application of the extrema constraint γ onto $T_\gamma(I)$, whereby γ is first applied to I. For instance, the recursive rule of our Example 2 should be re-written with the insertion into the rule of the additional goal$^{\backslash\texttt{is\_min((X),Dx)}/}$, which we will call the *drop-in* goal, producing:

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

Observe that adding the drop-in goal corresponds to pre-applying the γ constraint to the argument I in $(T_\gamma(I))$. For $\mathscr{P}reM$ to hold, we must have $T_\gamma(\gamma(I)) = T_\gamma(I)$ which states that the drop-in goal has not changed the ICO mapping specified by the original recursive endo-min rule. For Example 2, consider the γ constraint applied to the pair (Y,D) that produces the head of the rule, pth(Y,D): this constraint is is_min((Y),D), which specifies that the second argument of pth is minimized for each value of the first argument of pth. Therefore, the drop-in goal inserted after the goal is pth(X,Dx) is $^{\backslash\texttt{is\_min((X),Dx)}/}$. Then $\mathscr{P}reM$ is proven once we prove that the addition of this goal does not change the ICO mapping defined by the rule. Providing such a proof can be difficult in many cases, including the one of our running example. Therefore, we will next identify common patterns that guarantee the validity of $\mathscr{P}reM$ for such complex examples.

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

In many cases $\mathscr{P}reM$ can be inferred from the properties of extrema goals and the multivalued dependencies that hold in the equivalent relational views of the (function-free) predicates in the rule body. Let I be an interpretation of our program P. Then, $Rq = \{(x_1,\ldots,x_n) \mid \texttt{q}(\texttt{x}_1,\ldots,\texttt{x}_n) \in I\}$ will be said to be the *relational view* of the predicate q for the given I.

Definition 2 (*Functional Dependencies on tuples). Let $R(\Omega)$ be a relation, and $X \subset \Omega$ and $A \in \Omega - X$. Then, we say that a given tuple $t \in R$ satisfies the FD $X \rightarrow A$ if R does not contain any tuple having the same X-value but a different A-value.*

Now if the domain of A is totally ordered, $X \rightarrow A$ holds for a tuple t if R contains no tuple having the same X-value and an A-value that is either larger or smaller than the A-value of t. We next define the concept of min-constraint and max-constraint that only excludes the presence of smaller tuples and larger tuples, respectively.

Definition 3. *We will say that a tuple $t \in R$ satisfies the min-constraint* is_min((X),A) *and write $X \xrightarrow{min} A$ when R contains no tuple having the same X-value and a smaller A-value. Symmetrically, we say that the tuple $t \in R$ satisfies the max-constraint* is_max((X),A) *and write $X \xrightarrow{max} A$ when* R *contains no tuple with the same X-value and a larger A-value.*

Informally $X \xrightharpoonup{min} A$ and $X \xrightharpoondown{max} A$ can be viewed as "half-FDs", since both must hold before we can conclude that $X \rightarrow A$. Moreover, while min-constraints and max-constraints on single tuples are much weaker than regular FDs, they preserve some of their important formal properties including the ones discussed next that also involve MVDs on single tuples, which we define next.

Definition 4. *Let $t \in R(\Omega)$, and X, Y, and Z be subsets of Ω where $Z = \Omega - (X \cup Y)$. Then a tuple $t \in R$ satisfies the MVD* $\texttt{X} \twoheadrightarrow \texttt{Y}$ *when the following property holds: if t' is a tuple in R that is equal to t in its X-values, then R also contains (i) some tuple that is identical to t in the $X \cup Y$ values and identical to t' in the $X \cup Z$ values, and (ii) some tuple that is identical to t' in the $X \cup Y$ values and identical to t in the $X \cup Z$ values.*

Observe that these definitions of tuple-based FDs and MVDs are consistent with the standard ones used for whole relations since a relation satisfies a given set of MVDs and FDs (with one attribute on their right side) iff these MVDs and FDs are satisfied by each of its tuples. Therefore, the following properties hold for tuple constraints (i.e., for min-constraints, max-constraints, and tuple MVDs) and also illustrate the appeal of the arrow-based notation:

Min/Max Augmentation: If $X \xrightharpoonup{min} A$ and $Z \subseteq \Omega$, then $X \cup Z \xrightharpoonup{min} A$.
If $X \xrightharpoondown{max} A$ and $Z \subseteq \Omega$, then $X \cup Z \xrightharpoondown{max} A$.
MVD Augmentation: If $X \twoheadrightarrow Y$, $Z \subseteq \Omega$ and $Z \subseteq W$, then $X \cup W \twoheadrightarrow Y \cup Z$.
Mixed Transitivity: If $Y \twoheadrightarrow Z$ and $Z \xrightharpoonup{min} A$, with $A \notin Z$, then $Y \xrightharpoonup{min} A$.
If $Y \twoheadrightarrow Z$ and $Z \xrightharpoondown{max} A$, with $A \notin Z$, then $Y \xrightharpoondown{max} A$.

The augmentation property for min and max constraints, follows directly from the definition, while the mixed-transitivity property is proven in [13].

3.2 Declarative Algorithms

We will now show how advanced declarative algorithms can be expressed using recursive programs with aggregates via the $\mathscr{P}$*reM* condition that can be easily proven using the properties of min/max constraints displayed above.

Bill of Materials. A classical recursive application for traditional databases is Bill of Materials (BOM), where we have a Directed Acyclic Graph (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, since 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.

```
deliv(Part,Days) ← basic(Part,Days), is_max((Part),Days).
deliv(Part,Days) ← deliv(Sub,Days), assbl(Part,Sub), is_max((Part),Days).
```

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

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

Thus we want to prove that the drop-in goal does not change the mapping defined by this rule. In our proof we will refer to $\texttt{is\_max((Sub),Days)}$ as the *drop-in* constraint and to $\texttt{is\_max((Part),Days)}$ as the *original* constraint. Let $R(Sub, Days, Part)$ be the natural join of the relational views of $\texttt{deliv(Sub,Days)}$ and $\texttt{assbl(Part,Sub)}$. Then the following MVDs hold for all tuples in R: $Sub \twoheadrightarrow Days$ and $Sub \twoheadrightarrow Part$. Now for any tuple in R that satisfies the constraint $Part \xrightarrow{max} Days$, it also satisfies $Sub \xrightarrow{max} Days$ by Mixed Transitivity. Thus if R' is the set of tuples in R that have the property $Part \xrightarrow{max} Days$, R' also satisfies $Sub \xrightarrow{max} Days$. Thus the drop-in constraint does not change R' or the mapping defined by it, since applying a constraint to a relation that already satisfied it does not change the relation. □

Connected Components in a Graph. In this application we have an undirected graph, where an edge connecting, say, `a` and `b` is represented by the pairs `edge(a,b)` and `edge(b,a)`. Then, if we represent the nodes by integers, we can select the node with the lowest integer to serve as the representative for its clique.

Example 5 (Connected Components in an undirected graph.).

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

Here we can observe that in $R(\texttt{X,Y,Z})$ obtained as the natural join of $\texttt{cc(X,Y)}$ and $\texttt{edge(Z,Y)}$ the following MVD holds: $\texttt{Y} \twoheadrightarrow \texttt{Z}$. Thus for tuples that satisfy the constraint $\texttt{Z} \xrightarrow{\texttt{min}} \texttt{X}$, $\texttt{Y} \xrightarrow{\texttt{min}} \texttt{X}$ also holds. This is the drop-in constraint, which therefore does not change the mapping defined by our rule.

Minimal Distances in Directed Graph. Let us now return to our Example 2, and let us re-write its recursive rule with drop-in goal into the following equivalent one:

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

Given a relation $R(X, Y_1, \ldots Y_n, Z)$, the fact that $Y_1, \ldots, Y_n \twoheadrightarrow X$ and $Y_1, \ldots, Y_n \twoheadrightarrow Z$ hold guarantees that the tuples that satisfy the condition $\texttt{is\_min}((\texttt{Y}_1 \ldots, \texttt{Y}_n), \texttt{X} + \texttt{Z})$ are exactly those that satisfy both conditions $Y_1 \ldots, Y_n \xrightarrow{min} X$ and $Y_1 \ldots, Y_n \xrightarrow{min} Z$. Therefore, for the example at hand, let R be the natural join of the relational views of $\texttt{path(X,Dx)}$ and $\texttt{arc(X,Y,Dxy)}$. Then we can reason as follows:

Step 1 [Find MVDs in R]: $\texttt{X} \twoheadrightarrow \texttt{Dx}$ and $\texttt{X} \twoheadrightarrow \texttt{Y,Dxy}$.
Step 2 [Augment MVDs as needed to distribute distribute min]
$\texttt{X,Y} \twoheadrightarrow \texttt{Dx}$ and $\texttt{X,Y} \twoheadrightarrow \texttt{Y,Dxy}$ hold in R. Then tuples that satisfy $\texttt{X,Y} \xrightarrow{\texttt{min}} \texttt{Dx+Dxy}$ also satisfy $\texttt{X,Y} \xrightarrow{\texttt{min}} \texttt{Dx}$ and $\texttt{X,Y} \xrightarrow{\texttt{min}} \texttt{Dxy}$.
Step 3 [Augment left sides for mixed transitivity and apply it]: From $\texttt{X} \twoheadrightarrow \texttt{X,Y,Dxy}$ and $\texttt{X,Y} \xrightarrow{\texttt{min}} \texttt{Dx}$ augmented into $\texttt{X,Y,Dxy} \xrightarrow{\texttt{min}} \texttt{Dx}$, we infer our drop-in constraint: $\texttt{X} \xrightarrow{\texttt{min}} \texttt{Dx}$. □

As illustrated by the next example, these patterns can be applied mechanically, whereby $\mathscr{P}$*reM* can be verified by compilers and users who do not know DB theory.

Non-linear Shortest paths. Shortest paths can also be computed with non-linear rules:

Example 6 (Shortest paths à la Floyd).

$$\begin{array}{ll} \mathtt{r_0}\text{: } \mathtt{qsp(X,Y,Vxy)} \leftarrow & \mathtt{arc(X,Y,Dxy), is\_min((X,Y),Vxy).} \\ \mathtt{r_1}\text{: } \mathtt{qsp(X,Z,V)} \leftarrow & \mathtt{qsp(X,Y,Vxy), qsp(Y,Z,Vyz),} \\ & \mathtt{V = Vxy{+}Vyz, is\_min((X,Z),V).} \end{array}$$

Here we must check $\mathscr{P}$*reM* for two drop-in constraints, as follows:

$$\begin{array}{l} \mathtt{qsp(X,Z,V)} \leftarrow \mathtt{qsp(X,Y,Vxy)}^{\backslash \mathtt{is\_min((X,Y),Vxy)}/}, \mathtt{qsp(Y,Z,Vyz)}^{\backslash \mathtt{is\_min((Y,Z),Vyz)}/}, \\ \qquad\qquad \mathtt{V = Vxy + Vyz, is\_min((X,Z),V).} \end{array}$$

So let $R(\mathtt{X,Y,Vxy,Z,Vyz})$ be the natural join on the column $\mathtt{Y}$ of $\mathtt{qsp(X,Y,Vxy)}$ and $\mathtt{qsp(Y,Z,Vyz)}$, we have:

Step 1 [Find MVDs in R] $\mathtt{Y} \twoheadrightarrow \mathtt{X,Vxy}$ and $\mathtt{Y} \twoheadrightarrow \mathtt{Z,Vyz}$.

Step 2 [Augment MVDs and min-constraints to match and distribute] $\mathtt{X,Y,Z} \twoheadrightarrow \mathtt{Vxy}$ and $\mathtt{X,Y,Z} \twoheadrightarrow \mathtt{Vyz}$ hold in R. Also augment $\mathtt{X,Z} \xrightarrow{\mathtt{min}} \mathtt{Vxy + Vyz}$ to $\mathtt{X,Y,Z} \xrightarrow{\mathtt{min}} \mathtt{Vxy + Vyz}$, from which we derive $\mathtt{X,Y,Z} \xrightarrow{\mathtt{min}} \mathtt{Vxy}$ and $\mathtt{X,Y,Z} \xrightarrow{\mathtt{min}} \mathtt{Vyz}$.

Step 3 [Augment min constraint and apply mixed transitivity] From the second MVD in Step 1 and the first min-constraint in Step 2, we infer: $\mathtt{X,Y} \twoheadrightarrow \mathtt{X,Y,Z,Vyz}$ and $\mathtt{X,Y,Z,Vyz} \xrightarrow{\mathtt{min}} \mathtt{Vxy}$. From these two we infer: $\mathtt{X,Y} \xrightarrow{\mathtt{min}} \mathtt{Vxy}$. Symmetrically, from $\mathtt{Y,Z} \twoheadrightarrow \mathtt{X,Y,Z,Vxy}$ and $\mathtt{X,Y,Z,Vxy} \xrightarrow{\mathtt{min}} \mathtt{Vyz}$, we infer $\mathtt{Y,Z} \xrightarrow{\mathtt{min}} \mathtt{Vyz}$. □

3.3 The Aggregates Count and Sum in Recursive Rules

The traditional `count` and `sum` aggregates can be viewed as the maximized versions of their cumulative (a.k.a. progressive) versions, which we will, respectively, denote by `mcount` and `msum` [4]. For instance, the following example shows how the progressive count of items can be defined using Horn clauses.

Example 7 (Progressive, monotonic count of elements `item`*).*

$$\begin{array}{ll} \mathtt{mcnt(1,[IT])} \leftarrow & \mathtt{item(IT).} \\ \mathtt{mcnt(J1,[IT|Allpr])} \leftarrow & \mathtt{item(IT), mcnt(J,Allpr),} \\ & \mathtt{notin(IT,Allpr), J1 = J{+}1.} \end{array}$$

$$\begin{array}{l} \mathtt{notin(IT1,[IT2\mid Rest])} \leftarrow \mathtt{IT1} \neq \mathtt{IT2,\ notin(IT2,\ Rest).} \\ \mathtt{notin(IT,\ [\,]).} \end{array}$$

The above program progressively enumerates and counts all the items in every possible order; the program is clearly monotonic in the lattice of set-containment, whereby the standard least-fixpoint semantics applies. The `mcount` aggregate is supported in [7] as an efficient built-in aggregate that only visits and progressively counts the items in the order in which they are stored. Moreover, the standard `count` aggregate is the maximum value returned by the progressive count, whereby `count` can be used instead `mcount` in programs where $\mathscr{P}$*reM* is satisfied for max.

For instance, in the following example, in addition to the organizers, we include people who see that three or more of their friends have joined the event, using the `mcount` built-in, which takes three arguments: the first is the group-by argument, the second is the item being counted, and the third is the result, i.e., the progressive count being returned.

Example 8 (Joining the event).

$$\begin{array}{ll} \texttt{jnd(X,0)} \leftarrow & \texttt{organizer(X)}. \\ \texttt{jnd(Y,Ycnt)} \leftarrow & \texttt{jnd(X,Cx)}, \texttt{Cx} \geq 3, \texttt{friend(Y,X)}, \texttt{mcount((Y),X,Ycnt)}. \\ \texttt{result(Y,Ycnt)} \leftarrow & \texttt{jnd(Y,Ycnt)}. \end{array}$$

Thus, if `tom` has joined the event along with five of his/her friends, the use of `mcount` produces the following progressive result: `result(tom,1)`, `result(tom,2)`, `result(tom,3)`, `result(tom,4)`, `result(tom,5)`. If we are only interested in the actual count, i.e., to get back `result(tom,5)`, then we can add the additional goal `is_max((Y),Ycnt)` to the final rule. Then a natural question to arise is whether `is_max((Y),Ycnt)` can actually be pre-mapped into the recursive rule, which would become the one below:

$$\begin{array}{ll} \texttt{jnd(Y,Ycnt)} \leftarrow & \texttt{jnd(X,Cx)}, \texttt{Cx} \geq 3, \texttt{friend(Y,X)}, \\ & \texttt{mcount((Y),X,Ycnt)}, \texttt{is\_max((Y),Ycnt)}. \end{array}$$

where `mcount((Y),X,Ycnt),is_max((Y),Ycnt)` can then be evaluated by the regular count `count((Y),X,Ycnt)`.

For that, we will have to show that a drop-in goal `is_max((X),Cx)` does not change the mapping defined by this rule. Indeed, for a given `X`-value, there exists some `Cx` value such that $\texttt{Cx} \geq 3$ if this inequality is satisfied by the largest of those `Cx`-values. Thus, $\mathscr{P}$*reM* holds and `mcount` can be optimized into the regular `count`. Observe that unlike the examples in Section 3.2, $\mathscr{P}$*reM* is established without exploiting the relationships between the cost arguments of the original `is_max` goal and the drop-in goal. Also observe that $\mathscr{P}$*reM* no longer holds if the condition $\texttt{Cx} \geq 3$ is replaced by $\texttt{Cx} = 3$ or $\texttt{Cx} < 3$.

From Monotonic Sum to Regular Sum. The notion of monotonic sum (`msum`) for positive numbers was introduced in [4] using the fact that the semantics of `msum` and `sum` for positive numbers can be easily reduced to that of `mcount` and `count`, respectively by the fact that, given a set of integers S, their sum is equal to the count of the pairs $\{(N,j) | N \in S \text{ and } 1 \leq j \leq N\}$. Thus the rule patterns for which $\mathscr{P}$*reM* holds and thus the maximized `msum` can be evaluated as a regular sum are basically those where `mcount` can be evaluated as a regular `count`. These patterns actually apply to a large number of examples discussed in the literature, including, e.g., counting the paths in a directed graph, Viterbi algorithm, Company Control, and other algorithms that were covered in [3,4,7]. Furthermore, as discussed in [4], besides positive integers, arbitrary floating-point numbers can also be used in the sum.

4 Conclusion

Recent work on Datalog programs using aggregates in recursion has delivered formal semantics [11] and efficient implementations [7,10] for programs that satisfy the $\mathscr{P}reM$ condition, which can be used to specify concisely and declaratively a wide spectrum of declarative algorithms. In this paper, we have addressed the problem of making $\mathscr{P}reM$ easier to use and thereby attractive to programmers. For that, we have proposed simple arrow-based patterns that will allow the user, or the compiler, to infer that $\mathscr{P}reM$ holds for their program. Simple extensions of the dependency theory of relational DBs have been used to prove the correctness of these patterns.

Besides exploring how to apply these advances to other query languages, including SQL, our current research seeks to understand and characterize the power and limitations of this approach in supporting declarative algorithms. In particular we would like to (i) characterize which algorithms can be expressed by $\mathscr{P}reM$ programs using aggregates in recursion (including the four discussed here and other aggregates as well), and (ii) how the pattern-based tools we have presented here can be generalized to simplify the life of programmers and the task of compilers in, respectively, specifying and supporting powerful declarative algorithms.

References

1. M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, et al. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382. ACM, 2015.
2. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *ICLP*, pages 1070–1080, 1988.
3. M. Mazuran, E. Serra, and C. Zaniolo. A declarative extension of Horn clauses, and its significance for Datalog and its applications. *TPLP*, 13(4-5):609–623, 2013.
4. M. Mazuran, E. Serra, and C. Zaniolo. Extending the power of Datalog recursion. *The VLDB Journal*, 22(4):471–493, 2013.
5. T. C. Przymusinski. Perfect model semantics. In *ICLP/SLP*, pages 1081–1096, 1988.
6. J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed SociaLite: a Datalog-based language for large-scale graph analysis. *PVLDB*, 6(14):1906–1917, 2013.
7. A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with Datalog queries on Spark. In *SIGMOD*, pages 1135–1149. ACM, 2016.
8. Teradata. *Data Bases Computer Concepts and Facilities*. Teradata: Document Number CO2-0001-00, 1983.
9. J. Wang, M. Balazinska, and D. Halperin. Asynchronous and fault-tolerant recursive Datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.
10. M. Yang, A. Shkapsky, and C. Zaniolo. Scaling up the performance of more powerful Datalog systems on multicore machines. *The VLDB Journal*, 26(2):229–248, 2017.
11. C. Zaniolo, M. Yang, A. Das, A. Shkapsky, T. Condie, and M. Interlandi. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *TPLP*, 17(5-6):1048–1065, 2017.
12. C. Zaniolo, M. Yang, A. Das, A. Shkapsky, T. Condie, and M. Interlandi. Declarative algorithms in Datalog with aggregates: their formal semantics simplified. *submitted for publication*, pages 1–19, 2018.
13. C. Zaniolo, M. Yang, M. Interlandi, A. Das, A. Shkapsky, and T. Condie. Declarative algorithms by aggregates in recursive queries: their formal semantics simplified. Report no. 180001, Computer Science Department, UCLA, April, 2018.