Developing Big-Data Application as Queries: an Aggregate-Based approach

Carlo Zaniolo, Ariyam Das, Jiaqi Gu, Youfu Li, Mingda Li, Jin Wang
University of California at Los Angeles

Abstract

Recent advances on query languages (QLs) and DBMS suggest that their traditional role in application development can and should be extended dramatically in many big-data application areas, including graph, machine learning and data mining applications. This is made possible by the superior expressive power that database aggregates bring to recursive queries and the realization of their powerful non-monotonic semantics via efficient and scalable fixpoint-base operational semantics. Thus, in this paper, we discuss how classical algorithms can be expressed concisely using queries with aggregates in recursion that have a rigorous declarative semantics. Then we discuss what modifications, if any, are needed on such programs to have an efficient and scalable fixpoint-based operational semantics, whereby we can also identify queries that are conducive to bulk-synchronous and stale-synchronous parallelism.

1 Introduction

Relational DBMS and their logic-based QLs made possible for programmers to develop applications without having to navigate database storage structures via statements written in a procedural language. Many initial skeptics notwithstanding, relational DBMS proved quite effective in terms of usability, performance and scalability. In fact their success led to and was reinforced by significant extensions, including the introduction of very powerful aggregate functions, such as OLAP functions that enable direct support for descriptive analytics by SQL queries. Another important extension was the SQL support for recursive queries which allows simple algorithms, such as transitive closure, to be expressed directly as queries. However, the quantum leap in expressive power achievable by combining recursive queries with aggregates was never realized because of SQL stratification requirement, which specifies that non-monotonic constructs can be applied to the results of recursive definitions but cannot be used in the recursive definitions. This requirement was then enforced to avoid the major semantic problems faced by recursive reasoning via non-monotonic constructs. However, significant progress was made since then by researchers focusing on the use of aggregates in AI, logic programming and Datalog: for instance, the concept of Stable Models has gained wide acceptance as the formal basis for declarative semantics in the logic programming arena [5] [6]. So far, however, these advances did not have much impact upon the database field because of two main issues. The first issue is that the non-constructive definition of Stable Model Semantics (SMS) for programs with negation is making difficult for programmers to show that their queries with aggregates satisfy SMS, and the second issue is that establishing the SMS for a program does not guarantee its efficient constructive realization, and significant re-writing of the original program is often needed to implement it via fixpoint computations and the recursive query implementation techniques of SQL DBMS, as well as Datalog systems.

In this paper, we describe an approach that addresses these two issues and proved successful in a number of advanced applications [1, 8, 9, 11–13, 15]. We will start with an intuitive treatment of the declarative SMS of recursive queries with extrema, and show that queries with count, sum and average can be reduced to queries with max. Then, we provide simple criteria to detect when the SMS of such queries can be turned directly into an efficient and scalable fixpoint computation and when these instead require significant rewriting by the techniques described in the paper. While in our examples we use Datalog programs, we will show how these can be expressed using SQL queries for which the same conclusions apply. Throughout the paper, we will refer to queries and programs as synonyms.

2 Stable Model Semantics and Fixpoint Computation for Programs with Extrema

A simple application of extrema in recursive queries consists in finding the min or max distance from a given initial node a of all nodes in the graph where the edges have positive length. The following program, computing max distances, exemplifies key semantic issues.

Example 1 (*A stratified program to compute the max distance from* a **):**

$$\begin{array}{ll} r_1 : \texttt{dist}(\texttt{a}, 0). & \\ r_2 : \texttt{dist}(\texttt{Y}, \texttt{Dy}) \leftarrow & \texttt{dist}(\texttt{X}, \texttt{Dx}), \texttt{arc}(\texttt{X}, \texttt{Y}, \texttt{Dxy}), \texttt{Dy}{=}\texttt{Dx}{+}\texttt{Dxy}. \\ r_3 : \texttt{mxdist}(\texttt{Y}, \texttt{max}\langle\texttt{Dy}\rangle) \leftarrow & \texttt{dist}(\texttt{Y}, \texttt{Dy}). \end{array}$$

Assume for instance that we have the following fact base that describes an acyclic directed graph:

$$\texttt{arc}(\texttt{a}, \texttt{b}, 10) \quad \texttt{arc}(\texttt{a}, \texttt{c}, 20) \quad \texttt{arc}(\texttt{b}, \texttt{c}, 18) \quad \texttt{arc}(\texttt{c}, \texttt{d}, 12)$$

Then, the *semi-naive fixpoint* computation on the first two rules derives the following new atoms at each step (whereas the *naive fixpoint* includes the atoms produced at previous steps along with those produced at this step):

Step 1	Step 2	Step3	Step 4
$\texttt{dist}(\texttt{a}, 0)$	$\texttt{dist}(\texttt{b}, 10),\ \texttt{dist}(\texttt{c}, 20)$	$\texttt{dist}(\texttt{c}, 28),\ \texttt{dist}(\texttt{d}, 32)$	$\texttt{dist}(\texttt{d}, 40)$

With the computation on the first two rules having reached fixpoint, rule r_3 is applied next, whereby Step 5 produces $\texttt{mxdist}(\texttt{a}, 0)$, $\texttt{mxdist}(\texttt{b}, 10)$, $\texttt{mxdist}(\texttt{c}, 28)$, and $\texttt{mxdist}(\texttt{d}, 40)$, while $\texttt{mxdist}(\texttt{c}, 20)$ and $\texttt{mdxist}(\texttt{d}, 32)$ are not derived since they are dominated by the previous atoms[1] and thus they are not maximal.

Our stratified derivation can be optimized by pushing the max constraint into recursion, while keeping the original stratification whereby the fixpoint computation of dist by rules r_1 and r_2 must be completed before r_3 can be used to derive mxdist.

Example 2 (*Using max in recursion to compute the max distance from* a *):*

$$\begin{array}{ll} r_1 : \texttt{dist}(\texttt{a}, 0). & \\ r_2 : \texttt{dist}(\texttt{Y}, \texttt{max}\langle\texttt{Dy}\rangle) \leftarrow & \texttt{dist}(\texttt{X}, \texttt{Dx}), \texttt{arc}(\texttt{X}, \texttt{Y}, \texttt{Dxy}), \texttt{Dy}{=}\texttt{Dx}{+}\texttt{Dxy}. \\ r_3 : \texttt{mxdist}(\texttt{Y}, \texttt{Dy}) \leftarrow & \texttt{dist}(\texttt{Y}, \texttt{Dy}). \end{array}$$

The semi-naive fixpoint computation of our rules r_1 and r_2 so revised produces atoms that are dominated by other atoms produced at later steps. These non-maximal atoms are called *provisional max* atoms, and they are depicted as cancelled in the picture below, since they will be lost at later steps.

Step 1	Step 2	Step3	Step 4
$\texttt{dist}(\texttt{a}, 0)$	$\texttt{dist}(\texttt{b}, 10),\ \cancel{\textit{dist}(\texttt{c}, 20)}$	$\texttt{dist}(\texttt{c}, 28),\ \cancel{\textit{dist}(\texttt{d}, 32)}$	$\texttt{dist}(\texttt{d}, 40)$

Max (min) atoms that are not provisional are called *final* max (min) atoms. Then, with the fixpoint computation for dist completed Step 4, r_3 at Step 5 simply renames the final max atoms so obtained producing $\texttt{mxdist}(\texttt{a}, 0)$, $\texttt{mxdist}(\texttt{b}, 10)$, $\texttt{mxdist}(\texttt{c}, 28)$ and $\texttt{mxdist}(\texttt{d}, 40)$.

The minimal model of a monotonic program can be derived by an *eager* computation that, at each step, derives all the possible consequences of the current interpretation. A derivation that at each step only derives a non-empty subset of those consequences will be called *judicious*. Then, we have that *a program with extrema has M as its stable model iff M can be derived via a judicious derivation that contains no provisional atom* [16].

For instance, the stratified derivation shown for Example 1 is a judicious one since the fixpoint computation of dist by rules r_1 and r_2 produces no provisional atom, and r_3 then selects from the atoms so generated the

[1] We say that $\texttt{atom}(\texttt{X1}, \texttt{Y1})$ dominates $\texttt{atom}(\texttt{X2}, \texttt{Y2})$ when $\texttt{X1} = \texttt{X2}$ and $\texttt{Y1} > \texttt{Y2}$.

final max mxdist atoms. Indeed, the iterated fixpoint computation of every max-stratified program defines a judicious computation that produces its stable model.

On the other hand, although the eager computation in Example 2 contains provisional atoms, it still computes the stable model of our program. Indeed, after all those provisional atoms are cancelled during the fixpoint computation, we have a valid judicious derivation for the stable model of our program. Programs that have this property will be said to be *resilient*. The program in Example 2 is resilient.

The requirement that programs must have SMS is now widely accepted in the field because otherwise programs might not have sound logic-based semantics, as illustrated by the following example. Say, for instance, that users are only interested in nodes whose distance from a along the longest path is < 40, and they add the condition Dy$<$40 to r_3 in Example 2. The modified program is still stratified and returns dist(a, 0), dist(b, 10), dist(c, 28) dist(d, 40) and mxdist(a, 0), mxdist(b, 10), mxdist(c, 28), which defines its stable model.

However, if Dy$<$40 is added to r_2, instead of r_3, then the eager computation generates the following derivation:

Example 3 (*Eager computation for Ecample 2 after* dist(c, 20) *is added to* r_2*):*

Step 1	Step 2	Step 3
dist(a, 0)	dist(b, 10), ~~*dist(c, 20)*~~	dist(c, 28), dist(d, 32)

Then in Step 4, r_3 derives from these: mxdist(a, 0), mxdist(b, 10), mxdist(c, 28), mxdist(d, 32).
The logical problem with this outcome is that the atom dist(c, 20) is no longer in the result; thus, there is no justification for dist(d, 32) and mxdist(d, 32) to be in the answer.

Therefore, for Datalog programs with aggregates to produce logically sound results, the programs must have SMS. We will first discuss how to achieve this objective for programs with max and min, and then for programs with count, sum and average whose formal semantics is actually defined using max.

Pre-Mappable Extrema and Stable Model Semantics: In our previous examples, we have seen that an eager fixpoint computation produces a stable model for some queries but not for others. Thus programmers need simple criteria to guarantee that their programs belong to the first group (i.e., they are resilient) and the notion of pre-mappable ($\mathcal{P}reM$) constraints [17] is introduced next to satisfy this requirement.

The mapping defined by a set of rules in a program is called their Immediate Consequence Operator (ICO) and it is denoted by T. Then, the constraint γ is said to be pre-mappable ($\mathcal{P}reM$) to T when, for every interpretation I of P, we have that: $\gamma(T(I)) = \gamma(T(\gamma(I)))$.

For instance, in rule r_2 of Example 1, to find the maximal Dy for a given Y, we only need to consider the max Dx value for the X value that, via arc(X, Y, Dxy), produces that Y. Thus, in Example 2, the constraint is_max(Y, Dy) applied to the head of r_2 can pre-applied to dist(X, Dx) in the body of the rule, and from there to the rules generating dist(X, Dx) in the previous step of the fixpoint computation without changing the results of the fixpoint computation. This $\mathcal{P}reM$ property allows us to transform Example 1 into Example 2 with assurance that the same mxdist results will be obtained. More recently, it was proved [16] that $\mathcal{P}reM$ also guarantees that recursive programs with extrema satisfying $\mathcal{P}reM$ have a SMS that can be computed using the fixpoint techniques now used in the efficient implementation of recursive queries by SQL and Datalog systems. General conditions for $\mathcal{P}reM$ to hold is a given program were discussed in [18].

The $\mathcal{P}reM$ property provides a very useful sufficient condition for recursive queries with aggregates to have a declarative SMS that can be computed quite efficiently using the techniques currently used for monotonic queries. Nevertheless, many queries have SMS although they do not satisfy $\mathcal{P}reM$. For instance, consider Example 2 with Dy < 40 added to r_2. Then, instead of using an eager derivation we can omit deriving dist(c, 20) at Step 2, and derive dist(c, 28) at Step 3: this completes a judicious derivation free of provisional atoms that thus delivers its stable model. In this revised program, the $\mathcal{P}reM$ property is lost and and eager fixpoint derivation does not produce its stable model, whereby other techniques, such as those discussed in the next sections are needed to compute it.

Programs with MIN. The formal properties of programs with min can be derived by duality from those of programs with max and imply that many min programs of practical interest have SMS. For instance, if we revise our examples to use min instead of max, we find that $\mathtt{r_2}$ still satisfies $\mathcal{P}reM$ after the addition of $\mathtt{Dx} < \mathtt{40}$, and therefore an eager fixpoint computation can be used to compute its stable model. Moreover, the presence of cycles of positive length in $\mathtt{graph}(\_\,,\_\,)$ does not compromise the SMS of our min programs, since the second time a node Y in the cycle is visited during derivation, its larger Dy value is discarded, whereas in max queries larger values turn previous values into provisional ones. A wide range of classical graph algorithms can be expressed concisely using aggregates in recursive rules that have the $\mathcal{P}reM$ property and are thus conducive to efficient and scalable implementations [8, 11, 13, 15]. In fact, along with other optimization techniques, the semi-naive fixpoint can be extended to programs with extrema, with the simple provision that the new atoms generated at each step will replace any provisional atom they dominate.

3 Semantics of Programs Using the Continuous Count and Final Count Aggregates

The OLAP functions of SQL introduced the continuous count aggregate which returns all positive integers up to the cardinality of the set. Continuous count will be denoted by mcnt, since it is monotonic in the lattice of set containment, i.e. if $\mathtt{S1} \subseteq \mathtt{S2}$ then $\mathtt{mcnt(S1)} \subseteq \mathtt{mcnt(S2)}$. The final count fcnt that computes the cardinality of a set can be expressed as the max of the monotonic count on the set. For instance, with $\mathtt{bintbl}(\_\,,\_\,)$ denoting an arbitrary binary table the following rule computes the cardinality of the set of distinct Y-values associated with a group-by value X:

$$\mathtt{r} : \mathtt{mcagr}(\mathtt{X}, \mathtt{mcnt}\langle\mathtt{Y}\rangle) \leftarrow \mathtt{bintbl}(\mathtt{X}, \mathtt{Y}). \tag{1}$$

The semantics of this rule is as follows:

Definition 1 (*Defining the semantics of continuous monotonic count* mcnt **):**

$$\begin{aligned}
&\mathtt{r_a} : \mathtt{mcnt}(\mathtt{X}, \mathtt{0}) \leftarrow && \mathtt{bintbl}(\mathtt{X}, _\,).\\
&\mathtt{r_b} : \mathtt{mcnt}(\mathtt{X}, \mathtt{C1}) \leftarrow && \mathtt{mcnt}(\mathtt{X}, \mathtt{C}), \mathtt{bintbl}(\mathtt{X}, \mathtt{Y}), \mathtt{onenext}(\mathtt{Y}, (\mathtt{X}), \mathtt{C}), \mathtt{C1} = \mathtt{C} + \mathtt{1}.\\
&\mathtt{r_c} : \mathtt{mcagr}(\mathtt{X}, \mathtt{max}\langle\mathtt{C}\rangle) \leftarrow && \mathtt{mcnt}(\mathtt{X}, \mathtt{C}).
\end{aligned}$$

The predicate $\mathtt{onenext}(\mathtt{Y}, (\mathtt{X}), \mathtt{C})$ guarantees that, for each group-by value X , each value Y is counted exactly once. In Datalog this can be expressed by replacing $\mathtt{onenext}(\mathtt{Y}, (\mathtt{X}), \mathtt{C})$ in $\mathtt{r_b}$ with the pair of goals $\mathtt{choice}((\mathtt{X}, \mathtt{C}), \mathtt{Y}), \mathtt{choice}((\mathtt{X}, \mathtt{Y}), \mathtt{C})$, since programs that use the choice construct have SMS [7]. In the actual implementations, the use of the get_next construct, that visits the tuples in bintb exactly once, guarantees that this constraint is never violated. Also observe that the zero count assigned by $\mathtt{r_a}$ is always eliminated by $\mathtt{r_c}$. Thus the semantics of the final count that compute the cardinality of the set can now be defined using mcnt and max. For example, the semantics of the following rule:

$$\mathtt{fcagr}(\mathtt{X}, \mathtt{fcnt}\langle\mathtt{Y}\rangle) \leftarrow \mathtt{bintbl}(\mathtt{X}, \mathtt{Y}).$$

is defined by the following two rules:

$$\begin{aligned}
&(\mathtt{a}) : \mathtt{magr}(\mathtt{X}, \mathtt{mcnt}\langle\mathtt{Y}\rangle) \leftarrow \mathtt{bintbl}(\mathtt{X}, \mathtt{Y}).\\
&(\mathtt{b}) : \mathtt{fcagr}(\mathtt{X}, \mathtt{max}\langle\mathtt{C}\rangle) \leftarrow \mathtt{magr}(\mathtt{X}, \mathtt{C}).
\end{aligned}$$

A Bill of Materials (BoM) Example. Here, $\mathtt{basic}(\mathtt{Part}, \mathtt{Cost})$ is the price charged by the supplier of a part.

Example 4 (basic *describes the cost of basic parts, and* arc *defines the part-subpart graph*)**:**

$$\begin{aligned}
&\mathtt{basic}(\mathtt{a}, 6.2).\ \mathtt{basic}(\mathtt{b}, 9.4).\ \mathtt{basic}(\mathtt{c}, 13.2).\ \mathtt{basic}(\mathtt{d}, 4.8).\ \mathtt{basic}(\mathtt{e}, 4.8).\\
&\mathtt{arc}(\mathtt{a}, \mathtt{f}).\ \ \mathtt{arc}(\mathtt{b}, \mathtt{f}).\ \mathtt{arc}(\mathtt{c}, \mathtt{f}).\ \mathtt{arc}(\mathtt{c}, \mathtt{g}).\ \mathtt{arc}(\mathtt{d}, \mathtt{g}).\ \mathtt{arc}(\mathtt{e}, \mathtt{g}).\ \mathtt{arc}(\mathtt{f}, \mathtt{g}).
\end{aligned}$$

Assume now that we introduce the (somewhat artificial) notion of complex assemblies as those which are assembled from three or more components which are either basic parts or complex subassemblies. Then the following query returns the complex assemblies in our BoM.

Example 5 (*Find the assemblies and the total count of basic parts they contain when this is* ≥ 3*):*

$$\begin{array}{ll}
\rho_1 : \texttt{cassb}(\texttt{To}, 3) \leftarrow & \texttt{basic}(\texttt{To}, _\,), \texttt{C} = 3. \\
\rho_2 : \texttt{cardc}(\texttt{To}, \texttt{fcnt}\langle\texttt{Frm}\rangle) \leftarrow & \texttt{cassb}(\texttt{Frm}, _\,), \texttt{arc}(\texttt{Frm}, \texttt{To}). \\
\rho_3 : \texttt{cassb}(\texttt{To}, \texttt{Totcnt}) \leftarrow & \texttt{cardc}(\texttt{To}, \texttt{Totcnt}), \texttt{Totcnt} \geq 3.
\end{array}$$

The semantics of this program is defined by the following program obtained by expanding ρ_2 into rules ρ_{2a} and ρ_{2b} that express the computation of fcnt aggregate via mcnt and max:

Example 6 (*Defining the semantics of* fcnt *in Example 5 as the* max *of* mcnt. **):**

$$\begin{array}{ll}
\rho_1 : \texttt{cassb}(\texttt{To}, 2000) \leftarrow & \texttt{basic}(\texttt{To}, _\,). \\
\rho_{2\texttt{a}} : \texttt{cardc}(\texttt{To}, \texttt{mcnt}\langle\texttt{Frm}\rangle) \leftarrow & \texttt{cassb}(\texttt{Frm}, _\,), \texttt{arc}(\texttt{Frm}, \texttt{To}). \\
\rho_{2\texttt{b}} : \texttt{mxcard}(\texttt{To}, \texttt{max}\langle\texttt{C}\rangle) \leftarrow & \texttt{cardc}(\texttt{To}, \texttt{C}). \\
\rho_3 : \texttt{cassb}(\texttt{To}, \texttt{Totcnt}) \leftarrow & \texttt{mxcard}(\texttt{To}, \texttt{Totcnt}), \texttt{Totcnt} \geq 3.
\end{array}$$

The max aggregate is the only non-monotonic construct in this program. Therefore, to achieve SMS, we must find a judicious derivation that does not produce any provisional max. One such derivation could e.g. use ρ_1 and ρ_{2a} to produce $\texttt{cardc}(\texttt{f}, 3)$ and $\texttt{cardc}(\texttt{g}, 3)$. At this point, our judicious derivation produces $\texttt{mxcard}(\texttt{f}, 3)$ but not $\texttt{mxcard}(\texttt{g}, 3)$. Then, from $\texttt{mxcard}(\texttt{f}, 3)$ we derive $\texttt{cassb}(\texttt{f}, 3)$. From this, we derive $\texttt{cardc}(\texttt{g}, 4)$, which yields $\texttt{mxcard}(\texttt{g}, 4)$ and $\texttt{cassb}(\texttt{g}, 4)$. Since this derivation has produced no provisional max, SMS is guaranteed. On the other hand, an eager computation woud have used $\texttt{cardc}(\texttt{g}, 3)$ to produce $\texttt{mxcard}(\texttt{g}, 3)$ before deriving $\texttt{mxcard}(\texttt{g}, 4)$ and $\texttt{cassb}(\texttt{g}, 4)$, which remove the provisional $\texttt{mxcard}(\texttt{g}, 3)$ but not $\texttt{cassb}(\texttt{g}, 3)$. Thus we cannot use the eager fixpoint technology of our current systems to compute the stable model for this program. As we look for a solution, we see that there is a simple one that consists in revising ρ_3 into the following rule:

$$\rho_3' : \texttt{cassb}(\texttt{To}, \texttt{max}\langle\texttt{Totcnt}\rangle) \leftarrow \quad \texttt{mxcard}(\texttt{To}, \texttt{Totcnt}), \texttt{Totcnt} \geq 3.$$

The *max-enhanced* version of the program so obtained has the same SMS as the original program and its eager fixpoint computation produces its stable model, thus we will say that our program is *quasi-resilient*. While many programs of interest are quasi resilient, others are not, and no max enhancement will deliver their SMS. For instance, say that we add the additional goal $\texttt{Totcnt}{<}4$; then, the stable model for this program contains $\texttt{cassb}(\texttt{f}, 3)$, but it does neither contains $\texttt{cassb}(\texttt{g}, 3)$ nor $\texttt{cassb}(\texttt{g}, 4)$. However the fixpoint of its max-enhanced version now contains $\texttt{cassb}(\texttt{g}, 3)$. Thus the max-enhancement here neither preserves the original SMS nor it makes it resilient. To address this situation, users must be provided with (i) criteria to recognize programs that can be max-enhanced into equivalent programs that are resilient, and (ii) more general implementation methods for programs that have a SMS but do not belong to group (i). The notion of *Reverse Premappability* (R$\mathcal{P}reM$) provides a simple answer to (i). With T denoting the the ICO of one or more rule, we will say that R $\mathcal{P}reM$ holds for T if $\gamma(T(I)) = T(\gamma(I))$ for every I. Since $\mathcal{P}reM$ holds for max in rule ρ_3, max can be introduced into this rule without changing the result; indeed, it only enforces the max constraint on atoms generated from rule $\rho_{2\texttt{b}}$, i.e. atoms that already satisfy the max constraint. However if we instead use rule ρ_3', then a max atom, such as $\texttt{mxcard}(\texttt{g}, 4)$ obtained from $\rho_{2\texttt{b}}$, will be filtered out by the < 4 condition, whereby ρ_3' will now return $\texttt{mxcard}(\texttt{g}, 3)$ as max: a clear violation of SMS. Therefore, to realize the SMS for the program with the <4 condition, we need different solutions, such as the one that relies on the *pre-counting* technique discussed in the next section.

Dealing with Duplicates. Duplicates are immaterial for extrema because of their idempotence property, but not for the other aggregates a special notation is needed to specify that duplicates are not excluded from the computation. Say for instance that we have a ternary table $\texttt{terntbl}(\texttt{A1}, \texttt{A2}, \texttt{A3})$ and for each value of A1 we

would like to count all occurrences of A2, where every occurrences of A2 associated with different A3 value contributes to the count. Then, instead of the following rule,

$$\texttt{dbagr}(\texttt{X}, \texttt{fcnt}\langle[\texttt{Y}, \texttt{Z}]\rangle) \leftarrow \texttt{terntbl}(\texttt{X}, \texttt{Y}, \texttt{Z}). \tag{2}$$

we can use the rule (3) below with the special duplicate notation that will also be used for sum and average:

$$\texttt{dbagr}(\texttt{X}, \texttt{fcnt}\langle\texttt{Y}, \texttt{Z}\rangle) \leftarrow \texttt{terntbl}(\texttt{X}, \texttt{Y}, \texttt{Z}). \tag{3}$$

4 Queries Using Sum and Average.

The semantics of the sum of the elements in a set can be specified by adding up its elements while counting them so that the sum value associated with the final count can be returned.[2]

Consider for instance the following example that for each assembly derives the total of the costs of the basic parts it uses, by adding up those of its subparts. The notation $\texttt{sum}\langle\texttt{Cost}, \texttt{Frm}\rangle)$ denote that duplicate Cost from different subparts Frm all contribute to the sum.

Example 7 (*For each assembly, find the total cost of the basic parts it uses):*

$$\begin{array}{ll}
\rho_1 : \texttt{pcst}(\texttt{To}, \texttt{Cost}) \leftarrow & \texttt{basic}(\texttt{To}, \texttt{Cost}). \\
\rho_2 : \texttt{ragr}(\texttt{To}, \texttt{sum}\langle\texttt{Cost}, \texttt{Frm}\rangle) \leftarrow & \texttt{pcst}(\texttt{Frm}, \texttt{Cost}), \texttt{arc}(\texttt{Frm}, \texttt{To}). \\
\rho_3 : \texttt{pcst}(\texttt{To}, \texttt{Cost}) \leftarrow & \texttt{ragr}(\texttt{To}, \texttt{Cost}).
\end{array}$$

Now, the computation of $\texttt{sum}\langle\texttt{Cost}, \texttt{Frm}\rangle$ requires (a) an initial step where count and sum are initialized to zero, (b) an iterative step that adds one to the current count and adds the new Cost to the current sum, and (c) a final max step that selects the final count, which is used to return the final sum value associated with it.

Example 8 (*The max-based formal semantics for Example 7):*

$$\begin{array}{ll}
\rho_1 : \texttt{pcst}(\texttt{To}, \texttt{Cst}) \leftarrow & \texttt{basic}(\texttt{To}, \texttt{Cst}). \\
\rho_{2\texttt{a}} : \texttt{rsum}(\texttt{To}, \texttt{C}, \texttt{S}) \leftarrow & \texttt{pcst}(\texttt{Frm}, __), \texttt{arc}(\texttt{Frm}, \texttt{To}), \texttt{C} = 0, \texttt{S} = 0. \\
\rho_{2\texttt{b}} : \texttt{rsum}(\texttt{To}, \texttt{C1}, \texttt{S1}) \leftarrow & \texttt{pcst}(\texttt{Frm}, \texttt{Cst}), \texttt{arc}(\texttt{Frm}, \texttt{To}), \texttt{rsum}(\texttt{To}, \texttt{C}, \texttt{S}), \\
& \texttt{onenext}(\texttt{Cst}, \texttt{Frm}, (\texttt{To}), \texttt{C}), \texttt{C1}{=}\texttt{C}{+}1, \texttt{S1}{=}\texttt{S} + \texttt{Cst}. \\
\rho_{2\texttt{c}} : \texttt{ragr}(\texttt{To}, \texttt{max}\langle\texttt{C}\rangle) \leftarrow & \texttt{rsum}(\texttt{To}, \texttt{C}, __). \\
\rho_3 : \texttt{pcst}(\texttt{To}, \texttt{Cst}) \leftarrow & \texttt{ragr}(\texttt{To}, \texttt{Fcnt}), \texttt{rsum}(\texttt{To}, \texttt{Fcnt}, \texttt{Cst}).
\end{array}$$

To define the semantics of average, rule ρ_3 above will be modified to return Cst/Fcnt instead of Cst.

As we now investigate the declarative and operational semantics of this example, we see that we can assume that no cycle exists in the BoM part-subpart graph, no provisional max value is ever generated by ρ_{2c} and SMS is thus guaranteed. However, when it comes to operational semantics, the situation of sum is quite different from that of count; this can be easily seen by comparing rule $\rho_{2\texttt{a}}$ in Example 6, where successive count values are ignored by $\texttt{cassb}(\texttt{From}, \_\_)$, against rule $\rho_{2\texttt{a}}$ in Example 8 in which successive values are cumulatively added incorrectly to the sum. Thus, except for special cases, such as that of perfectly balanced trees, an eager computation will not deliver the correct sum value. To realize SMS, we must therefore revise the original program to make sure that, for each node, the Cst contributions of its predecessors are added all at once. One technique to achieve that is the *pre-counting* approach used in [12] which will compute the sum at a node only after the sum is computed at each of its immediate predecessors. To implement this technique, the program in Example 7 is revised into that in Example 9, below, by the addition of rule $\bar{\rho}_0$ that precomputes the in-degree of each node. Then, ρ_2 is modified into $\bar{\rho}_2$ to keep count of the number of a node's immediate predecessors that have so far contributed to its sum. Thus, the correct final sum value used in $\bar{\rho}_3$ is the one obtained when count is equal to the in-degree of the node. In passing, observe that $\bar{\rho}_2$ also illustrates how multiple aggregates sharing the same group-by value can be specified in the head of a rule.

[2]Using the max of the continuous count on the set elements does not identify their sum when negative numbers are present.

Example 9 (*Cost of nodes by pre-computing the number of their incoming arcs):*

$$\begin{array}{ll}
\bar{\rho}_0 : \texttt{indgr}(\texttt{To}, \texttt{fcnt}\langle\texttt{Frm}\rangle) \leftarrow & \texttt{arc}(\texttt{Frm}, \texttt{To}). \\
\bar{\rho}_1 : \texttt{pcst}(\texttt{To}, \texttt{Cost}) \leftarrow & \texttt{basic}(\texttt{To}, \texttt{Cost}). \\
\bar{\rho}_2 : \texttt{ragr}(\texttt{To}, \texttt{sum}\langle\texttt{Cost}, \texttt{Frm}\rangle, \texttt{fcnt}\langle\texttt{Frm}\rangle) \leftarrow & \texttt{pcst}(\texttt{Frm}, \texttt{Cost}), \texttt{arc}(\texttt{Frm}, \texttt{To}). \\
\bar{\rho}_3 : \texttt{pcst}(\texttt{To}, \texttt{Cost}) \leftarrow & \texttt{ragr}(\texttt{To}, \texttt{Cost}, \texttt{NNods}), \texttt{indgr}(\texttt{To}, \texttt{NNods}).
\end{array}$$

Thus we have now a program where the eager FPC realizes its SMS, returning the same $\texttt{pcst}(\texttt{To}, \texttt{Cost})$ atoms as the original program. Pre-counting is also applicable to programs with count and average and represents a simple technique to derive an efficient and scalable fixpoint computation for programs that have a declarative SMS. However, in the simple form we have discussed, pre-counting relies on the assumption that all the nodes are reachable form basic parts. When that is not the case, more complex programs could be used to compute the actual count of immediate predecessors of a node reachable form the basic parts. Alternatively, the Group-by Layering technique described in [16] can be used to avoid the generation of provisional values, by delaying the derivation of each group-by node until a derivation step where the last of its predecessors is computed. In acyclic graphs, this step could be determined using topological sorting, but this requires a complex Datalog program and implies the serialization of nodes that could be computed in parallel. The technique described in [16] instead computes for each node its maximum distance from `basic` nodes and is amenable to parallelism.

5 Parallelism and Layered Computation.

By applying pre-counting on Example 7, we obtained Example 9 where the computation a new node is started only after it is completed at its predecessors. In many applications of interest, this kind of revision is not needed since the completion conditions that enable the fixpoint computation to realize SMS follow directly from the structure of the program and the parallelization strategy used by the system. Among such applications, we find those using algorithms such as Markov-Chains, Lloyd's Clustering and Batch Gradient Descent.

Markov Chains. This algorithm is interesting because of its similarity to the Page Rank algorithm that led to Map-Reduce. Thus, in Example 10, we assume a database of facts $\texttt{mov}(\texttt{Frm}, \texttt{To}, \texttt{Perc})$, where `Perc` denotes the fraction of population that every year relocates from city `Frm` to city `To`. For each city, there is also a non-zero arc from the city back to itself showing the fraction of people who remain in the same city. Therefore, the sum of `Perc` for the arcs leaving a city (i.e., a node) is equal to one.

Now, assuming that initially every city has a population of $100{,}000$ people we would like to determine how the population evolves over the years. Also to assure termination, we stop the computation after 999 steps (i.e., a number of steps that is normally sufficient for the computation to either converge to a final state or reveal that no convergence should be expected). Then, we can use the following program, where $\texttt{sum}\langle\texttt{In}, \texttt{Frm}\rangle$ adds up the `In` contributions from all `Frm` cities, without discarding duplicate `In` values coming from different `Frm` cities:

Example 10 (The Markov Chains algorithm):

$$\begin{array}{ll}
\rho_1 : \texttt{markv}(1, \texttt{Cit}, \texttt{Pop}) \leftarrow & \texttt{mov}(\texttt{Cit}, \texttt{To}, __), \texttt{Pop} = 100000. \\
\rho_2 : \texttt{next}(\texttt{J}, \texttt{To}, \texttt{sum}\langle\texttt{In}, \texttt{Frm}\rangle) \leftarrow & \texttt{markv}(\texttt{J}, \texttt{Frm}, \texttt{Pop}), \texttt{mov}(\texttt{Frm}, \texttt{To}, \texttt{Perc}), \texttt{In} = \texttt{Pop} \times \texttt{Perc}. \\
\rho_3 : \texttt{markv}(\texttt{J1}, \texttt{Cit}, \texttt{Pop}) \leftarrow & \texttt{next}(\texttt{J}, \texttt{Cit}, \texttt{Pop}), \texttt{J} \leq 999, \texttt{J1} = \texttt{J} + 1.
\end{array}$$

Observe that we have here a program that is locally stratified by the value of the first argument in `markv`, whereby SMS hold. The same conclusion follows by representing as a directed arc the dependency from the group-by arguments $\texttt{markv}(\texttt{J}, \texttt{Frm}, \_\_)$ in ρ_2 to $\texttt{markv}(\texttt{J1}, \texttt{Cit}, \_\_)$ in the head of ρ_3. Since $\texttt{J1} = \texttt{J}+1$ the graph so established is acyclic, and this provides yet another proof that declarative SMS holds.

Now, for an operational semantics that correctly realizes the SMS of this program, we can observe that the number of incoming arcs representing the population that migrate into a city each year remains the same, and thus

we can use the pre-counting approach previously described. Then, the head ρ_2 will be extended with an additional argument that counts the number of contributions added so far. Them, ρ_3 ignores the sum values produced by ρ_2, until the count argument equals the pre-counted in-degree of the node. The stable model for the program generated by this pre-computing transformation is then computed by the eager derivation that is conducive to scalability via stale-synchronous parallelism [4, 16].

However, other solutions are available and actually preferable in Datalog systems that support more advanced operational semantics. Indeed some Datalog compilers [2] will recognize the explicit local stratification that holds for the group-by argument in our rules, and arrange for an evaluation where all computations at level J1=J+1 are performed after the $\mathtt{J}^{\mathtt{th}}$-level ones are completed. In these Datalog systems, the stable model for Example 10 can computed with no revision required in the program. The same conclusion holds for systems designed to support bulk-synchronous parallelism (BSP) in which a new distribution-computation cycle at level J+1 is not started until the completion of the previous cycle, in which aggregates at group-by level J were evaluated. This approach achieves high parallel performance [9].

Applications Requiring a Combination of Different Aggregates. Queries that combine multiple aggregates can express concisely a broad spectrum of advanced algorithms that support graph, data mining and ML applications with superior performance and scalability [3, 8–15]. For instance, we next discuss Lloyd's clustering algorithm that combines sum, average and min aggregates.

K-means Clustering: We are given a large set of D-dimensional points. Each point is described by a unique Pno and its coordinate values in each of the D dimensions, i.e., by D facts conforming to the following template: point(Pno, Dim, Value). We also have a small set of centroids, for which we generate an initial assignment center(0, Cno, Dim, Val) by the predicate init(Cno, Dim, Val) defined using any of the simple techniques described in the literature. Then, Lloyd's clustering algorithm can be expressed concisely as shown in Example 11.

Example 11 (Clustering a lá Lloyd.):

$$
\begin{array}{ll}
\mathtt{r_0 : center(J, Cno, Dim, Val)} \leftarrow & \mathtt{init(Cno, Dim, Val), J = 0.} \\
\mathtt{r_1 : cdist(J1, Pno, Cno, sum\langle SqD\rangle)} \leftarrow & \mathtt{point(Pno, Dim, Val), center(J0, Cno, Dim, CVal).} \\
 & \mathtt{SqD{=}(Val - Cval) * (Val{-}Cval), J1 = J0 + 1.} \\
\mathtt{r_2 : mdist(J2, Pno, min\langle[DSm, Cno]\rangle)} \leftarrow & \mathtt{cdist(J1, Pno, Cno, DSm), J2 = J1 + 1.} \\
\mathtt{r_3 : center(J0, Cno, Dim, avg\langle Val, Pno\rangle)} \leftarrow & \mathtt{mdist(J2, Pno, [\_, Cno]), points(Pno, Dim, Val),} \\
 & \mathtt{J2 \leq 999, J0 = J2 + 1.}
\end{array}
$$

At each step J Example 11 computes the quadratic distance of each point from each centroid, so that $\mathtt{r_2}$ can select for each point the centroid closest to it. Then, for each centroid, $\mathtt{r_3}$ recomputes its position by averaging the coordinates of the points that have this as their nearest centroid[3]:

This program is structurally similar to the Markov Chains program of Example 10, and the two programs have similar properties in terms of declarative and operational semantics. In fact, the first argument in the heads of their rules define an explicit stratification that implies SMS. Moreover, since the cardinality of Pno and Cno sets remain constant throughout the computation, we can use the pre-counting approach to construct the stable model by an eager fixpoint computation that is conducive to SSP scalability. However, with bulk-synchronous parallelish (BSP) the SMS of this program can be realized quite naturally and efficiently by simply synchronizing the BSP steps with the first arguments in the head of the rule, i.e., with J0, J1, J2. This guarantees that the declarative and operational semantics of our programs are completely aligned and conducive to efficient implementations

[3]Thus min assume a total ordering where $[\mathtt{X1}, \mathtt{Y1}] \leq [\mathtt{X2}, \mathtt{Y2}]$ holds if $\mathtt{X1} < \mathtt{X2}$ or if $\mathtt{X1} = \mathtt{X2}$ and $\mathtt{Y1} \leq \mathtt{Y2}$. This ordering is easily extended to lists of arbitrary length.

that are scalable via BSP. In fact, Lloyd's algorithm is among the several data mining algorithms implemented on Apache Spark using Datalog with recursive aggregates. The performance and scalability of these algorithms have undergone extensive experimentation, showing that they perform as well or better than the same algorithms expressed by lengthy[4] procedural-language programs [1, 8, 9, 11–13, 15].

Expressing ML Applications The problem of supporting efficient and scalable ML applications concisely expressed as Datalog queries with aggregates was studied in [12]. For instance, to support gradient descend, we can use a verticalized representation $\texttt{vtrain}(\texttt{Id}, \texttt{C}, \texttt{V}, \texttt{Y})$ for the training set, where Id denotes the id of a training instance, Y denotes its label, C and V denote the dimension and the value along that dimension, respectively. Then a Batch Gradient Descent application can be expressed by the program in Example 12 where

- $\texttt{model}(\texttt{J}, \texttt{C}, \texttt{P})$ is the training model in verticalized form, where J is the iteration counter C is a dimension in the model, and P is the parameter value for that dimension.
- $\texttt{gradient}(\texttt{J}, \texttt{C}, \texttt{D})$ contains the gradient result G at iteration J for the $\texttt{C}^{th}$ dimension.
- $\texttt{predict}(\texttt{J}, \texttt{Id}, \texttt{YP})$ represents the intermediate prediction results, where Id denotes the id of the training instance, and YP its the predicted y value at iteration J.

Firstly, the model is initialized according to some predefined mechanisms in $\texttt{r}_0$ (Here we use 0.01). Then the function f is used to make predictions on all training instances according to the model obtained in the previous iteration in $\texttt{r}_1$. Next the gradient is computed by the function g (derived according to the loss function L) using the predicted results in $\texttt{r}_2$. Finally, in $\texttt{r}_3$ the model is updated w.r.t the gradients (and optional regularization Ω). Here, lr denotes the learning rate and n is the number of training instances. Then, the training process moves on to the next iteration.

Example 12 (Batch Gradient Descent (BGD)):

$$
\begin{array}{ll}
\texttt{r}_0 : \texttt{model}(\texttt{J}, \texttt{C}, \texttt{P}) \leftarrow & \texttt{vtrain}(\_, \texttt{C}, \_, \_), \texttt{J} = 0, \texttt{P} = 0.01. \\
\texttt{r}_1 : \texttt{predict}(\texttt{J}, \texttt{Id}, \texttt{sum}\langle \texttt{Y0} \rangle) \leftarrow & \texttt{vtrain}(\texttt{Id}, \texttt{C}, \texttt{V}, \_), \texttt{model}(\texttt{J}, \texttt{C}, \texttt{P}), \texttt{Y0} = \texttt{f}(\texttt{V}, \texttt{P}), \texttt{J1}{=}\texttt{J}{+}1. \\
\texttt{r}_2 : \texttt{gradient}(\texttt{J2}, \texttt{C}, \texttt{sum}\langle \texttt{G0} \rangle) \leftarrow & \texttt{vtrain}(\texttt{Id}, \texttt{C}, \texttt{V}, \texttt{Y}), \texttt{predict}(\texttt{J1}, \texttt{Id}, \texttt{YP}), \texttt{G0}{=}\texttt{g}(\texttt{YP}, \texttt{Y}, \texttt{V}), \texttt{J2}{=}\texttt{J1}{+}1. \\
\texttt{r}_3 : \texttt{model}(\texttt{J}, \texttt{C}, \texttt{NP}) \leftarrow & \texttt{model}(\texttt{J}, \texttt{C}, \texttt{P}), \texttt{gradient}(\texttt{J}, \texttt{C}, \texttt{G}), \\
 & \texttt{NP} = \texttt{P} - lr * (\texttt{G}/n + \Omega(\texttt{P})), \texttt{J}{=}\texttt{J3}{+}1.
\end{array}
$$

As in previous two examples, the J argument in the group-by establishes a structure that guarantees SMS and superior scalability via BSP. The same is true for a wide spectrum of ML algorithms that can be expressed with different functions f, g, Ω in above program. In particular, the Mini-batch Gradient Descent (MGD) require only minor changes to above queries [12].

6 Conclusion

The usage of aggregates in recursive Datalog programs entails a concise expression for very powerful algorithms that combine a formal declarative SMS with a powerful and scalable fixpoint-based operational semantics. This paper has unified the semantics of programs with different aggregates by turning them all into equivalent programs with extrema, and thus significantly simplified the verification of their SMS by reducing it to the exclusion of provisional extrema from the derivation. Using the $\mathcal{P}reM$ property we identified a large class of resilient or quasi-resilient programs for which the current implementation techniques used in Datalog provide an efficient and SSP-scalable implementation of their SMS. For programs that do not belong to this class, the paper proposed simple rewriting techniques, such as pre-counting to realign their operational semantics with declarative one. Finally in many programs of practical interest, the synchronized execution produced by SSP also guarantees the

[4] For instance, the corresponding procedural version requires twenty-fold the lines of codes used in Example 11.

correct derivation of their SMS. These findings suggest that many big-data algorithms that are now developed using procedural languages can instead be developed directly using query languages, including SQL, since recursive Datalog queries with aggregates, can be translated into equivalent SQL queries, as discussed in the appendix.

References

[1] Foto N Afrati, Vinayak Borkar, Michael Carey, et al. Map-reduce extensions and recursive queries. In *EDBT*, pages 1–8. ACM, 2011.

[2] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang, and Carlo Zaniolo. The deductive database system LDL++. *TPLP*, 3(1):61–94, 2003.

[3] Ariyam Das, Youfu Li, Jin Wang, Mingda Li, and Carlo Zaniolo. Bigdata applications from graph analytics to machine learning by aggregates in recursion. In *ICLP'19*, 2019.

[4] Ariyam Das and Carlo Zaniolo. A case for stale synchronous distributed model for declarative recursive computation. In *35th International Conference on Logic Programming*, ICLP'19, 2019.

[5] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth Joint International Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

[6] Michael Gelfond and Yuanlin Zhang. Vicious circle principle and logic programs with aggregates. *Theory Pract. Log. Program.*, 14(4-5):587–601, 2014.

[7] Sergio Greco, Carlo Zaniolo, and Sumit Ganguly. Greedy by choice. In Moshe Y. Vardi and Paris C. Kanellakis, editors, *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California, USA*, pages 105–113. ACM Press, 1992.

[8] Jiaqi Gu, Yugo Watanabe, William Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark. In *ACM SIGMOD Int. Conference on Management of Data, SIGMOD 2019*, 2019.

[9] Youfu Li, Jin Wang, Mingda Li, Ariyam Das an Jiaqi Gu, and Carlo Zaniolo. Kddlog: Performance and scalability in knowledge discovery by declarative queries with aggregates. *International Conference on Data Enginering, ICDE*, 2021.

[10] Jiwon Seo, Stephen Guo, and Monica S Lam. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289. IEEE, 2013.

[11] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with Datalog queries on Spark. In *SIGMOD*, pages 1135–1149. ACM, 2016.

[12] Jin Wang, Jiacheng Wu, Mingda Li, Jiaqi Gu, Ariyam Das, and Carlo Zaniolo. Formal semantics and high performance in declarative machine learning using datalog. *VLDB J.*, 2021.

[13] Jin Wang, Guorui Xiao, Jiaqi Gu, Jiacheng Wu, and Carlo Zaniolo. RASQL: A powerful language and its system for big data applications. In *SIGMOD 2020 Int. Conference on Management of Data*, pages 2673–2676. ACM, 2020.

[14] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *PVLDB*, 8(12):1542–1553, 2015.

[15] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. Scaling up the performance of more powerful datalog systems on multicore machines. *VLDB J.*, 26(2):229–248, 2017.

[16] Carlo Zaniolo, Ariyam Das, Youfu Li, Mingda Li, and Jin Wang. Declarative and operational semantics for datalog programs with aggregates. In *Submitted for Publication*, pages 1–15, 2021.

[17] Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, Alexander Shkapsky, and Tyson Condie. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *TPLP*, 17(5-6):1048–1065, 2017.

[18] Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, Alexander Shkapsky, and Tyson Condie. Declarative bigdata algorithms via aggregates and relational database dependencies. In *Proceedings of the 12th Alberto Mendelzon International Workshop on Foundations of Data Management, Cali, Colombia, May 21-25, 2018.*, 2018.

A Aggregates in Recursive SQL Queries

The single source shortest path Datalog query of Example 1 can be expressed in SQL.

Example 13 (SSSP on base table: `edge(Src: int, Dst: int, Cost: double)` **by stratified SQL):**

```
WITH recursive sssp (Dst, Cost) AS
  (SELECT "a", 0)
   UNION
  (SELECT edge.Dst, sssp.Cost + edge.Cost
   FROM sssp, edge WHERE sssp.Dst = edge.Src)
SELECT Dst, MIN(Cost) AS minCost FROM sssp GROUP BY Dst
```

To express the query of Example 2 by a compact representation, we will assume that the aggregate columns, such as MIN(Cost) are implicitly grouped by the other columns in `SELECT`: i.e., by `Dst` for the example at hand.

Example 14 (SSSP on base table: `edge(Src: int, Dst: int, Cost: double)` **by unstratified SQL):**

```
WITH recursive sssp (Dst, min(Cost) AS minCost GROUP BY Dst) AS
  (SELECT "a", 0)
   UNION
  (SELECT edge.Dst, sssp.minCost + edge.Cost
   FROM sssp, edge WHERE sssp.Dst = edge.Src)
   SELECT Dst, minCost FROM sssp
```

Similar translations apply to the other examples and aggregates discussed in the paper, with the provision that the keywords `ALL` and `DISTINCT` will be added to specify their behavior in the presence of duplicates.