TLP **13** (4–5): Online Supplement, July 2013.
URL: http://dx.doi.org/10.1017/S1471068413000380

A declarative extension of horn clauses, and its significance for datalog and its applications

MIRJANA MAZURAN
Politecnico di Milano DEIB

EDOARDO SERRA
University of Maryland

CARLO ZANIOLO
University of California, Los Angeles

submitted 10 April 2013; revised 23 June 2013; accepted 5 July 2013

Abstract

FS-rules provide a powerful monotonic extension for Horn clauses that supports monotonic aggregates in recursion by reasoning on the multiplicity of occurrences satisfying existential goals. The least fixpoint semantics, and its equivalent least model semantics, hold for logic programs with FS-rules; moreover, generalized notions of stratification and stable models are easily derived when negated goals are allowed. Finally, the generalization of techniques such as seminaive fixpoint and magic sets, make possible the efficient implementation of Datalog^{FS}, i.e., Datalog with rules with Frequency Support (FS-rules) and stratified negation. A large number of applications that could not be supported efficiently, or could not be expressed at all in stratified Datalog can now be easily expressed and efficiently supported in Datalog^{FS} and a powerful Datalog^{FS} system is now being developed at UCLA.

KEYWORDS: horn clauses, datalog, monotonic aggregates, stable models

1 Introduction

We are currently experiencing a resurgence of interest in Datalog in areas such as parallel and distributed programming (Hellerstein 2010; Abiteboul *et al.* 2011) and Description Logic for ontological queries (Gottlob *et al.* 2011). Other lines of work include execution of recursive queries in the MapReduce framework (Afrati *et al.* 2011) and Data Stream Management Systems (Zaniolo 2011). This renaissance (Barceló and Pichler 2012) and the demands of new applications underscore the need to tackle and solve crucial Datalog problems that have remained open for a long time, and in particular the challenge of aggregates in recursion that has been the focus of much previous work (Mumick *et al.* 1990; Kolaitis 1991; Mumick and Shmueli 1995; Ross and Sagiv 1997; Zaniolo *et al.* 1997; Greco and Zaniolo 2001). The problem is challenging since traditional aggregates, such as those of SQL

or XQuery, violate the requirement of monotonicity on which the least fixpoint semantics of Datalog and declarative logic-based languages is based.

In the next section, we introduce Datalog^{FS} via examples. Then, in Section 3 we define the least-fixpoint semantics of Datalog^{FS} programs. Then, In Section 4 we discuss negation-stratified programs and stable models. In Section 5, we present Datalog^{FS} queries that could not be expressed in Datalog with stratified negation. In Section 6 we consider optimization issues, the use of floating-point numbers in Datalog^{FS} and new applications made possible by these extensions. More complex examples, and the generalization of the magic-set method to Datalog^{FS} programs are given in the appendix.

2 A monotonic extension for horn clauses

There is a big party on campus, and every student is likely to attend the party if at least one of his/her friends will attend. This can be expressed by the following rule:

$$\texttt{attend(Y)} \leftarrow \quad \texttt{student(Y)}, \texttt{attend(X)}, \texttt{friend(Y,X)}.$$

A logical equivalent of this rule is

$$\forall \texttt{Y}(\texttt{attend(Y)} \leftarrow \quad \texttt{student(Y)}, \exists \texttt{X}[\texttt{attend(X)}, \texttt{friend(Y,X)}].)$$

and thus we say that `Y` is the global universal variable and `X` is the local existential one.

But say that, because of finals, the campus mood has turned less festive, and now our student requires that 3 friends attend the party before he or she joins in. Then, we could expand the bracketed expression above into: [`attend(X)`, `friend(X,Y1)`, `friend(X,Y2)`, `friend(X,Y3)`, $\texttt{Y1} \neq \texttt{Y2}, \texttt{Y2} \neq \texttt{Y3}, \texttt{Y3} \neq \texttt{Y1}$]. However, such an expansion becomes unwieldy when the number of required friends increases, and much more complex expansions will actually be required if, instead of a constant value such as 3, we want to have a variable. Therefore, we introduce the following notation:

$$\texttt{attend(Y)} \leftarrow \quad \texttt{student(Y)}, \texttt{3} : [\texttt{attend(X)}, \texttt{friend(Y,X)}].$$

Thus, `3 : [attend(X), friend(Y,X)]` means that there are at least three distinct occurrences of the local variable `X` that make the expression in the brackets true. Following (Mazuran *et al.* 2012), we refer to the conjunct in the brackets as the `b-expression`, to `"3:"` as the `FS-term`, and the whole `3 : [attend(X), friend(Y,X)]` is the `Running FS-goal`. We refer to rules such as that above as FS-rules, i.e., rules with Frequency Support.

The formal semantics of programs with these new constructs is given in the next section, and here we consider a couple of examples that illustrate natural applications of the new construct. The first is our campus-party example.

Example 1

The party organizers always attend. The other students join in when they learn that three or more friends are attending.

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

```
organizer(tom).   organizer(sue).   organizer(pat).

friend(marc, sue).    friend(marc, tom).    friend(marc, pat).
friend(ann, tom).     friend(ann, pat).     friend(ann, marc).

student(marc).        student(ann).

attend(X) ← organizer(X).
attend(Y) ← student(Y), 3 : [attend(X), friend(Y, X)].
```

Thus, tom, pat and sue will attend the party they organize. Also marc is a student who is a friend of the three of them, so he will attend too, as per the first application of the recursive rule. Then ann, who views tom pat and marc as her friends, also joins the party.

Transitive Closure of Functional Dependencies (FDs). The design of relational database schemas is based on the computation of their FDs minimal covers. We assume, without loss of generality, that FDs only have one attribute in their right-hand side, and represent them as shown in Example 2: given the FD $f_j : \mathtt{A}_1, \ldots \mathtt{A}_n \rightarrow \mathtt{B}$, where $\mathtt{A} \ldots \mathtt{A}_n$ and B are the attributes of the relation (i.e., constant strings), its left side is $\mathtt{lsd(fj, A)}, \ldots, \mathtt{lsd(fj, A_n)}$ and its right side is rsd(fj, B, n), where n is the number of arguments of the left side.

Example 2
Representing FDs as facts

```
A → B      lsd(f1, "A").                    rsd(f1,"B", 1).
B → A      lsd(f2, "B").                    rsd(f2,"A", 1).
A → D      lsd(f3, "A").                    rsd(f3,"D", 1).
A, C → E   lsd(f4, "A").  lsd(f4,"C").      rsd(f4,"E", 2).
B, C → E   lsd(f5, "B").  lsd(f5,"C").      rsd(f5,"E", 2).
```

The FS-rules in Example 3 can be used to detect which of the given FDs is redundant, i.e. it can be inferred from the other FDs using their formal properties. Whether an FD is redundant is determined by: (a) entering all attributes from its left-side in the closure set (first rule in Example 3), and (b) adding to the closure set the right-side of each FD iff all its left-side attributes are in the current closure set. This is checked by the FS-goal in the recursive rule , where the condition $\mathtt{F} \neq \mathtt{Fd}$ ensures that the FD being checked for redundancy is not used in the derivation. The last rule in Example 3 checks if the right side of the FD being checked is in the closure set, in which case the FD is redundant.

Example 3
Find FDs derivable from the others.

```
closure(Fd, Y) ← lsd(Fd, Y).
closure(Fd, Z) ← closure(Fd, Y), lsd(F, Y), F ≠ Fd, rsd(F, Z, K),
                 K : [closure(F, Y1), lsd(F, Y1)].
redundant(Fd) ← rsd(Fd, X, _), closure(Fd, X).
```

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

Here, we have represented each FD as a multi-source/single-sink node of a directed hypergraph; thus, the first two rules in our example solve the node-reachability problem for hypergraphs. Similar representations and computations based on hypergraphs have many applications, that can be expressed using FS-rules akin to those in Example 3.

The Apriori Algorithm. This important algorithm has inspired the "frequency support" terminology used in this paper. Apriori (Agrawal and Srikant 1994) computes frequent itemsets in a collection of market baskets. An itemset is considered frequent if its supports, i.e., the number of market baskets containing it, is greater than or equal to a given threshold. The first step of the algorithm finds the frequent items. Then, the algorithm combines these frequent items into a list containing pairs, and counts the frequency of these pairs. From step N=3 on, candidate frequent sets of cardinality N are produced as follows: each candidate set S is obtained by (i) taking the union of two frequent sets of size N-1 sharing N-2 elements, and (ii) verifying that all the N subsets of S of size N-1 are in fact frequent (otherwise S cannot be frequent because of the anti-monotonicity property of frequent itemsets). In the following program, we set the minimum frequency threshold to 35 and assume that a market basket is represented by `basket(BID, Itemlist)`, where BID is the ID of the basket and Itemlist is the ordered list of its items.

Example 4
The APRIORI computation of item sets with frequency support 35.

$$\texttt{freq}(1,[\texttt{Itno}]) \leftarrow 35:[\texttt{basket}(\texttt{BID},\texttt{List}),\texttt{subset}([\texttt{Itno}],\texttt{List})].$$
$$\texttt{cand}(2,[\texttt{A},\texttt{B}]) \leftarrow \texttt{freq}(1,[\texttt{A}]),\texttt{freq}(1,[\texttt{B}]),\texttt{A} < \texttt{B}.$$

$$\texttt{cand}(\texttt{N1},\texttt{ML}) \leftarrow \texttt{freq}(\texttt{N},[\texttt{A}|\texttt{L}]),\texttt{freq}(\texttt{N},[\texttt{B}|\texttt{L}]),\texttt{A} < \texttt{B},\ \texttt{ML} = [\texttt{A}|[\texttt{B}|\texttt{L}]],\texttt{N1} = \texttt{N}+1,\texttt{N1}:[\texttt{freq}(\texttt{N},\texttt{L1}),\texttt{subset}(\texttt{L1},\texttt{ML})].$$
$$\texttt{freq}(\texttt{N1},\texttt{ML}) \leftarrow \texttt{cand}(\texttt{N1},\texttt{ML}),35:[\texttt{basket}(\texttt{BID},\texttt{List}),\texttt{subset}(\texttt{ML},\texttt{List})].$$
$$\texttt{subset}([\,],\texttt{L}).$$
$$\texttt{subset}([\texttt{A}|\texttt{L}],[\texttt{A}|\texttt{List}]) \leftarrow \texttt{subset}(\texttt{L},\texttt{List}).$$
$$\texttt{subset}([\texttt{B}|\texttt{L}],[\texttt{A}|\texttt{List}]) \leftarrow \texttt{A} < \texttt{B},\texttt{subset}([\texttt{B}|\texttt{L}],\texttt{List}).$$

The first rule creates a list with all frequent 1-itemsets (i.e. itemsets containing only one item that appears at least 35 times in the baskets). The second rule generates the candidate 2-itemsets by combining the frequent 1-itemsets (note that the lists representing frequent itemsets are ordered). Then, the third rule generates all candidate itemsets `ML` of length `N+1`, by including both heads of two lists sharing the same tail of length $\texttt{N}-2$ (a computation that, for clarity of explanation is shown via an equality goal). The goal $\texttt{N1}:[\texttt{freq}(\texttt{N},\texttt{L1}),\texttt{subset}(\texttt{L1},\texttt{ML})]$ in the third rule guarantees that all $\texttt{N}+1=\texttt{N1}$ itemsets of length `N`, that are subsets of `ML`, are frequent. Finally, the fourth rule selects itemsets of any length that appear at least 35 times in the baskets. The last three rules are used to verify that a list is a subset of another list.

These examples illustrate that we have extended the traditional rules of logic programs by allowing goals of the form $\texttt{K}:[\texttt{b-expression}]$, where `b-expression`

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

is a conjunct of one or more atoms, in the body of the rules. The new goals are called running-FS goals; the new rules are called FS-rules, and an *FS-program P* is simply a finite set of *FS-rules*. We will now define the formal semantics of FS-programs along the lines used for Definite Clause programs (DC-programs) (van Emden and Kowalski 1976; Lloyd 1987).

3 Formal semantics

For both DC-programs, and FS-programs, let $\mathscr{C}_P$ and $\mathscr{F}_P$ denote, respectively, the constants and the function symbols of our program P. Then, the Herbrand universe for P is denoted by H_P and defined as the set of all possible ground terms recursively constructed by taking constants and function symbols from $\mathscr{C}_P$ and $\mathscr{F}_P$. These notions apply to both DC-programs and FS-programs but for the latter we must include the positive integers in $\mathscr{C}_P$, since these are used in FS-terms. A simpler approach consists in including a '1' constant in $\mathscr{C}_P$ and a successor function symbols `s` in $\mathscr{F}_P$, whereby the natural numbers are represented by $s(1), s(s(1)), \ldots$ as per Datalog$_{1S}$ (Chomicki and Imielinski 1988). We adopt this second representation under which the universe of an FS-program P is the same as that of a DC-program that happens to contain the term `s(1)`. Then, with P either a DC-program or an FS-program, the Herbrand Base of P is the set of all possible ground atoms whose predicate symbols occur in P and whose arguments are elements from H_P. Every subset of B_P is called an *interpretation* for P.

Let P be a DC-program. A *ground instantiation* of a rule r in P denotes any rule obtained by uniformly replacing every *universally quantified* variable in r by a ground term in H_P. The set of the ground instantiations of the rules in P is denoted by $ground(P)$. A model of P is an interpretation for which all the rules in $ground(P)$ are true. These definitions also apply when P is an FS-program. However, there is a major difference, since in DC-programs all variables are universally quantified, whereas in FS-rules the local variables in the b-expressions are considered existentially quantified: therefore these variables are not instantiated, i.e. $ground(P)$ for an FS-program P is only partially instantiated. Given an FS-program, P, we will next define when a rule $r \in ground(P)$ is true in a given interpretation of P, denoted I_P. Our rule r is true if its head is in I_P or some of its goals are not true. FS-rules have two kinds of goals. The first kind are traditional goals, (i.e., atoms as in DC-rules) which are true if they are in I_P. The second kind are the running FS-goals. Each such goal has form $K : [b{-}expr(X)]$, where X denotes the vector of its existentially quantified local variables which have not been instantiated (whereas K is instantiated since, when a variable, it is a universally quantified global variable). Then, $K : [b{-}expr(X)]$ is true in I_P if (i) K is a positive integer and (ii) there are at least K distinct instances of X that make $b{-}expr(X)$ true. ($b{-}expr(X)$ is a conjunct of one or more atoms, and thus it is true if every of its atoms instantiated according to X is in I_P; also two instances of the variable vector X are distinct if they differ in some component.) So, our running FS-goal is true whenever both (i) and (ii) hold. We can now state important properties of FS-programs: their proofs are similar to those of DC programs and is given in Appendix B.

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

Theorem 1
If M_1 and M_2 are two models of an FS-program P, their intersection $M_1 \cap M_2$ is also a model for P.

Least Model Semantics: Thus every FS-program P has the model intersection property, and therefore P has a unique minimal model denoted by $M(P)$. $M(P)$ defines the formal semantics of a given FS-program P.

Let P be an FS-program and I an interpretation for P: the immediate consequence operator $T_P(I)$ is the set of heads of all rules in $ground(P)$ whose body is made true by I. Now, $T_P(I)$ is monotonic and continuous in the lattice of set-containment and thus the following two important properties hold for every FS-program P:

A: *There always exists a least fixpoint for T_P, denoted $lfp(T_P)$.*
B: *Iterating on T_p starting from the empty set provides an equivalent operational semantics: $lfp(T_P) = T_P^{\uparrow\omega}(\emptyset)$.*

Moreover the least fixpoint and least model semantics are equivalent:

Theorem 2
Let P be an FS-program with immediate consequence operator T_P and least model $M(P)$. Then $lfp(T_P) = M(P)$.

The proof of this theorem is also given in Appendix B where we briefly present reasons to believe that an equivalent proof-theoretic semantics exists for FS-programs. However a discussion of SLD-resolution for FS-programs would take us beyond the limited scope of this paper, where we focus on $T_P^{\uparrow\omega}(\emptyset)$ rather than SLD-resolution for the efficient implementation of our Datalog^{FS} programs.

In summary, the key properties of logic programs are preserved in this significant extension that allows us to write simple declarative logic programs for a host of new and interesting applications, such as those of Examples 1, 3, 4. Also these declarative programs are amenable to very efficient implementation using the optimization techniques which will be discussed later. Before that, we would like to stress the naturalness of this extension, by showing that it also dovetails with key non-monotonic extension widely used for logic programs.

4 Negation and stable models

Support for negation and other non-monotonic constructs is critical and much research has focused on providing (i) formal semantics and (ii) efficient implementations for programs containing negated goals. As for (i) the concept of stable models entails a simple and very powerful semantics which encompasses many applications of interest (Gelfond and Lifschitz 1988). We next discuss its generalization to FS-programs. As for (ii) only restricted subfamilies of programs that have stable model semantics are amenable to efficient implementation. Therefore, many Datalog systems only support programs that are stratified with respect to negation, which are simpler to understand and implement efficiently. Stratified negation for Datalog^{FS} programs is discussed in Section 5.

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

The critical point in extending the notion of stable models to FS-programs is on how to treat negated FS-goals, such as in Example 5.

Example 5
Sociable people join parties not attended by two or more people they shun.

$$\texttt{att(X)} \leftarrow \quad \texttt{soc(X)}, \neg\, 2 : [\texttt{att(Y)}, \texttt{shun(X, Y)}].$$

$$\texttt{soc(tom)}. \quad \texttt{soc(sue)}. \quad \texttt{soc(ann)}. \quad \texttt{soc(eve)}.$$

$$\texttt{shun(sue, tom)}. \quad \texttt{shun(sue, ann)}. \quad \texttt{shun(tom, sue)}. \quad \texttt{shun(tom, eve)}.$$

In order to define the formal semantics for these programs we start with the definition of stability transformation:

Definition 1
Stability Transformation. Let I_P be an interpretation of an FS-program P. Then $ground_M(P)$ denotes the program obtained from $ground(P)$ by the following transformation:

(1) remove every rule where some negated goal evaluates to false, and
(2) remove all negated goals from the remaining rules.

For a standard atom g, $\neg g$ is false whenever $g \in I_P$, whereas $\neg K : [b{-}expr(X)]$ with X the vector of its local variables, is false whenever $b{-}expr(X)$ is true for at least K different instances of X. The stability transformation produces a negation-free FS-program. Thus the following definition of stability, for two-valued logic, provides a straightforward generalization of the original one (Gelfond and Lifschitz 1988).

Definition 2
Stable Models: Let P be a program with model M. M is said to be a *stable model* for P, when M is the least model of $ground_M(P)$.

For instance, the ground version of the program in Example 5 contains all the facts listed in the example and the following partially instantiated rules[1]:

$$r_1 : \texttt{att(tom)} \leftarrow \quad \texttt{soc(tom)}, \neg 2 : [\texttt{att(Y)}, \texttt{shun(tom, Y)}].$$
$$r_2 : \texttt{att(sue)} \leftarrow \quad \texttt{soc(sue)}, \neg 2 : [\texttt{att(Y)}, \texttt{shun(sue, Y)}].$$
$$r_3 : \texttt{att(ann)} \leftarrow \quad \texttt{soc(ann)}, \neg 2 : [\texttt{att(Y)}, \texttt{shun(ann, Y)}].$$
$$r_4 : \texttt{att(eve)} \leftarrow \quad \texttt{soc(eve)}, \neg 2 : [\texttt{att(Y)}, \texttt{shun(eve, Y)}].$$

Consider the first interpretation I_1 containing the facts in Example 5 and: `att(tom)`, `att(sue)`, `att(ann)`, `att(eve)`. For this interpretation, the stability transformation eliminates rules r_1 (`tom` shuns `sue` and `eve`) and r_2 (`sue` shuns `tom` and `ann`); also, the negated goals in r_3 and r_4 are removed. The minimal model of the program so produced no longer contains `tom` and `sue`. Thus I_1 is not a stable model.

[1] The ground version of this program also contains (i) the rules obtained by replacing, say, `tom` with arbitrary positive integers and (ii) the rules that generate the positive integers as successors of 1, but also the successors of `tom`, `sue`, `ann` (see Footnote 2) and `eve`. Rules (i) will be eliminated by the stability transformation, while (ii) are preserved, and the minimal model of our reduct remains the same.

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

Consider now interpretation I_2 containing the facts above and att(sue), att(ann), att(eve). The stability transformation eliminates r_1 since tom shuns sue and eve who are attending. The remaining rules remain without their negated goals that were removed by the stability transformation. We obtain a negation-free program having as unique minimal model I_2, which is thus a stable model. Symmetrically, an interpretation I_3 that contains the facts above and att(tom), att(ann), att(eve) is also a stable model.

The stable model semantics of FS-programs is general and elegant. To guarantee efficient computation, negation can be restricted in syntacticly ways. The most popular is stratified negation (with respect to predicates) that we will use in the rest of the paper.

5 FS-assert and final FS-goals

In the previous examples, the FS-values generated by the running FS-goals were only used in the bodies of the rules, but many applications demand that they be included in the head predicate. The FS-assert construct discussed next answers to this need.

For instance, to state that tom has five friends without stating their names we use the following fact: friends(tom) : 5 and view this notation as a shorthand for $\overline{\texttt{friends}}(\texttt{tom}, 1) \ldots \overline{\texttt{friends}}(\texttt{tom}, 5)$. Then, a rule to invite to the party students with more than four friends, could be as follows:

$$\texttt{invite(Y)} \leftarrow \quad \texttt{student(Y)}, 4 : [\texttt{friends(Y)}].$$

that will actually be re-written and evaluated as:

$$\texttt{invite(Y)} \leftarrow \quad \texttt{student(Y)}, 4 : [\overline{\texttt{friends}}(\texttt{Y}, \texttt{J})].$$

Thus, 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 q(X1, ..., Xn) : K and with body Body is replaced by

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

where J does not occur in Body and lesseq(J, K) is a distinguished predicate used to generate all positive integers up to K, included.[2]

(2) Each occurrence of q(X1, ..., Xn) in the body of rules of P so obtained is replaced by: $\overline{\texttt{q}}(\texttt{X1}, \ldots, \texttt{Xn}, \texttt{J})$ where J is a new variable in the rule.

According to these definitions, the assertion of a fact friends(tom) : K subsumes every fact friends(tom) : N with N < K, whereby the computation of the maximum is performed implicitly. The importance of this observation becomes clear when we

[2] In Datalog$_{1S}$ lessthan can be defined as folllows (pint defines the positive integer a la' Peano):

```
lessthan(K,K) ← pint(K).
lessthan(J,K) ← lessthan(s(J),K).
pint(1).
pint(s(J)) ← pint(J).
```

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

use it in recursive rules, as in the following Bill-Of-Materials (BOM) example. Here, our database contains a set of facts `assbl(Part, Subpart, Qty)` which, for each part number, gives 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 (BOM) graph (a 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.

The time needed to deliver a bicycle can be computed as the maximum of the number of days that the various basic parts require to arrive. Thus, we have Example 6, below.

Example 6
More than 10 Days for delivery

```
delivery(Pno):Days  ←  basic(Pno, Days).
delivery(Part):Days ←  assbl(Part, Sub, _), Days : [delivery(Sub)].
late(Part) ←           CDays : [delivery(Part)], CDays > 10.
```

The first rule establishes that for each basic part the number `Days` required for its delivery becomes its FS-value. The recursive rule propagates `Days` from the sub-parts. The last rule retains only the `Parts` that need more than 10 days to be delivered.

Stratifed Negation and Final-FS goals. All examples presented in this section and in Section 2 are monotonic. Non-monotonicity is however required in many real-life applications, and in the rest of the paper we will focus on the many applications that can be expressed using stratified negation for which efficient implementation techniques are known. Thus, we are requiring our programs to be stratified with respect to negation. In the presence of FS-goals the definition of stratified programs is as follows:

Definition 3
A Datalog^{FS} program P with negated goals is stratified when for each rule $r \in P$ the head of r belongs to a stratum higher than all (i) the negated predicates in the body of r and (ii) every predicate appearing in the `b—expression` of a negated FS-goal in r.

For instance, to find the actual number of days required for delivery we can use the following two rules in conjunction with those in Example 6:

Example 7
Exact Count of Days for delivery

```
actual(P, Days) ←  Days : [delivery(P)], ¬morethan(P, Days).
morethan(P, N) ←   N1 : [delivery(P)], N1 > N.
```

For a more concise expression of exact count `K` we introduce a new *exact-count construct denoted by* '=!'. Thus, the rule in the previous example simplifies into:

```
actual(P, Days) ←  Days=![delivery(P)].
```

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

The '=!' is called *Final-FS* construct and its semantics is defined through the rewriting[3] that expanded this last rule into the two of Example 7. Now, a program is stratified with respect to the final-FS construct whenever its expansion is stratified.

We can generalize Definition 3 by changing the second requirement into: (ii) every predicate appearing in the `b−expression` of a Final-FS goal or a negated FS-goal in r. It is easy to show that this is consistent with the rewriting of the rules from Example 7 into the single rule shown above. In fact, a program P is stratified with respect to Final-FS rules if the rewriting of these rules produces a program that is stratified w.r.t. negation.

Another important application is the Summarized Part Explosion query, which counts the number of copies of component `Sub` needed to construct one copy of `Part`. This query that cannot be expressed in Datalog with stratified aggregates (Mumick and Shmueli 1995) can be easily expressed in Datalog^{FS}, as shown in Example 8.

Example 8
Summarized Part Explosion

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

The first two rules establish that, to construct a `Part` we need `Qty` copies of its immediate subpart `Sub` and to construct `Sub` we need only 1 copy of it. The recursive rule adds up the number of copies of `Sub` needed to construct all subparts (immediate and non immediate) of `Part`. The final rule retains the exact count, that is, the maximum value of the counts.

In the next section we introduce optimization techniques that can be used to efficiently compute the products expressed by FS-goal (such as the one in the third rule).

6 Optimization and support for floating point numbers

The significance of stratified Datalog^{FS} is underscored by the fact that (i) it is amenable to efficient implementation, and (ii) it can also express naturally and efficiently computations requiring the use of rational-number or real-number arithmetic.

Efficient implementations of Datalog^{FS} can rely on the fact that the classical optimization techniques of Datalog, including magic sets and differential fixpoint

[3] In general, an arbitrary Final-FS goal `j` has the form $\mathtt{K_j}$ =![$\mathtt{expr_j(X_j, Y_j)}$], where $\mathtt{expr_j(X_j, Y_j)}$ is a b-expression, with the same syntax as that used for Running-FS goals. Thus, $\mathtt{X_j}$ and $\mathtt{Y_j}$ respectively denotes its global variables and the local variables. Now, the formal semantics of the Final-FS goal $\mathtt{K_j}$ =![$\mathtt{expr_j(X_j, Y_j)}$] is defined by rewriting it into the conjunction $\mathtt{K_j}$: [$\mathtt{expr_j(X_j, Y_j)}$], ¬$\mathtt{morethan_j}$ (X_j, K_j) where: $\mathtt{morethan_j(X_j, K)}$ ← $\mathtt{K1 : [expr_j(X_j, \_)], lesseq(s(K), K1)}$.

(Where `lesseq` was defined in the previous footnote.)

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

(a.k.a. seminaive computation), can be generalized to Datalog^{FS} (Mazuran *et al.* 2012). Since most of our readers are already familiar with those techniques, we postpone discussing their generalization to Datalog^{FS} till the Appendix, and focus here instead on the new optimization technique, called *max-optimization*, that was introduced specifically for Datalog^{FS} (Mazuran *et al.* 2012).

The max-based optimization is applied after the magic-set method and the differential fixpoint transformation have taken place, and we have an iterative program that applies the differential version of the rules until no new element can be added. We can now apply optimizations that preserve the *operational* semantics of those rules, and discover that the computation needs only to be performed on each maximum (max for short) produced by the running FS-goals. A first class of programs where the max-based optimization is applicable is exemplified by Example 8 where we have (1) a Final FS-goal, and (2) an FS-assert in the head that just copies, without any constraint, the FS value obtained in the body. Now, at the end of the fixpoint iteration, the Final-FS goal selects the max integer produced by the rules. Also at each step of the iteration, we have (a) the max integer value, let us call it *max*, and (b) all the other integers between 1 and *max*. However, we do not even need to produce the integers in (b), since they are all $\leqslant$ than the *max* value., and this property propagate from the last iteration to the previous ones, as long as the FS-values in the head are computed from those in the body via a *function that is monotonic on positive values*. In our examples so far, the trivial identity function was used to transfer FS-values to the head, but arbitrary arithmetic functions can be used, provided that they are monotonically increasing in the domain of positive numbers. For instance in Example 6, we might want to add to the days required for delivery an additional day for each step of processing and use the following rule:

$$\texttt{delivery(P)}:\texttt{D2} \leftarrow \quad \texttt{assbl(P, Sub, \_)}, \texttt{D}:[\texttt{delivery(Sub)}], \texttt{D2} = \texttt{D} + 1.$$

Datalog^{FS} programs that only use monotonic arithmetic and boolean expressions on conditions on FS-values (i.e., if the expression is true on positive value X then it is also true on every value $> X$) will be called *normal*.[4]

A second class of normal programs is that of Examples 1, 3 and 4 where the values generated by the running-FS goal are not passed to the heads, but are instead tested against conditions in the bodies of the rules. Here too, as long as the functions and the conditions applied upon the FS-values are monotonic, the rules are satisfied if and only if they are satisfied for the maximal values satisfying by the running FS-goals. Observe that Datalog^{FS} programs, other than normal ones, still have a least-fixpoint semantics, but can be very inefficient since they require the computation of the rules to be repeated for each value between one and the max. Moreover all the practical algorithms we have considered only require the use of normal Datalog^{FS} programs. Thus practical systems, e.g., the one developed

[4] The fact that a program is normal is easily checked by the compiler, when the program contains only arithmetic and simple functions, such as addition, multiplication and log, known to be monotonic on positive numbers. User-defined functions, imported with an explicit declaration of monotonicity, should also be allowed.

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

at UCLA, should only be required to support normal programs. As we shall see next, normal programs also include those where the FS-values are arbitrary positive numbers, not just integers.

6.1 From integers to floating-point numbers

The max-based optimization allows us to use very large integers for FS-values, without having to repeat this computation for every integer starting from 1 and up to the max. This is of great practical significance, since it entails support for rational numbers and floating-point numbers. Indeed, computations on rational numbers can be approximated by assuming that all numbers share a very large denominator D whereby all computations can be approximated by arithmetic on their numerators. For instance, given two numbers N_1/D and N_2/D their sum is $(N_1 + N_2)/D$). Moreover their product is $((N_1 \times N_2) \div D)/D$, where the integer division can cause a round-off error. However, round-offs are monotonic functions and thus do not jeopardize the max-based computation of our normal programs. Of course, large denominators are needed in order to obtain precise approximation and floating-point numbers achieve that and are supported very efficiently by modern hardware. For decimal floating points, the smallest value of exponent supported is -95 (or smaller), whereby every number can be viewed as the numerator over the denominator $D = 10^{95}$. But the precision of such representation is also limited by the fact that the mantissa is also of limited length, and this can cause additional roundoff errors. But again round-offs are monotonic in the domain of positive numbers, and the problem they can cause are not different from those that can occur in any other computations and users have learned to cope with via double precision and/or numerical analysis techniques. Therefore, positive floating-point numbers can be used freely in normal Datalog^{FS} programs, which have formal semantics, and provide a very efficient implementation through floating-point arithmetic.[5] Using floating-point numbers and real arithmetic, Datalog^{FS} can express a cornucopia of interesting applications.

6.2 Examples

Say that `arc(a,b):0.66` denotes that a trip started at point `a` will 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 9
Maximum Probability path in a directed Graph

```
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

[5] For simplicity, we have used a decimal base, but the same conclusions hold for other bases.

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

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. Finally, in the head of the last rule, we only retain the maximum V—i.e., the largest probability to succeed. □

Shortest Path in a Directed Graph. The shortest path computation in a directed graph can be expressed in several ways in Datalog^{FS}. The program in Example 10, already proposed in (Mazuran *et al.* 2012), finds the shortest path in a directed graph by implementing a Floyd-like algorithm that uses quadratic rules. While subtraction and division are antimonotonic w.r.t. their second arguments (i.e., if these becomes larger the results become smaller), the program takes advantage of the fact that 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–Warshall algorithm can be expressed as shown in Example 10.

Example 10
The Floyd–Warshall 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.

We now propose a formulation of the need to find the shortest path in a directed graph inspired by a situation where the arcs between nodes describe toll roads between cities. A traveller who is starting from source node a, with pockets full of cash, would like to reach each other node of the graph while retaining as much of that cash as possible in his pocket, i.e., following the lowest-cost path to the nodes. So let the arcs in our graph have the form edge(X, Y, K), where ⟨X, Y⟩ is the edge and K is its weight. Also, let upperb be an upper bound for the estimated lengths of the paths in our graph (for instance the sum of the weights for all arcs in the graph, or the product of an upper bound for the weights of its edges times its estimated diameter.) Then we have the following program:

Example 11
Shortest Path in a Directed Graph

```
spc(a) : upperb.
spc(Y) : K ←          K1 : [spc(X)], edge(X, Y, K2), K = K1 − K2.
sp(X, K) ←            K1 =![spc(X)], K = upperb − K1.
```

https://doi.org/10.1017/S1471068413000380 Published online by Cambridge University Press

The first rule states that we need to cover all edges to reach `a`. In the second rule, for each node `Y`, we subtract the weights of the edges that make `Y` reachable from `a`. Finally, we subtract from `max` the maximum value we have found to obtain the shortest path. □

Therefore, Datalog^{FS} provides several alternative ways to express the shortest path and similar computations in graphs. Other interesting applications discussed in (Mazuran *et al.* 2012) and (Mazuran *et al.* 2013) include Page Rank, Social Networks, Similarity Measures, Subsequence detection, Hidden Markov chains, and reachability in generalized hypergraphs. Yet, other algorithms, including many greedy algorithms, remain beyond the reach of Datalog^{FS} for reasons that are briefly discussed in the appendix.

7 Conclusion

Datalog^{FS} introduces a simple declarative extension for deductive databases that greatly enhances their effectiveness in a wide range of applications. The generalization of Horn Clauses provided by Datalog^{FS}, preserves monotonicity, continuity, and declarative semantics (based on least-fixpoint / minimal model equivalence). It also dovetails with non-monotonic extensions such as stable models and stratification. Thus Datalog^{FS} programs with stratified negation can support applications that could not be expressed at all in standard Datalog, or could only be expressed via very inefficient programs. Furthermore, Datalog^{FS} programs can be efficiently implemented extending Datalog bottom-up implementation technology through generalized seminaive fixpoint and magic-set method, and the newly introduced max-optimization method. A first Datalog^{FS} prototype, called $\mathscr{DeAL}$ (Deductive Application Language), is undergoing testing at UCLA (Shkapsky *et al.* 2013).

Acknowledgements

The authors wold like to thank Jack Minker, Alex Shakpski and the reviewers for their comments and suggested improvements. This material is based on work supported by the National Science Foundation under Grant No. IIS 1218471. Edoardo Serra's work was supported by the following projects: 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.

References

Abiteboul, S., Bienvenu, M., Galland, A. and Antoine, E. 2011. A rule-based language for web data management. In *PODS*, 293–304.

Afrati, F. N., Borkar, V. R., Carey, M. J., Polyzotis, N. and Ullman, J. D. 2011. Map-reduce extensions and recursive queries. In *EDBT*, 1–8.

Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules in large databases. In *VLDB*, 487–499.

Barceló, P. and Pichler, R., Eds. 2012. *Datalog in Academia and Industry–2nd International Workshop, Datalog 2.0*. LNCS, vol. 7494. Springer.

Chomicki, J. and Imielinski, T. 1988. Temporal deductive databases and infinite objects. In *PODS*, 61–73.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. MIT Press, 1070–1080.

Gottlob, G., Orsi, G. and Pieris, A. 2011. Ontological queries: Rewriting and optimization. In *ICDE*, 2–13.

Greco, S. and Zaniolo, C. 2001. Greedy algorithms in datalog. *TPLP 1,* 4, 381–407.

Hellerstein, J. M. 2010. Datalog redux: Experience and conjecture. In *PODS*, 1–2.

Kolaitis, P. G. 1991. The expressive power of stratified logic programs. *Inf. Comp. 90*, 50–66.

Lloyd, J. W. 1987. *Foundations of Logic Programming*, 2nd ed. Springer.

Mazuran, M., Serra, E. and Zaniolo, C. 2012. Extending the power of datalog recursion. In *The VLDB Journal*. Springer-Verlag, 1–23.

Mazuran, M., Serra, E. and Zaniolo, C. July 2013. *A Declarative Extension of Horn Clauses, and its Significance for Datalog and its Applications*. Tech. Rep., UCLA, Computer Science Department, Technical Report No. 130011.

Mumick, I. S., Pirahesh, H. and Ramakrishnan, R. 1990. The magic of duplicates and aggregates. In *VLDB*, 264–277.

Mumick, I. S. and Shmueli, O. 1995. How expressive is stratified aggregation? *Annals of Mathematics and Artificial Intelligence 15*, 407–435.

Ross, K. A. and Sagiv, Y. 1997. Monotonic aggregation in deductive database. *Journal of Computer and System Sciences 54,* 1, 79–97.

Shkapsky, A., Zeng, K. and Zaniolo, C. 2013. Graph queries in a next-generation datalog system. In *VLDB 2013, Demo Track*, 100–104.

van Emden, M. H. and Kowalski, R. A. 1976. The semantics of predicate logic as a programming language. *Journal of the ACM 23,* 4, 733–742.

Zaniolo, C. 2011. The logic of query languages for data streams. In *Logic and Databases 2011. EDBT 2011 Workshops*, 1–2.

Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R. T., Subrahmanian, V. S. and Zicari, R. 1997. *Advanced Database Systems*. Morgan Kaufmann.