The VLDB Journal
DOI 10.1007/s00778-012-0299-1

REGULAR PAPER

Extending the power of datalog recursion

Mirjana Mazuran · Edoardo Serra · Carlo Zaniolo

Received: 21 February 2012 / Revised: 26 September 2012 / Accepted: 12 October 2012

Abstract Supporting aggregates in recursive logic rules represents a very important problem for Datalog. To solve this problem, we propose a simple extension, called Datalog^{FS} (Datalog extended with frequency support goals), that supports queries and reasoning about the number of distinct variable assignments satisfying given goals, or conjunctions of goals, in rules. This monotonic extension greatly enhances the power of Datalog, while preserving (i) its declarative semantics and (ii) its amenability to efficient implementation via differential fixpoint and other optimization techniques presented in the paper. Thus, Datalog^{FS} enables the efficient formulation of queries that could not be expressed efficiently or could not be expressed at all in Datalog with stratified negation and aggregates. In fact, using a generalized notion of multiplicity called frequency, we show that diffusion models and page rank computations can be easily expressed and efficiently implemented using Datalog^{FS}.

Keywords Query languages · Logic programming · Graph algorithms

1 Introduction

Due to the emergence of many important application areas, we are now experiencing a major resurgence of interest in Datalog [1–3]. A first such research area focuses on the use of logic-based declarative specifications and design of Internet protocols and services [4]. More ambitious approaches are now being proposed that seek to develop Datalog-based foundations for parallel and distributed programming languages [1]. On the Semantic Web front, a novelty of great interest is represented by the introduction of languages, such as Linear Datalog, that support efficiently subsets of Description Logic for reasoning in ontological queries [5]. Yet another very important development is represented by the use of Datalog in program analysis [3]. Furthermore, other lines of work exploring the execution of Datalog queries in new computational environments have studied the optimization of recursive queries in the MapReduce framework [6], and the declarative and operational semantics of continuous Datalog queries in Data Stream Management Systems [7].

This torrent of new applications underscores the need to tackle and solve important Datalog problems that were recognized more than 20 years ago but still remain unsolved and restrict the range of practical effectiveness of this elegant declarative-programming paradigm. For database applications in particular, the most vexing of these problems is represented by the constraints placed upon aggregates in recursive Datalog programs. Indeed, with the introduction of OLAP functions and massive analytics for decision support and web mining, the usage of aggregates in modern information systems and web applications has grown by leaps and bounds—making limitations upon aggregates increasingly undesirable. Therefore, Datalog extensions that can improve its ability to deal with aggregates would represent a big step forward. Technically, however, the problem is very challenging since basic aggregates violate the requirement of monotonicity on which the least fixpoint semantics of

M. Mazuran
DEI, Politecnico di Milano, Milan, Italy
e-mail: mazuran@elet.polimi.it

E. Serra
DEIS, University of Calabria, Rende, Italy
e-mail: eserra@deis.unical.it

C. Zaniolo (✉)
University of California, Los Angeles, CA, USA
e-mail: zaniolo@cs.ucla.edu

Datalog is based.[1] Therefore, the many solutions proposed in the past were too complex or not general enough and have not gained wide acceptance.

In this paper, we propose Datalog^{FS} that extends Datalog by introducing frequency support goals that allow to count the distinct occurrences satisfying given goals or conjunction of goals in rules.

Consider a database of facts whose predicates are `person` and `son`, for example `person(tom). son(tom,adam)`. The following rule defines fathers with at least two sons:

$$\mathtt{twosons(X)} \leftarrow \mathtt{person(X), son(X, Y1), son(X, Y2),}\\ \mathtt{Y2} \neq \mathtt{Y1.}$$

Datalog^{FS} allows the following equivalent expression for our `twosons` rule:

$$\mathtt{twosons(X)} \leftarrow \mathtt{person(X), 2{:}[son(X, Y)].}$$

The goal, `I:[b-expression]`, will be called a frequency support goal (an *FS Goal* for short), and "`I:`" where `I` is a positive integer, will be called its *Running-FS term*. Also, the expression in the bracket is called its bracket-expression (*b-expression* for short) and can either consist of a single positive predicate or the conjunction of positive predicates.

The convenience of FS goals becomes clear if we want to find people who have a large number of sons. For instance:

$$\mathtt{sixsons(X)} \leftarrow \mathtt{person(X),\ 6{:}[son(X, Y)].}$$

will retrieve all persons who have at least six sons (i.e., 6 is the frequency support required for the predicates or conjunction of predicate within brackets).

Naturally, an equivalent rule can be expressed using the $\neq$ operator. Indeed, we can start as follows:

$$\mathtt{sixsons(X)} \leftarrow \mathtt{person(X), son(X, Y1),}\\ \mathtt{5{:}[son(X, Y2), Y2} \neq \mathtt{Y1].}$$

and then proceed inductively and obtain a rule containing six goals $\mathtt{son(X, Y_j)}$ with $\mathtt{j} = 1, \ldots, 6$, and also 6×5 goals specifying that every `Y` must be different from (i.e., $\neq$) every other `Y`. Of course, the approach based on $\neq$ becomes totally impractical when, instead of paternity between people, we have links between web pages, which can easily be in number of thousands. The COUNT aggregate might instead be used in Datalog for these applications. However, aggregates bring in the course of non-monotonicity whereby recursion becomes a problem. Our Datalog^{FS} rules with FS goals can instead be viewed as standard Horn clauses (although long and impractical ones) whereby the standard monotonicity-based semantics of negation-free Datalog is preserved. Moreover, the well-known implementation techniques used for Datalog can also be extended to Datalog^{FS} and, in combination with others introduced in Sect. 6, entail a very efficient implementation.

The next two examples clarify the meaning and the scope of variables in Datalog^{FS}. We will use the predicate `friend(X,Y)` to denote that the person with name `X` views the person with name `Y` as his/her friend (thus, the conjunct `friend(X,Y), friend(Y,X)` denotes that `X` and `Y` are mutual friends).

Example 1 Pairs of mutual friends `X, Y`, where `X` has three friends and `Y` also has three friends.

$$\mathtt{popularpair(X, Y)} \leftarrow \mathtt{friend(X, Y), friend(Y, X),}\\ \mathtt{3{:}[friend(X, V1)], 3{:}[friend(Y, V2)].}$$

□

Now, Example 2, below, requires that `X` and `Y` have at least three friends in common (unlike the previous example where `X` and `Y` are not required to have friends in common):

Example 2 Pairs of mutual friends `(X, Y)` who have at least three friends in common.

$$\mathtt{sharethree(X, Y)} \leftarrow \mathtt{friend(X, Y), friend(Y, X)}\\ \mathtt{3{:}[friend(X, V), friend(Y, V)].}$$

□

Thus, there are two kinds of variables in rules with FS goals. The first are those, such as `X` and `Y` in Example 2, that appear in the head of the rule or in goals outside the b-expressions (i.e., outside the brackets). These will be called *global* variables. Global variables are basically the universally qualified variables of the standard Horn Clauses and have the whole rule as their scope. Thus the variables `X` and `Y` are global in the rule of Example 1, and in that of Example 2 as well.

The remaining variables are those that only appear in b-expressions and their scope is *local* to the b-expression where they appear. For instance `V1` and `V2` in Example 1, and `V` in Example 2, are local variables. Local variables in the b-expression of `K:[b-expression]` can be naturally viewed as existential variables with the following minimum frequency support constraint: *There exist at least* `K` *distinct occurrences of the b-expression*. For instance, Example 2 states that there exist at least 3 distinct `V` occurrences each denoting a person who is viewed as friend by both `X` and `Y`.

It is also important to remember that the scope of such existential variables is local to the b-expression where they appear: thus, for Example 1 replacing both `V1` and `V2` with `V` would not change its meaning.

The simple examples shown so far do not use recursion and thus can also be expressed using the count aggregate. However, in the paper, we will introduce more complex examples that use FS goals in recursion. Indeed, the meaning and

[1] A good example of this problem due to Ross and Sagiv [8] is given at the end of Sect. 2—see Example 3.

efficient implementation, of Datalog programs with recursive rules, are based on their least fixpoint semantics,[2] which is only guaranteed to exist when the program rules define monotonic mappings. The solution of this problem proposed by Ross and Sagiv [8] exploits a partial ordering that is different from the set-containment ordering used in the standard definition of the semantics of Datalog and other logic programs. While several interesting programs with aggregates are in fact monotonic in the partial order used in [8], many others are not, and it is unclear how a compiler can decide which is which [10]. As we shall see next, the standard notion of monotonicity w.r.t set-containment ordering is instead used for Datalog^{FS} programs, whereby the basic properties and techniques of deductive databases (e.g., stratification and differential fixpoint) remain valid.

We will show that Datalog^{FS} is very effective at expressing a broad range of new applications requiring count-based aggregates, such as Bill of materials, social networks, and Markov Chains. Concepts such as stratification and, in general, all the techniques and methods that provide the enabling technology for traditional Datalog remain valid and effective for Datalog^{FS}, paving the way for a faster adoption and deployment of this powerful extension.

This paper is organized as follows. In the next section, we introduce the preliminary definitions, then in Sect. 3, we give the syntax and semantics of Datalog^{FS}. Then in Sect. 4, we present several examples of applications of stratified Datalog^{FS} and in Sect. 5, we characterize the expressive power of this language. Then, in Sect. 6, we proceed on a more practical vein and provide a simple plan and various optimization techniques that produce a very efficient implementation for Datalog^{FS} programs that also include arithmetic operators. In Sect. 7, we introduce the notion of scaling which allows us to reason with frequencies that are decimal numbers rather than integers. Then, in Sect. 8, we review some important new applications, such as social networks and Markov chains, that follow from these extensions. Finally, in Sect. 9 we review the related work.

2 Preliminary definitions

Although we expect that our readers are quite familiar with logic programs [9] and Datalog [11,12], we will now present a short review to clarify the formal basis upon which Datalog^{FS} is being defined.

A *logic program* (or, simply, a *program*) P is a finite set of rules. Each rule of P has the form $A \leftarrow A_1, \ldots, A_m$, where A is an atom (the *head* of the rule) and $A_1, \ldots, A_m$ are literals (the *body* of the rule). Each literal can be either a positive atom (A_i) or a negated atom ($\neg A_i$). Each atom is in the form $p(t_1, \ldots, t_n)$ where p is the predicate and $t_1, \ldots, t_n$ are terms which can be constants, variables or functions. A rule with an empty body is called a *fact*.

Given a logic program P, the Herbrand universe for P, denoted H_P, is the set of all possible ground terms recursively constructed by taking constants and function symbols occurring in P. The Herbrand Base of P, denoted B_P, is the set of all possible ground atoms whose predicate symbols occur in P and whose arguments are elements from its Herbrand universe. A *ground instance* of a rule r in P is a rule obtained from r by replacing every variable X in r by a ground term in H_P. The set of ground instances of r is denoted by *ground(r)*; accordingly, *ground(P)* denotes $\bigcup_{r \in P} ground(r)$. An interpretation I of P is a subset of B_P. A ground positive literal A (resp. negative literal $\neg A$) is true w.r.t. an interpretation I if $A \in I$ (resp. $A \notin I$). A conjunction of literals is true in an interpretation I if all literals are true in I. A ground rule is true in I if either the body conjunction is false or the head is true in I. A (Herbrand) model M of P is an interpretation that makes each ground instance of each rule in P true. A model M for P is minimal if there is no model N for P such that $N \subset M$.

Let I be an interpretation for a program P. The immediate consequence operator $T_P(I)$ is defined as the set containing the heads of each rule $r \in ground(P)$ s.t. the body of r is true in I. The semantics of a logic program P is given by its unique minimal model; this minimum model coincides with the least fixpoint $T_P^{\infty}(\emptyset)$ of T_P [9] since the immediate consequence operator $T_P(I)$ is monotonic and continuous.

The abstract semantics of logic program given above provides the formal basis for a wide spectrum of logic-based languages, including the many versions of Prolog and Datalog described in the literature. However, as discussed in more details next, Datalog systems tend to differ from Prolog in their (i) treatment of negation, (ii) specific constructs added to or banned from the language and (iii) execution models.

In particular, negated goals in the rules represent an important extension for both Prolog and Datalog; however, there are important differences since only stratified logic programs are normally allowed in Datalog. Now, a logic program P is said to be negation-stratified when all its predicates are partitioned in strata, such that the following property holds for each rule r of P: the head predicate of r belongs to a stratum that (i) is higher than the stratum of every goal in r, and (ii) strictly higher than the stratum of every negated goal in r. Stratified logic programs have a unique stable model [13], which defines their canonical semantics, and can be computed by (i) partitioning the program into an ordered number of suitable subprograms (called strata) and then (ii) computing the fixpoint of every stratum starting from the bottom and moving up to higher strata [14,15]. Another important difference is that many Datalog systems although

[2] Naturally, by “least fixpoint” of a program, we mean “least fixpoint of its immediate consequence operator” [9].

not all, disallow function symbols and lists. In the rest of the paper, we will use the term *negation-stratified* Datalog programs to denote negation-stratified logic programs where function symbols are disallowed. In reality, the expressive power of negation-stratified Datalog programs is very limited, and therefore, arithmetic and comparison operators are also supported in many Datalog systems. This is made possible by the fact that these programs have (a) a formal semantics since arithmetic predicates can be re-expressed as recursive logic programs with functions symbols and (b) efficient implementations that bypass (a) and implement arithmetic directly. Now, for Datalog^{FS}, we will operate in a similar way: we will extend *negation-stratified* Datalog ($Datalog^{\neg s}$ for short) with new FS constructs, and the proceed by (a) showing that we can re-express our programs with the new constructs as negation-stratified logic programs, thus inheriting their formal semantics, and then (b) forgetting about (a) completely to devise efficient direct implementations for our Datalog^{FS} programs. Our implementation follows the standard bottom-up execution-model used by most Datalog systems. Therefore, differential (a.k.a. semi-naive) fixpoint improvement is used, to avoid the computation of $T_P^{\infty}(\emptyset)$ redundant generation of atoms produced at previous iteration. The differential fixpoint method uses a rewriting of the original program in which, for each Intensional predicate (i.e predicates that are involved in the recursion), a new δ predicate is introduced. Each atom with δ predicate was derived only at the previous step.

Let us now show the example from [8], which we had promised in the introduction, to illustrate the problems caused by the use of traditional aggregates in recursion.

Example 3 Non-monotonic aggregates

```
p(b).    q(b).
p(a) ← 1 = count : q(X).
q(a) ← 1 = count : p(X).
```

According to the syntax of [8], the body of the third (resp. the fourth) rule in our Example 3 is true whenever only one occurrence of `q(X)` (resp. `p(X)`) is true. Now {p(a), p(b), q(b)} is a minimal model for this program, and also a fixpoint for its immediate consequence operator—and the same is true for {q(a), p(b), q(b)}. Thus there is neither a unique minimal fixpoint nor a least model. Therefore, the formal semantics of logic programs no longer holds when aggregates are used in non-stratified programs (see Section 3.4).[3] □

[3] This example illustrates that these problems are caused by the non-monontonic nature of `count`. In fact, the following program where `count` is replaced by our running-FS operator has as unique minimal model {p(a), q(a), p(b), q(b)}:

```
p(b).    q(b).
p(a) ← 1:[q(X)].
q(a) ← 1:[p(X)].
```

3 Datalog^{FS} syntax and semantics

Datalog^{FS} extends Datalog by adding the following three constructs: (1) Running-FS goals, (2) multioccurring predicates, and (3) Final-FS goals. In this section, we first define the syntax of Datalog^{FS} programs and then the semantics of all three constructs by their rewriting into stratified logic programs with arithmetics and comparisons operators.

3.1 Datalog^{FS} syntax

We define Datalog^{FS} as an extension of Datalog without arithmetics and comparison operators (at the end of Sect. 5, we will introduce a version with these operators).

As shown in Sect. 2, a Datalog program is a set of rules. Similarly, a Datalog^{FS} program also consists of a set of rules where the head literal can either be an atom or an *FS-assert* statement. An FS-assert statement describes multi-occurring predicates and is an atom followed by `:K`, where `K` is either a constant or a variable. The body literals of Datalog^{FS} can be either atoms or negated atoms as in Datalog, or they can be *FS goals*. Now FS goals come in the following two forms:

– `Kj:[exprj]` for a *Running-FS goal,* and
– `Kj =![exprj]` for a *Final-FS goal.*

where `exprj` is a conjunction of positive atoms, and `Kj` can either be constant or a variable not contained in `exprj`.

We will next define the semantics of Running-FS goals and multi-occurring predicates by their rewriting into logic programs. Final-FS constructs are discussed in Sect. 3.4, where we also define the notion of stratified programs for this non-monotonic construct and negation.

3.2 Semantics of running-FS constructs

In Sect. 1, we have shown an example of rewriting of `twosons` and `sixsons` rules. These rewritings are correct, but assume that the integer in the running-FS term is a constant. We next specify a rewriting that does not depend on this assumption.

The formal semantics of a running-FS goal is defined by means of its rewriting into positive logic programs, with the help of lists and arithmetic sum operator. However, this list-based formulation will not be used in the actual implementation, which uses the more direct and efficient approach described in Sect. 6.

Now, each running-FS goal will be rewritten separately. Thus, let the jth such goal be

```
Kj:[exprj(Xj,Yj)]
```

where `Xj` and `Yj` are two vectors, denoting respectively the global variables and the local ones. Therefore, our rewriting replaces `Kj : [exprj(Xj, Yj)]` in the rule by `conj(Kj, Xj, _)`, where `conj` is defined as follows:[4]

```
conj(1, Xj, [Yj]) ←      exprj(Xj, Yj).
conj(N1, Xj, [Yj|T]) ← exprj(Xj, Yj), conj(N, Xj, T),
                        notin(Yj, T), N1 = N + 1.
```

where `notin` is defined without using negation, as follows:

```
notin(Z, [ ]).
notin(Z, [V|T]) ← Z ≠ V, notin(Z, T).
```

For a real-life example, let us consider the often used guideline that an assistant professor to be advanced to associate professor should have an H-index [17] of 13 or higher. This can be expressed in Datalog^{FS} as shown in Example 4, below.

Example 4 Our candidate must have authored at least 13 papers each of which has been referenced at least 13 times. The database table `author(Author, Pno)` lists all papers (co-)authored by a person, while the atom `refer(PnFrom, PnTo)` denotes that paper `PnFrom` contains a reference to paper `PnTo`.

```
atleast13(PnTo) ←  13:[refer(PnFrom, PnTo)].
hindex13(Author) ← 13:[author(Author, Pno),
                        atleast13(Pno)].
```

the first rule finds all papers that have been cited at least 13 times while the second rule finds all authors that have authored at least 13 of those papers. □

Following the rewriting of running-FS goals, the second rule in Example 4 becomes:

```
hindex13(Author) ← con2(13, Author, _).
```

where `con2` is defined as follows:

```
con2(1, Author, [Pno]) ←      author(Author, Pno),
                               atleast13(Pno).
con2(N1, Author, [Pno|T]) ← author(Author, Pno),
                               atleast13(Pno),
                               con2(N, Author, T),
                               notin(Pno, T),
                               N1 = N + 1.
```

Two similar rules define `con1` that re-expresses the first rule of Example 4. Although different pairs of rules are needed for each running-FS term, all these rules can use the same `notin` predicate, since its definition is generic.

In Example 4, we only have one local variable and one global variable. However, generalizing to the case where we have an arbitrary number of global variables and local variables is straightforward, since we can arrange them into lists of local variables and global variables (as shown above by `conj` rule in the generic rewriting). In Example 5, we show a rewriting with two local variables and one global variable.

Example 5 Each person that has at least 10 grandsons can be defined as follows:

```
grandson10(X) ← person(X), 10:[son(X, Y), son(Y, Z)].
```

Its rewriting is defined by using lists as shown below:

```
grandson10(X) ←         person(X), con3(10, X, _).
con3(1, X, [[Y, Z]]) ←  son(X, Y), son(Y, Z).
con3(N1, X, [[Y, Z]|T]) ← son(X, Y), son(Y, Z),
                          con3(N, X, T),
                          notin([Y, Z], T), N1 = N+1.
```

□

Another example of this rewriting is given in Appendix A. The definition of `notin` uses goals such as $Z \neq V$ denoting inequality between single variables. In general, we might have several local variables which can be organized in lists (or other complex objects). Then the inequality condition between two lists can simply express the fact that not all the corresponding elements in the list are equal.

This rewriting of Datalog^{FS} into a pure logic program proves that our running-FS construct is monotonic and can thus be freely used in recursion while preserving the formal (model-theoretic, proof-theoretic and least fixpoint) semantics of logic programs. Also observe that this definition does not assume any total order among the elements being counted: in other words, it satisfies the important principle of query genericity [18].

However, this list-based formulation of the running-FS construct is too inefficient to be used in the actual implementation. Later in Sect. 6, we will propose an implementation approach that takes us directly from Datalog^{FS} into an efficient implementation which is not based on the use of lists. Likewise, we expect that Datalog^{FS} programmers will use these running-FS constructs without ever thinking about lists and the abstract semantics discussed in this section.

3.3 Multi-occurring predicates

In all the examples considered so far, base predicates and derived predicates were always counted as providing support of one. But we will now introduce the concept of multi-occurring predicates, that is, predicates that provide a support greater than one. To explain this concept, suppose we want to keep track of the number of citations received by papers, using the base predicate `ref`. For instance, `ref("MousaviZ11")` indicates that the paper with DBLP

[4] Observe that while `N+1` can be viewed as a call to an arithmetic functions, we can stay in the framework of pure logic programs and view it as the postfix functor `+1` applied to `N`, as in Datalog$_{1S}$ [16]; this also supports comparison between positive integers without assuming a totally ordered universe.

identifier "MousaviZ11" has received one citation, and we say that `ref("MousaviZ11")` has support one. Now, if the paper has, say, six citations we could consider storing in our fact base six occurrences of `ref("MousaviZ11")`, but that would be of no logical consequence since idempotence is assumed in the semantics of logic programs. Therefore, to account for these six identical facts, we will instead add one more attribute to the predicate `ref` and assume that our database contains the facts: `ref("MousaviZ11", 1), …, ref("MousaviZ11",6)`. Each of these six facts contributes with support one, and thus we will say that `ref("MousaviZ11")` has support 6 and represent this information by the following notation:

```
ref("MousaviZ11"): 6.
```

This denotes that the paper with DBLP identifier "MousaviZ11" is currently cited in six papers, whereby `Pno="MousaviZ11"` now contributes with a count of six to the b-expression of the rule in Example 6, below:

Example 6 Total reference count for an author.

$\mathtt{tref(Authr): N \leftarrow N:[author(Authr, Pno), ref(Pno)].}$

where the occurrences of `tref(Authr)` are the sum of occurrences of all papers by `Authr`. □

The terms `":6"` and `":N"`, respectively, used in the `ref` fact above, and the head of the rule of Example 6 will be called *FS-assert* terms. The FS-assert construct provides a very useful extension, since there are numerous examples, such as those presented in [19], where it is desirable to count certain predicates as providing a support level greater than one.[5]

The formal semantics of programs P with frequency assert terms is defined by *expanding* it into its $\bar{P}$ equivalent, which is obtained as follows:

1. Each rule in P with head $\mathtt{q(X1, \ldots, Xn): K}$ and with body `Body` is replaced by

   $\mathtt{\bar{q}(X1, \ldots, Xn, J) \leftarrow lessthan(J, K), Body.}$

   where `J` cannot occur in `Body` and `lessthan(J, K)` is a distinguished predicate used to generate all positive integers up to `K`, included, since:

   $\mathtt{lessthan(1, K) \leftarrow\ K \geq 1.}$
   $\mathtt{lessthan(J1, K) \leftarrow lessthan(J, K), K > J, J1 = J + 1.}$

2. Each occurrence of `q(X1, …, Xn)` in the body of rules of the program so obtained is replaced by: $\mathtt{\bar{q}(X1, \ldots, Xn, J)}$ where `J` is required not to appear elsewhere.

Thus, the meaning of our program,

$\mathtt{ref("MousaviZ11"):6.}$
$\mathtt{tref(Authr):N \leftarrow \quad N:[author(Authr, Pno), ref(Pno)].}$

is defined by its expansion into the following program:

$\mathtt{\overline{ref}("MousaviZ11", J) \leftarrow lessthan(J, 6).}$
$\mathtt{\overline{tref}(Authr, J) \leftarrow \quad lessthan(J, N),}$
$\mathtt{\quad N:[author(Authr, Pno), \overline{ref}(Pno, J1)].}$

where `lessthan` is defined as shown in point 1, above. Also observe here that, in this special case where multiplicity is generated by a running-FS, the rewriting can be simplified into:

$\mathtt{\overline{tref}(Authr, N) \leftarrow N:[author(Authr, Pno), \overline{ref}(Pno, J1)].}$

This does only hold if the running-FS term is not constant. Moreover, as we will show in Sect. 3.4, the same consideration does not hold if a final-FS goal is present in the body of the rule because its semantics does not imply the `lessthan` predicate which thus cannot be removed.

As a result of the presented expansion, we have that, in Example 6, `ref("MousaviZ11"): 6` contributes with six to the reference count of each author of that paper.

An important property of frequency statements is that, when multiple statements hold for the same fact (base fact or derived fact) only the largest value is significant, the others are subsumed and ignored. Thus, if the following two predicates are derived,

```
ref("MousaviZ11"):6.
ref("MousaviZ11"): 4.
```

the second fact carries no additional information and can be simply dropped. This property is useful in many applications that require maxima, as discussed in Sect. 9. However, if instead of the maximum frequency, we would like to take the sum of frequencies, we can simply add an additional argument. Indeed, the following two facts that also show the source of the citations imply a total count of 10:

```
ref("MousaviZ11", journals): 6.
ref("MousaviZ11", others): 4.
```

3.4 Final-FS goals

The rewriting rules defined in the previous section make it possible to use variables rather than constants in the specification of FS goals. This is useful in many situations. For instance, to find the actual number of sons that a person has, we can use the Datalog^{FS} program in Example 7, below.

[5] This discussion suggests that the formal semantics of Datalog^{FS} can also be defined without using lists–that is, in terms of Herbrand bases and interpretations that only use the constants and predicates in the programs and no function symbols. This interesting topic is left for future research.

Example 7 How many sons does a person have?

$$\texttt{csons(PName, N)} \leftarrow \texttt{person(PName)}, \texttt{N =![son(PName, Sname)]}.$$

where the goal `N=![son(PName, Sname )]` allows to retrieve the exact number `N` of distinct assignments to variable `Sname` that make `son(PName, Sname)` true. □

Now the semantics of Example 7 above is defined by using running-FS goals, as follows:

```
csons(PName, N) ←      person(PName),
                       N:[son(PName, Sname)],
                       ¬morethan(PName, N).
morethan(PName, N) ← N1:[son(PName, _)], N1 > N.
```

while the final-FS construct does not increase expressive power, it allows us to express queries in a more compact and intuitive form and is also useful for compilation, as we will see later. In general, a Final-FS goal has the form,

`Kj =![exprj(Xj, Yj)]`

where `exprj(Xj, Yj)` is a b-expression, with the same syntax as that used for Running-FS goals. Thus, `Xj` and `Yj` respectively denotes its global variables and the local variables. The formal semantics of this Final-FS goal is defined by rewriting it into the conjunction:

$\texttt{Kj : [exprj(Xj, Yj)],¬morethan}(\texttt{X}_\texttt{j}, \texttt{K})$

where:

$\texttt{morethan}(\texttt{X}_\texttt{j}, \texttt{K}) \leftarrow \texttt{K1 : [exprj}(\texttt{X}_\texttt{j}, \_)], \texttt{K1} > \texttt{K}.$

Obviously, the rewriting in logic programming is obtained by substituting the Running-FS goal with its rewriting. Moreover, the rewritten program must be stratified with respect to negation. Thus we have the following definition:

Definition 1 A Datalog^{FS} program is stratified if its rewriting into a logic program is negation-stratified.

The question of whether a Datalog^{FS} program P is stratified can be answered by simply inspecting the given program and ensuring the following condition holds. Basically each rule of P can contain two kinds of goals:

(a) the normal goals of standard Datalog programs and
(b) Running-FS and Final-FS goals, `K:[b-expression]`, or `K=![b-expression]`, where `b-expression` is the conjunct of one or more predicates.

For (a) the stratification conditions have not changed: the head of the rule must belong to a stratum that is not lower than that of every goal, and strictly higher than that of every negated goal.

For (b) the head of the rule must belong to a stratum that is (i) not lower than that of every predicate in the b-expression of `K : [b − expression]` and (ii) strictly higher than that of every predicate in the b-expression of `K=![b-expression]`.

4 Stratified Datalog^{FS} by example

In the previous sections, we have illustrated the semantics of Datalog^{FS} with the help of simple examples. We will now introduce more complex examples to illustrate the many applications of stratified Datalog^{FS} using FS goals in recursive rules, which make use of positive integers but do not use function symbols or arithmetic.

Consider Example 8, below, that is based on an example by Ross and Sagiv [8].

Example 8 Some people will come to the party for sure. Others will also come once they learn that three or more of their friends will come.

```
willcome(X) ← sure(X).
willcome(Y) ← 3:[friend(Y, X), willcome(X)].
```

where `sure` denotes people who will come even if none of their friends will. □

We extend Example 8 considering that one person might be more timid than another, and different people could require a different number of friends before they also join the party. This scenario is described by Example 9, below.

Example 9 A person will join the party if a sufficient number of friends join.

```
join(X) ← sure(X).
join(Y) ← requires(Y, K), K:[friend(Y, X), join(X)].
```

where `requires(person, number)` denotes the number of friends required by a person, and `number` must be a positive integer. □

The reachability problem in directed hypergraphs calls for a strategy, similar that used in Example 9, inasmuch as several nodes must be reached before we can reach the next node.

A directed hypergraph H is a pair $H = (X, E)$ where X denotes the nodes, and E denotes the hyperedges. Each hyperedge is a pair $\langle L, \mathrm{v_L}\rangle$, where L is a non-empty set of nodes (the source nodes) and v_L is a node not belonging to L (the sink node). Hypergraph reachability is modeled by the following recursive definition: (i) a node `X` is always reachable from itself and (ii) a node `Y` is reachable from `X` if there is a hyperedge such that its sink is `Y` and each of its sources is reachable from `X`. Thus the program presented in Example 10 is the reachability query for hypergraphs.

Example 10 Let the nodes of a hypergraph be represented by the monadic predicate node, while its edges are represented with two dyadic predicates source and sink. Thus, the edge $ID = \langle L, \mathtt{v_L} \rangle$ is represented by the fact sink(ID, v$_L$), and a fact source(ID,X) for each X ∈ L. Then the reachability query for hypergraphs is:

```
reach(X, X) ← node(X).
reach(X, Y) ← node(X), K =![source(ID, _)],
              K:[source(ID, Z), reach(X, Z)],
              sink(ID, Y).
```

the first rule states that each node X is reachable from itself. The second rule says that a node Y can be reached from X if there is a hyperedge ID whose sink is Y and each source Z of ID is reachable from X (imposed by the conjunction of goals K=![source(ID, _)], K:[source(ID, Z),reach(X,Z)]). Thus, when all the source nodes of a given edge ID have been reached, the sink node of ID is reached as well. □

Let us now introduce some examples using multi-occurring predicates. If staff(Lab, Name) describes the current staff in each laboratory, the following rule can be used to determine the total number of references pointing to (papers by researchers in) each laboratory.

```
labref(Lab): Tot ← Tot:[staff(Lab, Name),
                        tref(Name)].
```

We next show some examples that use multi-occurring predicates in recursive rules. Bill-of-materials (BOM) applications represent a well-known example of the need for recursive queries. For instance, our database might contain records such as assbl(Part, Subpart, Qty) which, for each part number, give the immediate subparts used in its assembly and the quantity in which they are used. For instance, a bicycle has one frame and two wheels as immediate subparts. At the bottom of the Bill-of-material DAG, we find the basic parts that are purchased from external suppliers and provide the raw materials for our assembly. Basic parts are described by basic(Pno, Days) denoting the days needed to obtain that basic part. Several interesting BOM applications are naturally expressed by combining aggregates and recursion, as the one in Example 11.

Example 11 How many basic parts does an assembled part contain?

```
cassb(Part, Sub): Qty ← assbl(Part, Sub, Qty).
cbasic(Pno): 1 ←        basic(Pno, _).
cbasic(Part): K ←       K:[cassb(Part, Sub), cbasic(Sub)].
cntbasic(Prt, C) ←      C =![cbasic(Prt)].
```

For each assembled part, we count the number of basic parts as a recursive sum, using goal K:[cassb(Part, Sub), cbasic(Sub)]. Basic parts have occurrence 1, as stated by the second rule. The multi-occurring predicates cassb and cbasic in the running-FS goal allow to sum, in a recursive manner, the occurrences of each subpart of an assembled part. □

Of course, the total count of basic parts used by a part (which is derived using the last rule in Example 11), such as wheel, should not be retrieved using a goal such as N: [cbasic(wheel)] since this returns all the positive integers up to the max N value for which the running-FS goal holds. A goal such as N=! [cbasic(wheel)] should be used instead inasmuch as this returns the exact count of the basic subparts for wheel. Similar observations hold also for the third rule in Example 12.

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 delivery a bicycle to the customer can be computed as the maximum of the number of days that the various basic parts require to arrive. Thus we have Example 12, below.

Example 12 How many days until delivery?

```
delivery(Pno): Days   ←   basic(Pno, Days).
delivery(Part): Days  ←   assbl(Part, Sub, _),
                          Days:[delivery(Sub)].
actualDays(Part, CDays) ← CDays =![delivery(Part)].
```

For each assembled part, we find each basic subpart along with the number of days this takes to arrive. By using the multi-occuring predicate delivery inside the FS-goal Days:[delivery(Sub)] we find, for a given Part, the maximum among the delivery times of its subparts. □

Another interesting application of Datalog^{FS} is the one proposed by Mumick, Pirahesh and Ramakrishnan [19] and presented in Example 13, below.

Example 13 Companies can purchase shares of other companies; in addition to its directly owned shares, a company A controls the shares controlled by a company B when A has a controlling majority (50 %) of B's shares (in other words, when A bought B). The shares of each company are subdivided into 100 equal-size lots.

```
cshares(C2, C3, dirct): P ←    owned_shares(C2, C3, P).
cshares(C1, C3, indirct): P ← P:[bought(C1, C2),
                                  cshares(C2, C3, _)].
bought(C1, C2) ←  C1 ≠ C2,  50:[cshares(C1, C2, _)].
```

where dirct and indirct are constants. □

These examples show that stratified Datalog^{FS} allows us to express various useful applications that were used in the past to illustrate the limitations of negation-stratified Datalog. In the next section, we present a formal investigation of the expressivity properties of stratified Datalog^{FS}.

5 Expressive power of Datalog^{FS}

Our previous examples illustrate that Datalog^{FS} is a very powerful language. In fact, we will show that (i) stratified Datalog^{FS} is more expressive than Datalog with stratified negation and aggregates and (ii) many queries for which only inefficient formulations are possible with Datalog can now be expressed and supported by optimal algorithms in stratified Datalog^{FS}. In this section, we focus on (i) and analyze the impact upon expressive power of the new constructs. We will return to (ii) in next sections, where we study implementation and optimization issues.

5.1 Comparison with negation-stratified datalog

In this section, we will compare stratified Datalog^{FS} with negation-stratified Datalog.

We recall that in negation-stratified Datalog, arithmetic expressions and aggregates are not allowed and ordering of the domain is not given a priori (thus, there are no comparison operators). We have the following theorem:

Theorem 1 *Datalog*$^{\neg s} \subsetneq$ *stratified Datalog*^{FS}

Proof The containment of *Datalog*$^{\neg s}$ by stratified Datalog^{FS} is trivial because stratified Datalog^{FS} is an extension of *Datalog*$^{\neg s}$. The fact that stratified Datalog^{FS} is strictly more expressive than *Datalog*$^{\neg s}$ follows from Kolaitis' result [20] that the game query cannot be expressed in *Datalog*$^{\neg s}$, whereas Example 14 shows the game query expressed in stratified Datalog^{FS}. □

The Game Query. A two player game can be described by a graph $G = (V, V_0, V_1, E)$, with a partitioning $V = V_0 \cup V_1$ of nodes into positions where player 0 moves and position where player 1 moves. The possible moves are described by the edge relation $E \subseteq V \times V$.

A game is strictly alternating if every move of player 0 is followed by a move by player 1, and vice versa, that is $E \subseteq V_0 \times V_1 \cup V_1 \times V_0$. Consider now player 0; by the following recursive definition, we can establish whether, in a strictly alternating game, $v \in V_0$ is a winning position for player 0:

- if player 1 cannot move, that is $\nexists u \in V_1 : (v, u) \in E$, or
- for each move of player 1, there is a move of player 0, which leads to a winning state, that is $\forall u \in V_1$ such that $(v, u) \in E$, there exists a position $h \in V_0$ such that $(u, h) \in E$ and h is winning.

A game query defines the winning position of player 0 in a strictly alternating game. Extending a result by Dalhaus [21], Kolaitis proved the following result [20]:

Theorem 2 *(Dahlhaus and Kolaitis) Datalog*$^{\neg s}$ *cannot express the game query.*

The Example 14 express the game query in stratified Datalog^{FS}.

Example 14 Let `e(x,y)` be the facts that represent E, and let $\mathtt{v}_{0(\mathtt{x})}$ be the facts that represent V_0, the game query in stratified Datalog^{FS} is the following:

$$\mathtt{w(X)} \leftarrow \mathtt{v_0(X)}, \neg\mathtt{e(X,\_)}.$$
$$\mathtt{w(X)} \leftarrow \mathtt{v_0(X)}, \mathtt{e(X,Y)}, \mathtt{K} =![\mathtt{e(Y,Z)}], \mathtt{K}{:}[\mathtt{e(Y,Z)}, \mathtt{w(Z)}].$$

The first rule derives each state where player 1 cannot move. The second recursive rule derives the state where, for each move of player 1, there is a move of player 0 that leads to a winning state. In the second rule, $\mathtt{K} =![\mathtt{e(Y,Z)}]$ counts the number of `Z` positions and $\mathtt{K}{:}[\mathtt{e(Y,Z)}, \mathtt{w(Z)}]$ verifies that each `Z` position is a winning state. □

5.2 Comparison with aggregate-stratified datalog

Our FS goals are monotonic, but this is not true in general for all types of aggregates. In order to deal with non-monotonic aggregates, the concept of stratified aggregates has been introduced. In [22], Mumick and Shmueli describe D^a, that is, Datalog with stratified aggregates as an extension of Datalog that allows basic arithmetic functions $(+, -, *, \ldots)$ as built-in and "Groupby" predicates of the form

$$\mathtt{GROUPBY}(\mathtt{r}(\bar{\mathtt{t}}), [\mathtt{Y}_1, \mathtt{Y}_2, \ldots, \mathtt{Y}_m],$$
$$\mathtt{Z}_1 = \mathtt{A}_1(\mathtt{E}_1(\bar{\mathtt{t}})), \ldots, \mathtt{Z}_n = \mathtt{A}_n(\mathtt{E}_n(\bar{\mathtt{t}})))$$

to appear as subgoals in Datalog body rules. The "Groupby" predicate takes as arguments: a predicate `r` with its attribute list $\bar{\mathtt{t}}$, a grouping list $[\mathtt{Y}_1, \mathtt{Y}_{2,\ldots}, \mathtt{Y}_m]$ contained in $\bar{\mathtt{t}}$ and a set of aggregation terms $\mathtt{Z}_1 = \mathtt{A}_1(\mathtt{E}_1), \ldots, \mathtt{Z}_n = \mathtt{A}_n(\mathtt{E}_n)$. For each aggregation term $\mathtt{Z}_i = \mathtt{A}_i(\mathtt{E}_i(\bar{\mathtt{t}}))$, $\mathtt{Z}_i$ is a new variable, $\mathtt{E}_i(\bar{\mathtt{t}})$ is an arithmetic expression that uses the variables $\bar{\mathtt{t}}$ and $\mathtt{A}_i$ is an aggregate operator, for example `sum, count, max, min` and `avg`. Stratification means that if a derived relation $r1$ is defined by applying aggregation on a derived relation $r2$, then $r2$'s definition does not depend, syntactically, on relation $r1$.

The expressive power of D^a was studied in [22]: when no function symbols or arithmetic are present, then the addition of stratified COUNT adds more power to the language than the addition of stratified negation (the latter can be expressed using the former but not vice versa). However, there remain many queries that are not expressible in D^a: in particular the Summarized Part Explosion query, which counts the number of copies of component `Sub` needed to construct one copy of `Part`, cannot be expressed in D^a [22]. Now, this query can be easily expressed in stratified Datalog^{FS} as shown in Example 15.

Example 15 Summarized Part Explosion

```
cassb(Part, Sub): Qty ← subpart(Part, Sub, Qty).
need(Sub, Sub): 1 ←     subpart(_, Sub, _).
need(Part, Sub): K ←   K:[cassb(Part, P1), need(P1, Sub)].
total(Part, Sub, K) ←   K =![need(Part, Sub)].
```

□

Therefore, we have the following theorem:

Theorem 3 *Stratified Datalog*$^{FS} \nsubseteq D^a$

In the Appendix C, we prove that, by adding arithmetic and comparison operators to stratified Datalog^{FS} it can express all queries expressible in D^a. Thus, it contains D^a.

5.3 Expressive power on ordered domains

We now compare stratified Datalog^{FS} with $D^{\neg +}$ which is Datalog$^{\neg s}$ with the addition of the built-in operator +. In [22], it is proved that $D^{\neg +}$ can express *all computable functions* on ordered domains. We recall that the built-in function X+Y=Z used in [22] assumes that there is a binding for X and Y that implies a binding for Z.

Then the following stratified Datalog^{FS} program in Example 16 can be used to express the same function for non-negative integers:

Example 16 Expressing integer sum.

```
adding(X, Y, 0) : K ← K = X.
adding(X, Y, 1) : K ← K = Y.
sum(0, 0, 0).
sum(X, Y, Z) ←      Z =![adding(X, Y, _)].
```

The first two rules derive, for the two addends X and Y, a multi-occurring predicate having the same occurrence as the value of X and Y respectively. The final-FS goal in the last rule finds the exact sum of the two addends and returns its value. □

It is important to observe that the final-FS construct belongs to a stratum that is lower than sum, which can therefore be used in recursive rules (and the program in Example 16 can be executed in a top-down fashion to avoid the enumeration of all possible derivations). Thus, we have the following result:

Theorem 4 $D^{\neg +} \subseteq$ *stratified Datalog*^{FS}.

Now, since $D^{\neg +}$ can express all computable queries on ordered domains so does stratified Datalog^{FS}.

5.4 Final considerations

The results in the previous sections show that the additional expressive power gained by stratified Datalog^{FS} is due to the running-FS construct. In fact, in Appendix B, we give a formal proof that is sufficient to use such construct, without arithmetic or comparator operators, to express the Final-FS and FS-assert.

While the FS-assert term does not contribute to the expressive power of stratified Datalog^{FS}, it plays a very useful practical role since it (i) simplifies the writing of programs and (ii) also simplifies their compilation and implementation, as discussed in detail in the next section. In the same vein, while the final-FS construct can also be expressed using stratified negation on running-FS goals, we will disallow this combination in our actual FS-programs and instead insist on the direct usage of Final-FS constructs (in programs stratified with respect to this construct) for the very reasons (i) and (ii) outlined above.

We actually conjecture that, besides the two examples given here for which we have formal proofs, there are many other interesting queries that can be expressed in stratified Datalog^{FS} but cannot be expressed in $Datalog^{\neg s}$ nor D^a. For instance, a query similar to the Summarized Part Explosion query in Example 15 is one that counts the number of paths between each pair of node in an acyclic graph, presented in Example 17.

Example 17 Consider an acyclic graph represented by the facts edge(x, y) that represent the edges of the graph. To count the number of paths between every node-pair, we write:

```
path(X, Y) : 1 ←          edge(X, Y).
path(X, Y) : K ←          K:[edge(X, Z), path(Z, Y)].
countpaths(X, Y, K) ← K =![path(X, Y)].
```

The first rule assigns occurrence 1 to each direct edge. The second rule assigns, to each pair of nodes X and Y, the number of paths between them by summing, for each node Z, that is reached from X in one edge, the number of paths between Z and Y. Finally, the last rule returns, for each pair of nodes, only the maximum found occurrence. □

Also the reachability query on directed hypergraphs of Example 10 is very difficult to formulate in $Datalog^{\neg s}$, because of the recursive pattern for universal quantification which is similar to that of the game query in Example 14. In summary, it appears that the expressive power of stratified Datalog^{FS} is significantly above that of $Datalog^{\neg s}$ and D^a, whereby it is now meaningful to ask what is the upper bound for stratified Datalog^{FS} power. If we assume that our domain is totally ordered (a reasonable assumption in most real-life situations), then the result is given by Theorem 4 that implies that all computable queries on ordered domain can be expressed in stratified Datalog^{FS}.

This important result suggests that stratified Datalog^{FS} has reached the ultimate level of expressive power, where its usefulness no longer depends on its ability of expressing the desired queries, instead it depends on the performance with which those queries are implemented. Thus, in the next section, we turn our focus on performance issues. Indeed, one of the historical limitations of Datalog is that, while many queries can be expressed by programs using stratified aggregates or negation, these programs cannot be compiled into implementations that achieve the optimal complexity that is delivered by procedural programs. The compilation and optimization techniques discussed in the next section go a long way toward solving these problems.

In as much as we will now focus on practical issues, the ability of stratified Datalog^{FS} to express integer arithmetic is only of theoretical interest, insofar as we need builtins that allow us to implement these functions very efficiently. Thus, from now on, we assume that these are part of our language and discuss the powerful queries and compilation issues they bring about.

For instance, in the examples presented in Sect. 3, the frequency assigned to the head has the same value as the running-FS term in the body, but this need not always be the case. For instance, if we estimate that, on the average, every paper authored by `A` in a particular laboratory is co-authored by another person in the same lab, then a better estimate for the number of papers published by a laboratory is `Tot ÷ 2`. Thus a better estimate of a pseudo H-factor for each laboratory could be the one expressed by the following program:

```
labref(Lab): H ← Tot:[staff(Lab, A), tref(A)],
                 H = Tot ÷ 2.
```

6 Compilation and optimization

The significant gain in expressive power realized by stratified Datalog^{FS} can have a significant practical impact only if combined with a very efficient implementation. Because of its declarative semantics, alternative implementation approaches are possible, including the top-down methods of Prolog; however, in this paper, we focus on bottom-up execution and leave top-down methods for later papers (including those that emulate top-down methods, such as magic-sets that are briefly discussed in [23]).

The efficient bottom-up computation of Datalog^{FS} programs is achieved through:

- Differential fixpoint optimization—that is, the extension of the seminaive fixpoint optimization to recursive Datalog^{FS} rules.
- Max-based optimization. This is a new technique introduced for Datalog^{FS} whereby the computation over the successive integers produced by running-FS constructs is avoided and replaced by a computation on their upper bounds.

6.1 Differential fixpoint

The differential fixpoint (a.k.a., the seminaive fixpoint) method, which represents the cornerstone of the bottom-up implementation for Datalog programs [24], is also applicable and effective for Datalog^{FS} programs. However, some nontrivial modification are required as discussed next.

The seminaive computation involves the rewriting of the recursive rule bodies to avoid redundant computation. The standard rewriting produces a new delta rule for each recursive goal in the rule. Now, a recursive Datalog^{FS} rule defining predicate p could contain a goal such as $100:[p(X,Y)]$. Then a naive implementation of the differential fixpoint could start by rewriting this goal into a conjunct of 100 similar goals and then differentiate that into 100 rules. Fortunately, one single differential rule is all that is needed, as it is shown next.

Toward that, let us start with the differentiation of $2:[p(X,Y)]$, where X and Y respectively denote the global variables and the local variables (or vectors thereof) in our b-expression. Now since

$$2:[p(X,Y)] = p(X,Y_1),\ p(X,Y_2),\ Y_1 \neq Y_2$$

its differentiation yields:

$$\delta p(X,Y_1),\ p(X,Y_2),\ Y_1 \neq Y_2 \cup p(X,Y_1),\ \delta p(X,Y_2),\ Y_1 \neq Y_2$$

Thus we are here taking the union of two expressions that are identical except for the renaming of the local variables whose names are immaterial. In other words, either conjunct can be dropped or our symbolic differentiation can be simplified into:

$$\delta p(X,Y_1),\ \ p(X,Y_2),\ \ Y_1 \neq Y_2$$

The generalization of this differentiation rule to the case where $K > 2$ can now be proven by induction. Thus in the following, we will prove that in order to differentiate an FS-goal, it is sufficient to differentiate only once the b-expression.

Consider the differentiation of $\big(K+1:[expr(X,Y)]\big)$. We will now use the following rewriting of this FS-expression, which we call a *strict factorization*:

$$\begin{aligned}&K+1:[expr(X,Y)] = expr(X,Y),\\&\quad K:[expr(X,Y_1),\, Y_1 \neq Y]\end{aligned}$$

Now, the symbolic differentiation of the two sides of equality (where the right-hand side is a quadratic expression) yields:

$$\begin{aligned}&\delta\big(K+1:[expr(X,Y)]\big)\\&\quad = \delta\big(expr(X,Y)\big),\, K:[expr(X,Y_1),\, Y_1 \neq Y] && (1)\\&\qquad\qquad\qquad \cup\\&\quad expr(X,Y),\, \delta\big(K:[expr(X,Y_1),\, Y_1 \neq Y]\big) && (2)\end{aligned}$$

But, by the inductive assumption, we have that our δ-rewriting is correct up to K, included. Thus,

$$\delta\big(K:[expr(X, Y_1), Y_1 \neq Y]\big) = \delta\big(expr(X, Y_1), Y_1 \neq Y, \\ K-1:[expr(X, Y_2), Y_2 \neq Y, Y_2 \neq Y_1]\big)$$

Therefore (2) becomes:

$$expr(X, Y), \delta\big(expr(X, Y_1)\big), Y_1 \neq Y, \\ K-1:[expr(X, Y_2), Y_2 \neq Y, Y_2 \neq Y_1]$$

We can now merge the first goal into the b-expression, whereby the old condition $Y \neq Y_1$ is subsumed by $Y_2 \neq Y_1$ in the new b-expression, and can thus be eliminated. Therefore, (2) reduces to (3) below:

$$\delta(expr(X, Y_1)), K:[expr(X, Y_2), Y_2 \neq Y_1] \tag{3}$$

Now (3) is identical to (1), modulo variable renaming; thus either (1) or (2) can be omitted.

Thus we obtain the following formula that is valid for $K > 0$ and is called a *strict differentiation* template.

$$\delta\big(K+1:[expr(X, Y)]\big) = \delta(expr(X, Y_1)), \\ K:[expr(X, Y_2), Y_2 \neq Y_1] \tag{4}$$

Relaxed Differentiation and Factorization. The general form of relaxed differentiation is as follows:

$$\delta\big(K+1:[expr(X, Y)]\big) = \delta(expr(X, Y_1)), \\ K+1:[expr(X, Y_2)] \tag{5}$$

For $K > 0$, this formula is equivalent to (4). We have that the first atom, $\delta(expr(X, Y_1))$, is the same for both. As for the second: in (4) we require that $K : [expr(X, Y_2), Y_2 \neq Y_1]$ (K assignments of Y_2 and each has to be different than Y_1, that is we require at least $K+1$ assignments of Y in total) and in (5) we require that $K+1 : [expr(X, Y_2)]$ without the inequality condition, thus we are still imposing at least $K+1$ assignments of Y. This is true because, by applying the differential fixpoint method, we have that $\delta(expr(X, Y_1))$ contains only the facts derived at the last step while $expr(X, Y_1)$ contains the facts derived at all the previous steps (thus also those in $\delta(expr(X, Y_1))$).

Moreover, the formula in (5) is also valid for $K = 0$, thus it holds for every positive integer K.

Furthermore, consider the following *relaxed factorization* equality:

$$K:[expr(X, Y)] = expr(X, Y_1), K:[expr(X, Y_2)] \tag{6}$$

Now, by observing the relationship between (5) and (6), we conclude that the differentiation of a b-expression can be performed as follows: (i) apply the relaxed factorization to the b-expression and (ii) differentiate the goals outside the brackets and leave the b-expression unchanged. This observation allows us to formulate a *canonical differentiation procedure* that consists of the following three steps:

1. **Relaxed Factorization**: all the goals of the b-expressions are rewritten by applying the relaxed factorization transformation described above.
2. **Reduction**: The relaxed factorization process often generates multiple occurrences of equivalent goals (i.e., goals that are identical except for a possible renaming of variables). These repeated equivalent goals are here removed from the rules.
3. **Reduced Differentiation**: The rules produced by the reduction step are differentiated with the b-expression treated as constants (i.e., as if the predicates in the brackets were not recursive or mutually recursive with those for which the delta rules are being generated).

Therefore, after the simple transformations of the first two steps, the third step of the canonical differential process differentiates Datalog^{FS} rules as they were traditional Datalog rules. The simplicity and amenability to efficient implementation of canonical differentiation can be appreciated from a few examples.

Take for instance the recursive rule of Example 12:

```
delivery(Part): Days  ← assbl(Part, Sub, _),
                        Days:[delivery(Sub)].
```

This can be factorized as follows (Sub is a global variable):

```
delivery(Part): Days  ← assbl(Part, Sub, _),
                        delivery(Sub),
                        Days:[delivery(Sub)].
```

No repeated goals have been produced here, thus we can move to the reduced differentiation step that yields:

```
δdelivery(Part): Days  ← assbl(Part, Sub, _),
                         δdelivery(Sub),
                         Days:[delivery(Sub)].
```

For a second example, consider now the following recursive rule (from Example 19 to be discussed later):

```
reach(Y): V ← reach(X), V:[reach(X), arc(X, Y)].
```

There are no local variables in the b-expression of this rule; thus the expansion step produces:

```
reach(Y): V ← reach(X), reach(X), arc(X, Y),
              V:[reach(X), arc(X, Y)].
```

Thus the reduction step yields:

```
reach(Y): V ← reach(X), arc(X, Y),
              V:[reach(X), arc(X, Y)].
```

Now the reduced differentiation step treats the b-expression as a constant whereby we obtain the following delta rules:

δreach(Y): V $\leftarrow$ δreach(X), arc(X, Y),
  V:[reach(X), arc(X, Y)].

For a more complex example, consider the following rule from Example 20:

fpath(X, Z): C $\leftarrow$ node(Y), C1:[fpath(X, Y)],
  C2:[fpath(Y, Z)], C = 1/(1/C1 + 1/C2).

The relaxed factorization yields:

fpath(X, Z): C $\leftarrow$ node(Y), fpath(X, Y), fpath(Y, Z),
  C1:[fpath(X, Y)], C2:[fpath(Y, Z)],
  C = 1/(1/C1 + 1/C2).

No redundant goals can be eliminated here, whereby we move the standard differentiation that yields:

δfpath(X, Z): C $\leftarrow$ node(Y), δfpath(X, Y), fpath(Y, Z),
  C1:[fpath(X, Y)], C2:[fpath(Y, Z)],
  C = 1/(1/C1 + 1/C2).
δfpath(X, Z): C $\leftarrow$ node(Y), fpath(X, Y), δfpath(Y, Z),
  C1:[fpath(X, Y)], C2:[fpath(Y, Z)],
  C = 1/(1/C1 + 1/C2).

Thus, we have now reduced the differential fixpoint optimization for Datalog^{FS} to that of standard Datalog, whereby we can use its well-understood and widely tested optimization techniques. For instance, a further optimization that is supported by many Datalog compilers [25] avoids having to repeat the computation of δfpath(X, Y),δfpath(Y, Z) in both rules. This improvement is thus applicable to Datalog^{FS} as well.

6.2 Max-based optimization

Therefore, as result of the differential fixpoint, our declarative program has been replaced by an operational program that iterates over the operational equivalent of the rules generated in the previous section. Therefore, we can now apply optimizations that preserve the *operational* semantics of those rules. Then, we will find out that the computation needs only to be performed on the max values produced by the running-FS term since all arithmetic and Boolean functions in our examples are monotonic.

The recursive computations specified in Datalog^{FS} programs are of two kinds. The computation of the first kind is specified by a final-FS construct that is applied to the values returned by the running-FS and FS-assert constructs in the underlying rules. Programs in this first kind are those in Examples 7, 11, 12, and 20.

The programs of the second kind do not pass the values generated by the running-FS goal to the head of their rules, but instead they test such values against conditions in the very bodies of the rules containing the running-FS goals. Examples 13, 10, and 14. In order to unify the implementation of the queries in this second group with those of the first group, we will replace equality conditions on values produced by running-FS goals with the $\geq$ condition on the value produced by final-FS goals or constants. Thus, the condition 3 : [friend(X, V1)] in Example 1 will be replaced by K:[friend(X, V1)], K $\geq$ 3, while K:[source(ID, Z)] in Example 10 is replaced by K1 : [source(ID, Z)], K1 $\geq$ K (since here K =![source(ID, _)] is defined w.r.t. a lower stratum and it is thus viewed here as a constant). This implementation approach is obviously correct, inasmuch as it preserves the operational semantics of the rules used in the seminaive fixpoint computation (and the nonmonotonicity of the final-FS construct is not an issue for operational semantics).

Now, the max-based implementation of a Datalog^{FS} program is performed in the following two steps:

Step 1: Compute the max value produced by the running-FS, and evaluate the rules for this max value, Max, and then
Step 2: Repeat the rule evaluation for all positive integers other than Max.

So, let us now discuss the implementation of the constructs FS-assert, running-FS and final-FS in Step 1.

FS-assert: We will store only one fact with the Max value as its new additional argument (rather than one fact for each value between 1 and Max). Therefore, in the differential fixpoint, only atoms with max multiplicity are kept (thus updates in-place will be performed on atoms of smaller multiplicities that were already present).

Running-FS and Final-FS. In Step 1, where we are only interested in the Max values, both constructs are computed as final-FS goals. For regular predicates (i.e., they are not among those defined as multi-occurring via FS-assert terms), we will need to count the occurrences of local variables grouped by the global ones.

Thus, in the first rule of Examples 4, we will count PnFrom for each PnTo—that is, count PnFrom grouped by PnTo in SQL parlance. For instance, in the second rule of Example 4, we have a conjunct where Author is the global (i.e., group-by) variable, and Pno is the local variable whose occurrences will be counted: when the resulting count is $\geq$ 13, then Author is returned to the head. In the recursive rule of Example 8, we must instead count occurrences of (X, Y) grouped by Y.

The SUM aggregate is instead used for b-expressions containing one or more multi-occurring predicates. Take for instance Example 11: both cassb and cbasic are multiple-occurring predicates, Part is the global variable and Sub is the local one in the recursive rule. Moreover, each tuple in cassb has a multiplicity Qty and each tuple in cbasic has multiplicity K. Thus for each Part, we must sum up the products Qty $\times$ K over all Sub values that satisfy the join condition and transfer the resulting sums to the head.[6] Likewise, in the first rule of Example 4, we will inspect `refer` and count the occurrences of the local variable `PnFrom` grouped by the global one `Pno`.[7]

Step 2 and Monotonic Arithmetic. In Step 1, we let the FS-count goals produce their Max values and evaluated our rules on those values. We can now proceed and repeat the evaluation of the rules on all the values between 1 and Max. But in all examples discussed in this paper, the execution of this step will add no new results to those obtained in Step 1. This is obvious in situations, such as those of Examples 7, 8, 9, 10, 13, 14, where the values produced by running-FS goals are simply tested against some monotonic Boolean conditions: these conditions evaluate to true iff they are true for the Max values. The other pattern is that of Examples 11, 12, and 20 where the values generated by the running-FS term are then fed to the FS-final term which disregards all the values but the Max ones. In fact, it is easy to realize that this conclusion holds for situations more complex than those: in fact, Step 2 can be omitted for situations where the values produced by running-FS terms, possibly transformed via monotonic arithmetic functions, are fed to monotonic boolean functions[8] or FS-final goals.

To illustrate this point, let us modify our Example 12 to compute the numbers of hours till the assembled product is assembled. Thus besides the days required for delivery (each day adding 24 h), we will now add 2 h needed to assemble a part after its subparts have arrived:

Example 18 Hours required till `Pno` is assembled.

```
ready(Pno): Hours       ←  basic(Pno, Days),
                            Hours = 24 ∗ Days
ready(Part): Hours      ←  assbl(Part, Sub, _),
                            H:[delivery(Sub)],
                            Hours = H + 2.
ActualHours(Part, CH) ← CH =![ready(Part)].
```

□

Here the `H` value is transferred to the head of the rule by the monotonic arithmetic function `H+2`. Since monotonic functions map maxima to maxima there is no need to consider any value but the maximum value. Therefore, we will require that all functions used to transfer FS values from the body to the head be monotonic.

Discussion. Thus, in addition to being quite expressive, Datalog^{FS} is amenable to very efficient implementation via the two optimization techniques presented in this paper. The first is a generalization of the differential fixpoint, and the second is the Max-based optimization where running-FS goals (the cornerstone of declarative semantics) were evaluated via final-FS constructs (the cornerstone of the operational semantics).

While the differential fixpoint optimization is applicable to every valid Datalog^{FS} program, the FS-based optimization is not: there are two kinds of valid Datalog^{FS} programs to which it is not applicable. The first kind occurs when the values produced by the running-FS goals, rather than being used to test a condition or to determine final-FS values, are used in traditional Datalog rules, or are returned as answers to the query. The second situation is when (i) the arithmetic functions that return the FS values to the final-FS aggregates are not monotonic or (ii) the boolean conditions applied to these values are not monotonic. Implementation techniques for these programs could be devised, but it is far from clear that we should do so. In fact, there are indications that this generalization might not lead to many useful applications, and it is actually counter-productive. Indeed, all the applications presented in this paper, and the many others we have considered, are all conducive to Max-based optimization, and programs that do not have this property are hard to find and they are very contrived. For instance, to select all parts that take 7 days or more to arrive, a user will find quite natural to use conditions such as `K : [delivery(Part)], K ≥ 7`, in the rules of Example 12, or 18. However, to find the parts that will arrive in less than 7 days, the user cannot write `K : [delivery(Part)], K < 7`. Because of the semantics of the running-FS, this query will in fact deliver every part (including the parts returned by the previous query). To find only the parts that are delivered in less than 7 days, the user will have instead to write `K =![delivery(Part)], K < 7`, which is conducive to Max-based optimization. The generalization of Datalog^{FS} discussed in the next section provides yet another reason for which a compiler should disallow Datalog^{FS} programs that are not Max-optimizable.

7 Positive rational numbers

The abstract semantics of Datalog^{FS} is based on counting the repeated occurrences of predicates and then applying integer arithmetic on such counts. However, inferences and computations on positive rational numbers can be easily re-expressed

[6] If this information was stored in SQL tables, then the head of the rule would be updated using the following information:

```
        select cassb.Part, sum(cassb.Qty ∗ cbasic.K)
    from cassb, cbasic where cassb.Sub = cbasic.Sub
                                   group by cbasic.Sub
```

.

[7] Determining local and global variables is already part of the binding passing analysis performed by current Datalog compilers [24].

[8] A boolean function $B(T)$ is monotonic w.r.t. the integer values T whenever $B(T)$ evaluating to true implies that $B(T')$ is true for every $T' > T$.

as computations on integers, and positive numbers containing decimal digits can also be managed in a similar fashion.

For instance, returning to our party example, we might want to say that our level of confidence in John coming to the party is 82 %. Thus we write:

`confidence("John"): 0.82`, and similar confidence rankings can be assigned to every person who could possibly attend the party. Then our program becomes:

```
confidence("John"): 0.82.
%...the rest of the facts....
willcome(X): C ←          C:[confidence(X)].
willcome(Y): 1 ←          3:[friend(X, Y), willcome(X)].
```

Assuming that the confidence rankings are less than one, we will need to have four or more people coming to the party before we cross the threshold of three.

Decimal numbers in rules can be viewed as fractions, where their denominators can all be set to the same large number, that we will call the *scale-up factor*. Also, the integers in these rules can be multiplied by the same scale-up factor and viewed as fractions over the common denominator. With all the denominators set to the scale-up factor, it is now simple to derive Datalog^{FS} rules that operate on the numerators of those scaled-up numbers: these rules are equivalent to the original ones since they produce integer results that divided by the scale-up factor are equal to those produced by computation of the original rules with decimal numbers. For instance, with a scale-up factor of 100, the program equivalent to the original program is:

```
confidence("John"): 82.
%...the rest of the facts....
willcome(X): C ←          C:[confidence(X)].
willcome(Y): 100 ←        300:[friend(X, Y), willcome(X)].
```

This program was obtained from the previous one by multiplying numbers by the scaleup factor of 100. Moreover this is a Datalog^{FS} program that has a least fixpoint solution. Thus the original program also has a least fixpoint, which is computed from this by dividing its integers by the scale-up factor.

Thus by assuming scaling, we can allow and support positive decimal numbers in our programs. This significantly expands the application range of Datalog^{FS}. For instance, say that `arc(a,b):0.66` denotes that a trip started at point `a` will actually take us to point `b` in 66 % of the cases. Then, the following program computes the probability of completing a trip from `a` to `Y` along the maximum-probability path:

Example 19 Maximizing the probability of reaching various nodes of the graph if we start from node `a`.

```
reach(a): 1.00.
reach(Y): V ←   reach(X), V:[reach(X), arc(X, Y)].
maxprob(Y, V) ← V =![reach(Y)].
```

The source `a` is reachable with probability 1. Then, the probability of reaching `Y` via an arc from `X` is the product of the probability of being in `X` times the probability that the segment from `X` to `Y` can be completed. This product is computed with the goal `V:[reach(X),arc(X,Y)]` in the first rule. Since the goal contains two multi-occurring predicates, the product `V` is computed as for integer values and then divided `N-1` times by the scale-up factor, where `N` is the number of multi-occurring predicates in the b-expression (in this example we have 2 multi-occurring predicates thus we divide `V` only once by the scale-up factor). Finally, in the head of the last rule, we only retain the maximum `V`—that is, we only retain the path with largest probability to succeed. □

In terms of implementation, it is clear that the approach and compilation techniques we used to implement Datalog^{FS} when FS values are positive integers remain valid and can now be used when FS values are arbitrary positive numbers (however SUM rather than COUNT will be used in the running-FS terms). This follows directly from the fact that, as we have described in the last section, we avoid expansions in Datalog^{FS} implementation. This also makes the semantics of our programs independent from the scale-up factor used —modulo some round-off issues discussed next.

Consider the program equivalent to that of Example 19 above, obtained using a scale-up factor of 100. This scale-up multiplies by 100 both the `reach` and `arc`, for a combined scaling of 100 × 100. Therefore, we must normalize back the resulting sums by dividing the results by 100.

Thus the equivalent Datalog^{FS} program for a scale-up factor of 100 is:

```
reach(a): 100.
reach(Y): W ← reach(X), V:[reach(X), arc(X, Y)],
                W = V ÷ 100, W ≠ 0.
```

This example also illustrates that an implicit round-off effect is connected with the scale-up factor. For instance if $V = 0.0048$, for a scale-up factor of 100, we obtain $W = 0$, which is rejected because we only allow positive multiplicities, whereas with scale-up factor of 10,000, we obtain $W = 48$. Therefore there is a round-off dependency connected with the round-off factor, and this dependency can be minimized by selecting higher scale-up factors. Thus we will assume that our Datalog^{FS} implementation can use the max scale-up factor that can be efficiently supported by the target hardware (32-bit or 64-bit). With that, round-off issues will not occur in most applications, but in the few situation where they are cause for concern, users can address them by "double precision" declarations and the numeric approximation techniques of scientific computing.

As discussed in the last section, explicit arithmetic expressions are also allowed, provided that they are monotonic in their FS-arguments. In Example 18, only one frequency variable was used in the expression. For a query containing a monotonic expression on two variables, consider the following equivalent formulation for the query in Example 19, that computes the maximum probability of reaching other nodes in the graph, starting from node a:

```
reach(a): 1.00.
reach(Y): V ← reach(X), V1:[reach(X)],
             V2:[arc(X, Y)], V = V1 ∗ V2.
```

In the domain of positive numbers, sum and multiplication are monotonic. However, subtraction and division are antimonotonic w.r.t. their second arguments (i.e., if these become larger, the results become smaller). However, the composition of two antimonotonic functions is monotonic. This provides a simple way to express shortest-path algorithms. Rather than using the arc-distance D, we use its conductance: 1/D. Therefore the celebrated Floyd's algorithm can be expressed as shown in Example 20.

Example 20 Floyd's algorithm:

```
fpath(X, Y): C ←          arc(X, Y, D), C = 1/D.
fpath(X, Z): C ←          node(Y), C1:[fpath(X, Y)],
                          C2:[fpath(Y, Z)],
                          C = 1/(1/C1 + 1/C2).
shortestpath(X, Z, D) ← C =![fpath(X, Z)], D = 1/C
```

where arc(X, Y, D) denotes the weight D of the arc between node X and node Y. Then, the first rule converts these weights into conductances, and the second rule computes the maximum conductance of a path from X to Z. Finally, the third rule converts this value into a weight (which thus corresponds to the minimum weight). □

So, 1/C1 and 1/C2 are antimonotonic, and so is their sum, whose reciprocal is therefore monotonic. Thus the last rule, selecting the highest conductance path, in fact produces the shortest path (and establishes this as a Max-optimizable program). Indeed, (i) paths of lower conductance, and thus all cycles, are automatically discarded during the computation, (ii) the max-based optimization ignores all values, but the max values, and (iii) the differential fixpoint computation produces the two δ-rules shown at the end of Sect. 6.1: in each of the two rules, the max-paths produced in the last step are joined with the max-paths produced in previous steps. Therefore we have that, after the max-based optimization, the differential fixpoint program obtained from Example 20 performs the same computation steps as a procedural version of Floyd's algorithm. Therefore, the compiled version of Example 20 achieves the well-known optimality properties of Floyd's algorithm on all-pairs shortest path computation.

8 Advanced graph applications

Many graph query languages were designed specifically for this application area [26], including some which were shown to have strong connections to Datalog [27–29]. Unlike those languages that were designed for graphs, Datalog^{FS} is designed as a general–purpose extension of Datalog. However, its powerful new constructs entail a concise formulation and efficient support for several graph-oriented applications that were beyond the reach of traditional Datalog. In this section, we discuss web-oriented applications, including social networks and page rank.

8.1 Diffusion models with Datalog^{FS}

The Jackson-Yariv Diffusion Model (JYDM) [30] provides a very powerful abstraction on how social structures influence the spread of a behavior and trends in Social Networks. Typical applications include diffusion of innovations, behaviors, information and political movements. An elegant logic-based formalization of the JYDM model was given in [31], whereby diffusion-related goals and optimization problems can then be formulated as queries on this model. Due to the need to ground probability annotations, however, the approach proposed in [31] can lead to inefficient computations.

We will now show that JYDM can be expressed as compact Datalog^{FS} programs, that are conducive to very efficient implementation. In JYDM, a graph $G = \langle V, E\rangle$ represents the set V of agents and the set E of edges that are the relationships between the agents. Each agent has a default behavior (A) and a new behavior (B). An agent i decides on whether to adopt behavior B depending on:

(i) A constant bc_i which quantifies the extent to which agent i is susceptible to make a change (typically, bc_i measures the reward/cost tradeoffs of making the change).
(ii) The percentage of the neighbors of the agent that took action B (e.g., the percentage of partners of the twitter user who forwarded this tweet).
(iii) A function $g : \{1, \ldots, |V|\} \to [0, 1]$ that modifies criterion (ii) to take into account, in addition to percentage of neighbors who took action B, their number.[9] Observe that g does not directly depend on the node, it only depends on the number of its neighbors; however, g and bc combined provide JYDM with much power and flexibility in characterizing the response w.r.t. both the numbers and percentages of neighbors agents who took action B.

[9] For instance, this function can be used to express the fact that an agent who sees, say, all his 89 partners switching to B experiences a much stronger push than another twitter user who sees his only partner moving to B (although percentage-wise the two situations are the same).

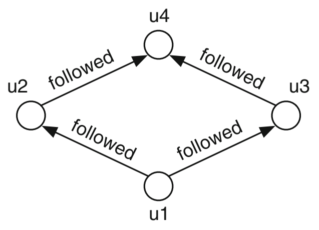


Fig. 1 A diffusion model for twitter

Now, if B_i denotes the social pressure experienced by agent i to adopt behavior B, then the Jackson-Yariv diffusion model is expressed by the following equation:

$$B_i = bc_i * g(\Gamma_i) * \frac{1}{\Gamma_i} * \sum_{(j,i)\in E} B_j, \ \forall i \in V \tag{7}$$

where Γ_i denotes the count of edges with end-node i

When a B_i crosses a threshold, then agent i switches from behavior A to behavior B. Given that bc_i can be scaled up/down to match the application requirements, the threshold can be, and typically is, set to 1.

The JYDM-based diffusion of retweets can easily be expressed in Datalog^{FS}. We represent the twitter network using atom `followed(X,Y)` to indicate that user X is followed by user Y. Moreover, in order to apply the diffusion model, we introduce the following facts:

- `bc(X,K)` means that K is the coefficient of node X.
- `g(N,K)` means that the function `g(N)` returns value K.

The default behavior A means that a user will not retweet, while behavior B means that the user will retweet. Now, the predicate `b(X)` represents the fact that X has retweeted. We assume that there is an agent `source(X)` who first posts the tweet and starts its diffusion.

Thus, Eq. 7 is modeled by the program in Example 21.

Example 21 Equation 7 in Datalog^{FS}.

```
coeff(X, C) ← K2 =![followed(Y, X)], bc(X, V1),
              g(K2, V3), C = V1 * V3/K2.

b(X) ←        source(X).
b(X) ←        coeff(X, C), K=1/C, K:[followed(Y, X), b(Y)].
```

In Eq. 7, the product $bc_i * g(\Gamma_i) * \frac{1}{\Gamma_i}$ is not recursive and it depends only on the features of node i. Therefore, the first rule computes this value for each user. The first goal in this rule determines the number of neighbors of node X; the next three goals in the rule compute the increment C that must be added to X for each of its neighbors who switched to B. Now, the last rule finds each node X for which the condition $K \times C \geq 1$ holds by testing the equivalent condition $K \geq 1/C$. When this is satisfied `b(X)` is set to true. □

The answer to the query `?b(Y)` will list all the users that propagated the tweet that originated from the node specified by `source`, as illustrated by the Example 8.1. Consider the network in Fig. 1 where user u_1 tweets about something. We add the following facts to the program in Example 21 and compute for each user if he/she will, or will not, retweet the post.

```
followed(u1, u2).  bc(u1,1).  g(1, 1.2).  source(u1).
followed(u1, u3).  bc(u2, 0.9).  g(2, 2.3).
followed(u2, u4).  bc(u3,0.5).
followed(u3, u4).  bc(u4,1).
```

From these facts, the program in Example 21 derives the following atoms:

`coeff(u2, 1.08)`, `coeff(u3, 0.6)`, `coeff(u4, 2.3)`, `b(u1)`, `b(u2)`, `b(u4)`.

Thus, first u2 and then u4 will retweet u1's post.

8.2 Markov chains with Datalog^{FS}

A Markov chain is a memory-less stochastic process that is represented by the transition matrix W of $s \times s$ components where the component w_{ij} is the probability to go from the state i to state j in one step. For each node i of the Markov chain we have: $\sum_{j=1}^{s} w_{ij} = 1$. A Markov chain is called *irreducible* if for each pair of nodes i, j, the probabilities to go from node i to node j in one or more steps is greater than zero.

Computing stabilized probabilities of a Markov chain has many real-world applications, such as estimating the distribution of population in a region, and determining the Page Rank of web nodes. Let P be a vector of stabilized probabilities of cardinality s, then for each component p_i of P the following property holds:

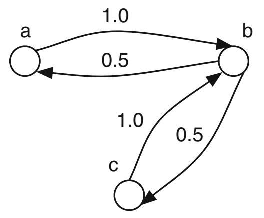
$$p_i = \sum_{j=1}^{s} w_{ji} \cdot p_j \tag{8}$$

This is the equilibrium condition, defines the Markov chain equation, that can be expressed using matrices as follows:

$$P = W \cdot P$$

Computing the fixpoint solution for this Markov chain equation is far from trivial. The simplest approach consists in assigning an initial value (e.g., 1) to all nodes and then iterating until the computation stabilizes. Unfortunately this approach could fail to converge even for irreducible chains that are guaranteed to have a non-trivial equilibrium solution. As an example consider the simple and irreducible Markov chain in Fig. 2. The computation of its stabilized probabilities is the following:

Step	a	b	c
1st	1.0	1.0	1.0
2nd	0.5	2.0	0.5
3rd	1.0	1.0	1.0
4rd	0.5	2.0	0.5
5rd	1.0	1.0	1.0
.	.	.	.
∞	.	.	.



	a	b	c
a	0.0	1.0	0.0
b	0.5	0.0	0.5
c	0.0	1.0	0.0

Fig. 2 An irreducible Markov chain

We can see that the computation flip-flops between two states and fails to converge to the equilibrium solution. Thus we have a Markov chain where the fixpoint algorithm for Eq. 8 fluctuates.

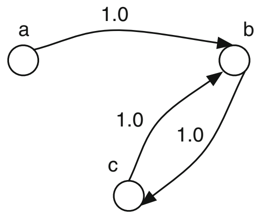
We will next show that Markov chains can be modeled quite naturally in Datalog^{FS}, and this approach provides a simple analysis and efficient algorithms for dealing with this much-studied problem. Let `p_state(X) : K` denote that `K` is the probability of node `X`, $1 \leq X \leq s$, and let `w_matrix(Y, X) : W` denote that the edge from `Y` to `X` has weight `W`. Then the following Datalog^{FS} program models the equilibrium $P = W \cdot P$ in a Markov Chain:

```
p_state(X): K ←    K:[p_state(Y), w_matrix(Y, X)].
w_matrix(1, 1): w11.
w_matrix(1, 2): w12.
⋮
w_matrix(s, s): wss.
```

where the facts in the program are used to assign the initial weights of edges while the first rule computes Eq. 8.

Thus, we have a positive logic program which comes endowed with well-known properties, including the fact that it defines a fixpoint equation that has one or more solutions. The least fixpoint solution is obtained when all `pstate(X)` are false. This is the null solution which is not of any interest in practice, since it corresponds to every city being empty, or every page having rank 0. Thus the case of interest is when there is some non-null solution: if we take a non-null solution vector and multiply each of its components for the same constant we obtain another solution vector. We refer to this transformation as *scaling* of solutions.

Now irreducible Markov chains are guaranteed to have non-null solutions; thus some of their `p_state` are true,



	a	b	c
a	0.0	1.0	0.0
b	0.0	0.0	1.0
c	0.0	1.0	0.0

Fig. 3 A reducible Markov chain

that is, `p_state(X) : K` holds for some node `X` and some $K > 0$ (i.e., node `X` has a positive solution). But since each node is reachable from every other node, via non-zero probability paths, we have that `p_state(X)` : K_X with $K_X > 0$ holds for every node `X`—that is, our Markov chain has a positive solution at every node. Now for Markov chains that have positive solutions at every node, we can use Datalog^{FS} to express and compute efficiently such solutions. All is needed is to state baseline facts that assert the same non-zero multiplicity for each node of our Markov chain. For instance, we can use the multiplicity of `1` and add the facts: `p_state(j)`, $j = 1, \ldots, s$. These will be called *baseline* facts, and the resulting program will be called a baseline program. Thus, for the example at hand we have the following Datalog^{FS} program where the three baseline facts are listed last.

Example 22 A baseline Program for the Markov Chain of Fig. 3.

```
p_state(X) : K ←    K : [p_state(Y), w_matrix(Y, X)].
w_matrix(a, b) : 1.0.
w_matrix(b, a) : 0.5.
w_matrix(b, c) : 0.5.
w_matrix(c, b) : 1.0.
p_state(a).
p_state(b).
p_state(c).
```

where the last three facts are used to denote that `a, b` and `c` are the nodes in the Markov Chain for which the equilibrium is computed. □

The iterative computation of the least fixpoint for the Datalog^{FS} program in Example 22 is as follows:

Step	p_state(a)	p_state(b)	p_state(c)
1st	1.0	1.0	1.0
2nd	1.0	2 .0	1.0
3rd	1.0	2 .0	1.0

where each line in the table corresponds to an iteration of the fixpoint operator, and in the first line, we have all 1s because the baseline is fixed to 1. The iteration converged in three steps to a solution that is a fixpoint for the above program and captures the correct ratio between the probabilities (or the population) of the various nodes. This might in fact be all is needed in applications such as page rank where we are primarily interested in determining the rank order of pages. For other applications, we might need to normalize these results to satisfy constraints about total population or probabilities. For instance, assuming that the sum of probabilities must add up to 1, then the probability for a node X is the final-FS value of this node divided by the sum of values at all nodes:

```
p_norm(X, Pr) ← K1 =![p_state(X)], K2 =![p_state(Y)],
                Pr = K1/K2.
```

Thus for example at hand, the normalized probability values are: $a = 0.25$, $b = 0.5$ and $c = 0.25$.

Therefore, in order to find the fixpoints for program P that models our Markov chain, we added baseline facts and obtained the baseline program Pbl. Pbl is a Datalog^{FS} program for which we can compute the least fixpoint efficiently. Moreover every fixpoint of Pbl is also a fixpoint for P. Indeed, for any interpretation I that contains all the baseline facts, the respective immediate consequence operators produce the same results: that is, $T_P(I) = T_{Pbl}(I)$. Therefore any fixpoint of T_P that contains all the baseline facts is also a fixpoint for T_{Pbl} and vice versa. But since, by its very definition, the least model of Pbl contains all the baseline facts, we have that every fixpoint for T_{Pbl} is also a fixpoint for T_P. The opposite of course is not true since the null fixpoint of T_P, and possibly others, is not fixpoint for T_{Pbl}. However, if T_P has a fixpoint that is positive at all nodes, then by multiplying the frequency at all nodes by a large enough finite constant, we obtain a fixpoint for T_P that contains all the baseline facts of T_{Pbl}. Since each irreducible Markov chain T_P has a finite fixpoint that is null at every node, then there exists a finite fixpoint for T_{Pbl}. Therefore, the least fixpoint for T_{Pbl} is finite. That is, we can state the following theorem:

Theorem 5 *The least fixpoint of the baseline Datalog^{FS} program that models an irreducible Markov chain is finite.*

The operation of multiplying the frequency of the solution at each node by a positive constant will be called scaling. Now since irreducible programs have positive solutions at every node we can state the following theorem:

Theorem 6 *Every non-null solution of an irreducible Markov chain can be obtained by scaling the least fixpoint solution of its baseline Datalog^{FS} model.*

In summary, while there has been a significant amount of previous work on Markov chains, the use of Datalog^{FS} provides us with a model and a simple computation algorithm which is valid for all irreducible Markov chains, including periodic ones.

Obviously there are many Markov chains that are not irreducible, including the one in Fig. 3 where, using 0.1 as base line, the computation of the least fixpoint is as follows:

Step	a	b	c
1st	0.1	0.1	0.1
2nd	0.1	0.2	0.1
3rd	0.1	0.2	0.2
4rd	0.1	0.3	0.2
5rd	0.1	0.3	0.3
.	.	.	.
∞	0.1	∞	∞

Thus, the computation only converges at the first ordinal, producing infinite values and the corresponding least fixpoint is not finite. A practical solution for these situations, which is used for example, in the Page Rank algorithm, is to introduce a damping factor that transforms the chain into an irreducible chain. We are also investigating a different approach, where instead of modifying the network configuration, we use techniques inspired by the computation of greatest fixpoints. This research line is still at its initial stage and will be discussed in future reports.

9 Related work

Logic-based query languages were quintessential in the introduction of the relational data model by E.F. Codd in the 70s; through major efforts in scalable implementation and query optimization this led to the development of relational DBMS in the 80s. Over time, the excitement generated by the extraordinary success of relational DBMS, on both research and business fronts, gave way to the realization that query languages more expressive than relational algebra and SQL-2 were needed; this motivated the work on Datalog and related languages that support rules and recursive queries. Recursive Datalog programs come with a simple and elegant least fixpoint semantics, which is also equivalent to both the logic-based model-theoretic and proof-theoretic semantics of the program clauses [9,25]. Furthermore, the least fixpoint semantics can be efficiently supported by the iterated execution of the rules enhanced by techniques such as the differential fixpoint (a.k.a. seminaive fixpoint) method [18,25,32,33]. This optimization step avoids redundant execution of rules in a bottom-up computation, that is, from the database to the rule goals. In addition to this method, techniques such as Magic-Sets and specialization of left/right recursive rules, are also applicable when some arguments in query goals are required to satisfy certain constraints [18,25,32,33].

However, these desirable properties only hold when the rules define monotonic transformations (w.r.t. set-containment), and negation or aggregates in rules compromise their monotonicity, along with the nice properties above, and the ability of supporting complex applications via recursion. Much research work has therefore sought ways to generalize (i) the semantics of Datalog and (ii) its efficient implementation when negation and aggregates are used in the rules. This work has produced the notion of *stratification* that satisfies both properties, and is also quite simple for users to master [18,25,32,33]. Unfortunately, Datalog programs that are stratified with respect to negation and contain no function symbols cannot support (i) the expression of many queries, including the simple game query discussed in [20], and the summarized part explosion problem discussed in [22], and (ii) the efficient formulation of many other graph optimization algorithms, which typically require the computation of extrema and counts in recursion.

The importance of optimization and graph applications have motivated much research work seeking to solve these problems. In general, said proposals follow three main approaches. The first consists in supporting infinite levels of stratifications using Datalog_{1S} programs [25,34,35]; a second approach instead attempts to preserve the fixpoint computation via continuous aggregates and non-deterministic *choice* constructs [24,36–38]. Finally, the third approach seeks to achieve monotonicity by using partial orders that are more general than set-containment [8,19]. The ingenious solutions proposed in these papers were of limited generality and often required very sophisticated users and compilers (e.g., it was quite difficult for the compiler to decide if the program at hand is monotonic [10]). Datalog^{FS} does not suffer from these problems as we have shown in the paper.

10 Conclusion

In this paper, we studied the important problem of allowing aggregates in recursive Datalog rules, and proposed a solution of surprising simplicity for this challenge. Our Datalog^{FS} approach is based on using continuous aggregate-like functions that allow us to query and reason on the frequency with which predicates and conjunctions of predicates occur. Thus, we extended the declarative semantics of the logic-based paradigm, and showed that Datalog^{FS} programs have very efficient bottom-up execution via a generalized seminaive fixpoint and the Max-based optimization introduced in Sect. 6.2. In particular, we showed that Datalog^{FS} entails concise formulations and efficient implementations for algorithms that require aggregates in logic programs. In fact, Datalog^{FS} supports recursive programs with (i) monotonic counts (Examples 8, 9, 10), (ii) extrema aggregates (Examples 19, 20), and (iii) sum aggregates over positive number (Examples 15, 1). Finally, in the Appendix, we show that aggregates on negative numbers can also be expressed in Datalog^{FS}. These findings suggest that simpler unified notations for aggregates in Datalog^{FS} and Datalog are possible and desirable in practice, and this will be a topic of future work.

The formal results and many examples presented in this paper illustrate that the application range of deductive databases is now significantly wider since (i) queries that are provably not expressible in Datalog with stratified negation and aggregates [20,22] can now be expressed in Datalog^{FS}, and (ii) Datalog^{FS} also entails the efficient formulations of many queries for which only sub-optimal formulations were possible using Datalog with stratified negation and aggregates. Even more significant is perhaps the fact that queries that had not been considered within the natural application domain of Datalog can now be concisely formulated and efficiently supported. In fact, the treatment of Markov Chains presented in Sect. 9 demonstrates that, by providing a simple and effective model for the analysis and solution of complex problems, Datalog^{FS} can also enrich our understanding of application domains. Exploring this opportunity provides a topic for future research, which will also address the parallel executions of Datalog^{FS} queries for advanced analytics.

In conclusion, the theoretical results and several applications presented in the paper prove that Datalog^{FS} entails concise formulations and efficient implementations for a large class of algorithms that could not be supported efficiently or could not be expressed at all in Datalog. These major improvements in power and generality were achieved by adding the monotonic Running-FS construct to Datalog with stratified negation, which is the basic non-monotonic construct that can found in most versions and implementations of this great language.

Acknowledgments We would like to thank Sergio Greco, Oded Shmueli and the reviewers for their comments and suggested improvements. Carlo Zaniolo's work was supported in part by NSF Grant No. IIS 1118107. Edoardo Serra's work was supported by the following projects: FRAME Grant No. PON01-02477, OpenKnowTeck Grant No. DM2130 and TETRIS Grant No. PON01-00451. Mirjana Mazuran's work was partially funded by the Italian project Sensori (Industria 2015—Bando Nuove Tecnologie per il Made in Italy)—Grant n. 00029MI01/2011.

Appendix A: Abstract semantics by example

The abstract semantics of a Datalog^{FS} program **P** is defined by its translation into a set of Horn clauses called the *expansion* of **P**. For example, let us consider the query in Example

11 that counts how many copies of each basic part are contained in assemblies:

```
cassb(Part, Sub): Qty ← assbl(Part, Sub, Qty).
cbasic(Pno): 1 ←           basic(Pno, _).
cbasic(Part): K ←          K:[cassb(Part, Sub), cbasic(Sub)].
```

We rewrite all multi-occuring predicates using `lessthan`, obtaining

$$\begin{array}{ll}
\overline{\mathtt{cassb}}(\mathtt{Part,Sub,J}) \leftarrow & \mathtt{assbl(Part,Sub,Qty)},\\
& \mathtt{lessthan(J,Qty)}.\\
\overline{\mathtt{cbasic}}(\mathtt{Pno,J}) \leftarrow & \mathtt{lessthan(J,1)}, \mathtt{basic(Pno,\_)}.\\
\overline{\mathtt{cbasic}}(\mathtt{Part,J}) \leftarrow & \mathtt{lessthan(J,K)},\\
& \quad \mathtt{K}:[\overline{\mathtt{cassb}}(\mathtt{Part,Sub,J1}), \overline{\mathtt{cbasic}}(\mathtt{Sub,J2})].
\end{array}$$

The third rule evaluates the join of `cassb(Part, Sub, _)` and `cbasic(Sub, _)` on `Sub`, and from their multiplicity of the two operands derives the counts associated with each `Part`. But in terms of formal abstract semantics no count aggregate is used: we instead use the `conj` predicate specific to each b-expression. For the case at hand we obtain (using lists to assemble the values of the three local variables into an object):

$$\begin{array}{ll}
\mathtt{conj(1,Part,[[Sub,J1,J2]])} \leftarrow & \overline{\mathtt{cassb}}(\mathtt{Part,Sub,J1}),\\
& \overline{\mathtt{cbasic}}(\mathtt{Sub,J2}).\\
\mathtt{conj(N1,Part,[[Sub,J1,J2]|T])} \leftarrow & \overline{\mathtt{cassb}}(\mathtt{Part,Sub,J1}),\\
& \overline{\mathtt{cbasic}}(\mathtt{Sub,J2}),\\
& \mathtt{conj(N,Part,T)},\\
& \mathtt{notin([Sub,J1,J2],T)},\\
& \mathtt{N1 = N+1}.\\
\mathtt{notin(Z,[\,])}. & \\
\mathtt{notin(Z,[V|T])} \leftarrow & \quad \mathtt{Z \neq V, notin(Z,T)}.
\end{array}$$

Therefore, the semantic-defining rewriting of our FS-literals expand our Datalog^{FS} program into standard Horn Clauses, for which model-theoretic, proof-theoretic and fixpoint semantics exist and are equivalent. Thus the FS terms can be used freely in recursive rules. However this rewriting does not provide a good starting point for implementation since it is extremely inefficient. Thus, in the example above, we have triples `[Sub, J1, J2]`, and we build all lists of such triplets without repetitions; as we do that, we also count the *N* elements in each of the *N!* lists.[10]

Efficient implementations that are fully consistent with the formal semantics, are however possible starting directly from the FS constructs in the rules and we suggest that this is the approach that should be taken in actual implementations of Datalog^{FS}. In fact, working directly with the FS constructs in the rules is much preferable in terms of performance and it also more attractive in terms of usability. Indeed, concepts such as multiplicity are simple and enough that users would rather program with running-FS and FS-assert terms, rather than their abstract definitions based on lists and recursive predicates. Although Prolog-like implementation approaches are also possible, in this paper we focused on Datalog implementation technology.

[10] A somewhat more efficient computation could be achieved via ordered lists. This approach however is undesirable inasmuch as it requires a totally ordered universe and compromises genericity [18].

Appendix B: The expressive power of running-FS

In Sect. 5, we claimed that the increased expressive power of stratified Datalog^{FS} is due only to the running-FS construct. In this section we prove such assertion by showing that final-FS goals and FS-assert constructs can be rewritten using only running-FS goals and stratified negation.

In Sects. 3.3 and 3.4 we have shown the rewriting in logic programming without lists of final-FS goals and FS-assert, respectively. However, the rewritings that we have shown use integer comparisons (operator $>$ for final-FS goals) and the sum operator (for FS-assert constructs). Now we will show how it is possible to rewrite final-FS goals and FS-assert constructs using only running-FS goals without integer comparison and built-in integer operation as the sum.

As shown in Sect. 3.4, a final-FS goal $\mathtt{Kj} =![\mathtt{exprj(Xj,Yj)}]$ is rewritten as the conjunction:

$$\mathtt{Kj}:[\mathtt{exprj(Xj,Yj)}], \neg\mathtt{morethan}(\mathtt{Y_j},\mathtt{K})$$

where the predicate `morethan` is defined as follows:

$$\mathtt{morethan}(\mathtt{Y_j},\mathtt{K}) \leftarrow \mathtt{K1}:[\mathtt{exprj}(\_,\mathtt{Y_j})], \mathtt{K1} > \mathtt{K}.$$

this rule uses the integer comparison $>$ and this could lead to think that its use adds expressive power to our language. However, this is not the case and in the following we provide a further rewriting that only uses the inequality operator $\neq$:

$$\begin{array}{ll}
\mathtt{morethan}(\mathtt{Y_j},\mathtt{K}) \leftarrow & \mathtt{exprj}(\mathtt{Z_j},\mathtt{Y_j}),\\
& \mathtt{K}:[\mathtt{exprj}(\mathtt{X_j},\mathtt{Y_j}), \mathtt{X_j} \neq \mathtt{Z_j}].
\end{array}$$

Similarly, in the case of FS-assert constructs we have shown, in Sect. 3.3, their rewriting which uses integer comparison and the sum operator as follows:

$$\begin{array}{l}
\mathtt{lessthan(1,K)} \leftarrow \ \mathtt{K} \geq 1.\\
\mathtt{lessthan(J1,K)} \leftarrow \mathtt{lessthan(J,K)}, \mathtt{K}>\mathtt{J}, \mathtt{J1} = \mathtt{J}+1.
\end{array}$$

these rules can be rewritten as follows:

$$\begin{array}{ll}
\mathtt{a(1,0)}. & \\
\mathtt{a(1,1)}. & \\
\mathtt{a(K,1)} \leftarrow & \mathtt{K}:[\mathtt{a(K1,X)}].\\
\mathtt{b(1,K,0)} \leftarrow & \mathtt{a(K,1)}.\\
\mathtt{b(1,K,1)} \leftarrow & \mathtt{a(K,1)}.\\
\mathtt{b(K,K1,1)} \leftarrow & \mathtt{a(K1,\_)}, \mathtt{K}:[\mathtt{b(K2,K1,X)}, \mathtt{K1} \neq \mathtt{K2}].\\
\mathtt{lessthan(K,K1)} \leftarrow & \mathtt{b(K,K1,1)}.
\end{array}$$

where predicate `a` generates all integer numbers from `1` until the maximum precision (as the first rule of the `lessthan` predicate in the previous two rules). Then, the 4th and 5th rule, generate, for each integer, two `b` predicates and the 6th rule, given a number `K1`, generates as many occurrences of predicate b as K1. Finally, the last rule will generate as many predicates lessthan as the required K.

Appendix C: Stratified aggregates

The expressive power of D^a on unordered domains (genericity), is also of significant theoretical interest. This issue has been studied by Mumick and Shmueli in [22], where the basic arithmetic functions $(+, -, *, \ldots)$ were also allowed as built-in. Even with these extensions, D^a cannot express the generalized part explosion query of Example 15. In the rest of this section, we will express Datalog aggregates in Datalog^{FS}. Then, from Example 15 we can conclude that, for unordered domains:

Theorem 7 *$D^a \subsetneq$ Datalog^{FS}*

Let us now discuss how to express D^a in Datalog^{FS}. Mumick and Shmueli describe Datalog with stratified aggregates as an extension of Datalog that permits a "Groupby" predicates of the form

$$\mathtt{GROUPBY}(\mathtt{r}(\overline{\mathtt{t}}), [\mathtt{Y_1}, \mathtt{Y_{2,\ldots,}}\, \mathtt{Y_m}], \mathtt{Z_1} = \mathtt{A_1}(\mathtt{E_1}(\overline{\mathtt{t}})), \ldots, \mathtt{Z_n} = \mathtt{A_n}(\mathtt{E_n}(\overline{\mathtt{t}})))$$

to appear as subgoals in Datalog body rules. The "Groupby" predicate takes as arguments: a predicate r with its attribute list $\overline{\mathtt{t}}$, a grouping list $[\mathtt{Y_1}, \mathtt{Y_{2,\ldots,}}\, \mathtt{Y_m}]$ contained in $\overline{\mathtt{t}}$ and a set of aggregation terms $\mathtt{Z_1} = \mathtt{A_1}(\mathtt{E_1}), \ldots, \mathtt{Z_n} = \mathtt{A_n}(\mathtt{E_n})$. For each aggregation term $\mathtt{Z_i} = \mathtt{A_i}(\mathtt{E_i}(\overline{\mathtt{t}}))$, $\mathtt{Z_i}$ is a new variable, $\mathtt{E_i}(\overline{\mathtt{t}})$ is an arithmetic expression that uses the variables $\overline{\mathtt{t}}$ and $\mathtt{A_i}$ is an aggregate operator, for example sum, count, max, min and avg. Stratification means that if a derived relation $r1$ is defined by applying aggregation on a derived relation $r2$, then $r2$'s definition does not depend, syntactically, on relation $r1$.

Example 23 Stratified aggregate.

$$\mathtt{p}(\mathtt{Y}, \mathtt{Z_1}, \mathtt{Z_2}) \leftarrow \mathtt{GROUPBY}(\mathtt{r}(\_, \mathtt{Y}, \mathtt{F}), [\mathtt{Y}], \mathtt{Z_1} = \mathtt{avg}(\mathtt{F}), \mathtt{Z_2} = \mathtt{sum}(\mathtt{F})).$$

This program returns for each Y value in r the average and sum of value assumed by F in r. □

We can express the predicate "Groupby" by using Datalog^{FS}. Without loss of generality we can consider only one aggregation term. In fact for each rule of the form

$$\begin{aligned}\mathtt{p}(\mathtt{Y_1}, \ldots, \mathtt{Y_m}, \mathtt{Z_1}, \ldots, \mathtt{Z_n}) \leftarrow\ & \mathtt{GROUPBY}(\mathtt{r}(\overline{\mathtt{t}}), [\mathtt{Y_1}, \ldots, \mathtt{Y_m}],\\ & \mathtt{Z_1} = \mathtt{A_1}(\mathtt{E_1}(\overline{\mathtt{t}})), \ldots, \mathtt{Z_n} = \mathtt{A_n}(\mathtt{E_n}(\overline{\mathtt{t}}))).\end{aligned}$$

we can rewrite the program in the following way:

$$\begin{aligned}&\mathtt{r_0} : \mathtt{p}(\mathtt{Y_1}, \ldots, \mathtt{Y_m}, \mathtt{Z_1}, \ldots, \mathtt{Z_n}) \leftarrow \mathtt{p_1}(\mathtt{Y_1}, \ldots, \mathtt{Y_m}, \mathtt{Z_1}), \ldots\\ &\qquad\qquad\qquad\qquad \ldots, \mathtt{p_n}(\mathtt{Y_1}, \ldots, \mathtt{Y_m}, \mathtt{Z_n}).\\ &\mathtt{r_1} : \mathtt{p_1}(\mathtt{Y_1}, \ldots, \mathtt{Y_m}, \mathtt{Z_1}) \leftarrow \quad \mathtt{GROUPBY}(\mathtt{r}(\overline{\mathtt{t}}), [\mathtt{Y_1}, \ldots, \mathtt{Y_m}], \mathtt{Z_1} = \mathtt{A_1}(\mathtt{E_1}(\overline{\mathtt{t}}))).\\ &\vdots\\ &\mathtt{r_n} : \mathtt{p_n}(\mathtt{Y_1}, \ldots, \mathtt{Y_m}, \mathtt{Z_n}) \leftarrow \quad \mathtt{GROUPBY}(\mathtt{r}(\overline{\mathtt{t}}), [\mathtt{Y_1}, \ldots, \mathtt{Y_m}], \mathtt{Z_n} = \mathtt{A_n}(\mathtt{E_n}(\overline{\mathtt{t}}))).\end{aligned}$$

where $r_1, \ldots, r_n$ are rules with "GROUPBY" predicate with only one aggregation term.

Count aggregate We start by showing how the "GROUPBY" predicate with count aggregate can be express in Datalog^{FS}. Consider the following rule

$$\mathtt{p}(\overline{\mathtt{y}}, \mathtt{Z}) \leftarrow \mathtt{GROUPBY}(\mathtt{r}(\_, \overline{\mathtt{y}}, \overline{\mathtt{x}}), [\overline{\mathtt{y}}], \mathtt{Z} = \mathtt{count}(\overline{\mathtt{x}})).$$

where $\overline{\mathtt{y}}$ is the list of grouping variables, $\overline{\mathtt{x}}$ is the list of count variables. The following program can be rewrite in this way:

$$\begin{aligned}&\mathtt{p}(\overline{\mathtt{y}}, \mathtt{Z}) \leftarrow\ \mathtt{r}(\_, \overline{\mathtt{y}}, \_), \mathtt{Z} =![\mathtt{r}'(\overline{\mathtt{y}}, \overline{\mathtt{x}})].\\ &\mathtt{r}'(\overline{\mathtt{y}}, \overline{\mathtt{x}}) \leftarrow\ \mathtt{r}(\_, \overline{\mathtt{y}}, \overline{\mathtt{x}}).\end{aligned}$$

where the rule with predicate r' represents a projection over relation r.

Sum aggregate The next step consists to show how to use the aggregate sum on decimal number. Suppose that we have the scale-up factor $\mathtt{d} = \mathtt{10^n}$ that represent the decimal precision number (see Sect. 7) where n is large enough to represent the decimal numbers used. The rule with sum aggregate has the following form:

$$\mathtt{p}(\overline{\mathtt{y}}, \mathtt{Z}) \leftarrow \mathtt{GROUPBY}(\mathtt{r}(\_, \overline{\mathtt{y}}, \overline{\mathtt{x}}), [\overline{\mathtt{y}}], \mathtt{Z} = \mathtt{sum}(\mathtt{E}(\overline{\mathtt{y}}, \overline{\mathtt{x}}))).$$

where E is a expression composed by built-in functions $(+, -, *, \ldots)$ that use some variables in $\overline{\mathtt{y}}$ and $\overline{\mathtt{x}}$. Initially, we compute for each fact of relation r the expression $\mathtt{E}(\overline{\mathtt{y}}, \overline{\mathtt{x}})$ and we convert such value in a multiplicity by distinguishing its sign.

$$\begin{aligned}&\mathtt{r}''(\overline{\mathtt{y}}, \overline{\mathtt{x}}, +) : \mathtt{K} \leftarrow \mathtt{r}(\_, \overline{\mathtt{y}}, \overline{\mathtt{x}}), \mathtt{Z} = \mathtt{E}(\overline{\mathtt{y}}, \overline{\mathtt{x}}), \mathtt{Z} > 0, \mathtt{K} = \mathtt{Z} * \mathtt{d}.\\ &\mathtt{r}''(\overline{\mathtt{y}}, \overline{\mathtt{x}}, -) : \mathtt{K} \leftarrow \mathtt{r}(\_, \overline{\mathtt{y}}, \overline{\mathtt{x}}), \mathtt{Z} = \mathtt{E}(\overline{\mathtt{y}}, \overline{\mathtt{x}}), \mathtt{Z} < 0, \mathtt{K} = -\mathtt{Z} * \mathtt{d}.\end{aligned}$$

As last we obtain the sum aggregate value by subtract the sum $\mathtt{Z}^-$ of negative numbers from the sum of positive numbers $\mathtt{Z}^+$ in the following way:

$$\begin{aligned}\mathtt{p}(\overline{\mathtt{y}}, \mathtt{Z}) \leftarrow\ & \mathtt{r}(\_, \overline{\mathtt{y}}, \_), \mathtt{Z}^+ =![\mathtt{r}''(\overline{\mathtt{y}}, \overline{\mathtt{x}}, +)],\\ & \mathtt{Z}^- =![\mathtt{r}''(\overline{\mathtt{y}}, \overline{\mathtt{x}}, -)], \mathtt{Z} = (\mathtt{Z}^+ - \mathtt{Z}^-)/\mathtt{D}.\end{aligned}$$

Note that the result value is scaled up by using a factor D.

Average aggregate The average aggregate can be obtained by combining the sum and count aggregate. Thus given the average aggregate rule:

$$\mathtt{p}(\overline{\mathtt{y}}, \mathtt{Z}) \leftarrow \mathtt{GROUPBY}(\mathtt{r}(\_, \overline{\mathtt{y}}, \overline{\mathtt{x}}), [\overline{\mathtt{y}}], \mathtt{Z} = \mathtt{avg}(\mathtt{E}(\overline{\mathtt{y}}, \overline{\mathtt{x}}))).$$

it is possible to rewrite it in Datalog^{FS} by using the following rule:

$$\begin{aligned}\mathtt{p}(\overline{\mathtt{y}}, \mathtt{Z}) \leftarrow\ & \mathtt{r}(\_, \overline{\mathtt{y}}, \_), \mathtt{Z}^+ =![\mathtt{r}''(\overline{\mathtt{y}}, \overline{\mathtt{x}}, +)], \mathtt{Z}^- =![\mathtt{r}''(\overline{\mathtt{y}}, \overline{\mathtt{x}}, -)],\\ & \mathtt{K} =![\mathtt{r}(\_, \overline{\mathtt{y}}, \overline{\mathtt{x}})], \mathtt{Z} = (\mathtt{Z}^+ - \mathtt{Z}^-)/(\mathtt{K} * \mathtt{D}).\end{aligned}$$

where the rule that defines the predicates r'' is the rule used in sum aggregate.

Min and max aggregates For min and max aggregate it is sufficient to use the Negation-stratified Datalog. It follows that for the min aggregate rule:

$$\mathtt{p}(\overline{\mathtt{y}}, \mathtt{Z}) \leftarrow \mathtt{GROUPBY}(\mathtt{r}(\_, \overline{\mathtt{y}}, \overline{\mathtt{x}}), [\overline{\mathtt{y}}], \mathtt{Z} = \mathtt{min}(\mathtt{E}(\overline{\mathtt{y}}, \overline{\mathtt{x}}))).$$

we have the following Negation-stratified Datalog:

$$\mathtt{p}(\overline{\mathtt{y}}, \mathtt{Z}) \leftarrow \ \mathtt{r}(\_, \overline{\mathtt{y}}, \overline{\mathtt{x}}), \mathtt{Z} = \mathtt{E}(\overline{\mathtt{y}}, \overline{\mathtt{x}}), \neg \mathtt{p}'(\overline{\mathtt{y}}, \overline{\mathtt{x}}).$$
$$\mathtt{p}'(\overline{\mathtt{y}}, \overline{\mathtt{x}}) \leftarrow \mathtt{r}(\_, \overline{\mathtt{y}}, \overline{\mathtt{x}}), \mathtt{r}(\_, \overline{\mathtt{y}}, \overline{\mathtt{x}}'), \mathtt{E}(\overline{\mathtt{y}}, \overline{\mathtt{x}}') < \mathtt{E}(\overline{\mathtt{y}}, \overline{\mathtt{x}}).$$

We can obtain the max aggregate by change the $\mathtt{E}(\overline{\mathtt{y}}, \overline{\mathtt{x}}') <$ $\mathtt{E}(\overline{\mathtt{y}, \mathtt{x}})$ atom in the second rule with $\mathtt{E}(\overline{\mathtt{y}, \mathtt{x}'}) > E(\overline{y}, \overline{x})$.

References

1. Hellerstein, J.M.: Datalog redux: experience and conjecture. In: PODS, pp. 1–2 (2010)
2. de Moor, O., Gottlob, G., Furche, T., Sellers, A.J.: Datalog Reloaded-First International Workshop, Datalog 2010, Oxford, UK, 16–19 March 2010, Springer 2011
3. Huang, S.S., Green, T.J., Loo, B.T.: Datalog and emerging applications: an interactive tutorial. In: SIGMOD Conference, pp. 1213–1216 (2011)
4. Loo, B.T., Condie, T., Garofalakis, M.N., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. Commun. ACM **52**(11), 87–95 (2009)
5. Gottlob, G., Orsi, G., Pieris, A.: Ontological queries: rewriting and optimization. In: ICDE, pp. 2–13 (2011)
6. Afrati, F.N., Borkar, V.R., Carey, M.J., Polyzotis, N., Ullman, J.D.: Map-reduce extensions and recursive queries. In EDBT, pp. 1–8 (2011)
7. Zaniolo, C.: Logical foundations of continuous query languages for data streams. Datalog 2012. pp. 177–189 (2012)
8. Ross, K.A., Sagiv, Y.: Monotonic aggregation in deductive database. J. Comput. Syst. Sci. **54**(1), 79–97 (1997)
9. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Berlin (1987)
10. Van Gelder, A.: Foundations of aggregation in deductive databases. In: DOOD, pp. 13–34 (1993)
11. Kanellakis, P.C.: Elements of Relational Database Theory. Technical report, Providence, RI (1989)
12. Ullman, J.D.: Principles of Database and Knowledge-Base Systems. Computer Science Press, Inc., New York (1988)
13. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. MIT Press, London, pp. 1070–1080 (1988)
14. Van Gelder, Allen, Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM **38**(3), 619–649 (1991)
15. Zaniolo, C., Faloutsos, S.C.S.C., Snodgrass, R.T., Subrahmanian, V.S., Zicari, R.: Advanced Database Systems. Morgan Kaufmann (1997)
16. Chomicki, J., Imielinski, T.: Temporal deductive databases and infinite objects. In: PODS, pp. 61–73 (1988)
17. Hirsch, J.E.: An index to quantify an individual's scientific research output that takes into account the effect of multiple coauthorship. Scientometrics **85**(3), 741–754 (2010)
18. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
19. Mumick, I.S., Hamid, Pirahesh, H., Ramakrishnan, R.: The magic of duplicates and aggregates, In: VLDB, pp. 264–277 (1990)
20. Kolaitis, P.G.: The expressive power of stratified logic programs. Inf. Comput. **90**, 50–66 (1991)
21. Dahlhaus, E.: Skolem Normal Forms Concerning the Least Fixpoint. Springer, London (1987)
22. Mumick, I.S., Shmueli, O.: How expressive is stratified aggregation? Ann. Math. Artif. Intell. **15**, 407–435 (1995)
23. Mazuran, M., Serra, E., Zaniolo, C.: Graph Languages in Datalog^{FS}: From Abstract Semantics to Efficient Implementation. Technical report, UCLA (2011)
24. Arni, F., Ong, K.L., Tsur, S., Wang, H., Zaniolo, C.: The deductive database system ldl++. TPLP **3**(1), 61–94 (2003)
25. Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R.T., Subrahmanian, V.S., Zicari, R.: Advanced Database Systems. Morgan Kaufmann, Los Altos (1997)
26. Angles, R., Gutiérrez, C.: Survey of graph database models. ACM Comput. Surv. **40**(1) (2008)
27. Cruz, I.F., Mendelzon, A.O., Wood P.T. A graphical query language supporting recursion. In: SIGMOD Conference, pp. 323–330 (1987)
28. Consens, M.P., Mendelzon, A.O. : Graphlog: a visual formalism for real life recursion. In: PODS, pp. 404–416 (1990)
29. Paredaens, J., Peelman, P., Tanca, L.: G-log: a graph-based query language. IEEE Trans. Knowl. Data Eng. **7**(3), 436–453 (1995)
30. Jackson, M.O., Yariv, L.: Diffusion on Social Networks. Economie Publique (2005)
31. Shakarian, P., Subrahmanian, V.S., Sapino, M.L.: Using generalized annotated programs to solve social network optimization problems. In: ICLP (Technical Communications), pp. 182–191 (2010)
32. Ramakrishnan, R., Gehrke, J.: Database Management Systems. WCB/McGraw-Hill, New York (1998)
33. Ullman, J.D., Widom, J.: A First Course in Database Systems. Prentice-Hall, Prentice (1997)
34. Zaniolo, C., Arni, N., Ong, K.: Negation and aggregates in recursive rules: the ldl++ approach. In: DOOD, pp. 204–221 (1993)
35. Lausen, G., Ludäscher, B., May, W.: On active deductive databases: the statelog approach. In: Transactions and Change in Logic Databases, pp. 69–106 (1998)
36. Ganguly, S., Greco, S., Zaniolo, C.: Minimum and maximum predicates in logic programming. In: PODS, pp. 154–163 (1991)
37. Giannotti, F., Pedreschi, D., Saccà, D., Zaniolo, C.: Non-determinism in deductive databases. In: DOOD, pp. 129–146 (1991)
38. Greco, S., Zaniolo, C.: Greedy algorithms in datalog. TPLP **1**(4), 381–407 (2001)