Efficient Evaluation of Disjunctive Datalog Queries with Aggregate Functions

Manuela Citrigno[1], Wolfgang Faber[2], Gianluigi Greco[3], and Nicola Leone[4]

[1] DEIS, Università di Bologna, 40136 Bologna, Italy
citrigno@deis.unibo.it
[2] Institut für Informationssysteme, TU Wien, 1040 Wien, Austria
faber@kr.tuwien.ac.at
[3] DEIS, Università della Calabria, 87030 Rende, Italy
ggreco@si.deis.unical.it
[4] Dip. di Matematica, Università della Calabria, 87030 Rende, Italy
leone@mat.unical.it

Abstract. We present a technique for the optimization of (partially) bound queries over disjunctive datalog programs enriched with aggregate functions (Datalog$^{\vee,\mathcal{A}}$ programs). This class of programs has been recently proved to be well-suited for declaratively formalizing repair semantics in data integration systems. Indeed, even though disjunctive programs provide a natural way for encoding the possible repairs (i.e., insertions or deletions of tuples) of an inconsistent database, they do not suffice for applications in real scenarios, where users usually want to build summary views of data residing in different databases.
The technique exploits the propagation of query bindings, and extends the Magic-Set optimization technique (originally defined for non-disjunctive programs without aggregate functions) to Datalog$^{\vee,\mathcal{A}}$ programs. All the algorithms presented in the paper have been fully integrated and implemented in the DLV system – the state-of-the-art implementation of disjunctive datalog.

1 Introduction

Disjunctive datalog (Datalog$^{\vee}$) programs are logic programs where disjunction may occur in the heads of rules [12, 11]. Disjunctive datalog is very expressive in a precise mathematical sense: it allows to express every property of finite ordered structures that is decidable in the complexity class Σ_2^P ($\mathrm{NP}^{\mathrm{NP}}$) [11]. Therefore, under widely believed assumptions, Datalog$^{\vee}$ is strictly more expressive than *normal* (*disjunction-free*) datalog which can express only problems of lower complexity. Importantly, besides enlarging the class of applications which can be encoded in the language, disjunction often allows for representing problems of lower complexity in a simpler and more natural fashion [10].

Recently, disjunctive datalog is employed in several "hot" application areas like information integration and knowledge management. In particular, several

approaches formalizing repair semantics in data integration system by using logic programs have been proposed (see, e.g., [1, 13, 7]), and the exploitation of disjunctive datalog for information integration is the main focus of the INFOMIX project (IST-2001-33570), funded by the European Commission.

Data integration is an important problem, given that more and more data are dispersed over many data sources. In a user-friendly information system, a data integration system provides transparent access to the data, and relieves the user from the burden of having to identify the relevant data sources for a query, accessing each of them separately, and combining the individual results into the global view of the data.

Informally, a data integration system $\mathcal{I}$ may be viewed as system $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ that consists of a global schema $\mathcal{G}$, which specifies the global (user) elements, a source schema $\mathcal{S}$, which describes the structure of the data sources in the system, and a mapping $\mathcal{M}$, which specifies the relationship between the sources and the global schema. Usually, the global schema also contains information about constraints, Σ, such as key constraints or exclusion dependencies issued on a relational global schema. When the user issues a query q on the global schema, the global database is constructed by data retrieval from the sources and q is answered from it. However, the global database might be inconsistent with the constraints Σ.

To remedy this problem, the inconsistency might be eliminated by modifying the database and reasoning on the "repaired" database. To this aim, the idea exploited in the mentioned papers is to encode the constraints Σ of $\mathcal{G}$ into a logic program, Π, using disjunction (or unstratified negation, as done in [7]), such that the stable models of this program yield the repairs of the global database. Answering a user query, q, then amounts to cautious reasoning over the logic program Π augmented with the query, and the retrieved facts $\mathcal{R}$.

An attractive feature of this approach is that disjunctive logic programs serve as executable logical specifications of repair, and thus allow to state repair policies in a declarative manner rather than in a procedural way. Moreover, the effectiveness of the approach is guaranteed by the availability of some efficient inference engines, such as the DLV system [19] and the GnT system [17], and by some optimization techniques for disjunctive programs which have been recently proposed in [14, 8].

However, in spite of its high expressiveness, it has been clearly recognized that classical Datalog$^\vee$ presents some limitations for its applicability to real data integration settings. In data integration contexts, it is reasonable to assume that users should be able to express most of the SQL2 queries. Indeed, SQL2 is widely accepted as the standard query language in the database context. However, Datalog$^\vee$ is not comparable with SQL2 in that some queries that can be expressed in Datalog$^\vee$ cannot be expressed in SQL2 and vice versa. For instance, Datalog$^\vee$ provides the power of recursion and disjunction which cannot be simulated in SQL2, and SQL2 allows aggregate operators (such as SUM, MIN, MAX) and ordering features which cannot be expressed or easily simulated

in classical Datalog$^\vee$. In some cases, aggregate operators can be simulated in Datalog$^\vee$, but this produces inefficient programs and unnatural encodings of the problems.

In [9], the above deficiency of Datalog$^\vee$ has been overcome by extending the language with a sort of aggregate functions (Datalog$^{\vee,\mathcal{A}}$), first studied in the context of deductive databases, and implementing them in DLV [10] – the state-of-the-art Disjunctive Logic Programming system. Under a computational point of view, the resulting formalism turned out to be equivalent to standard Datalog$^\vee$, since 'brave reasoning' for ground programs is Σ_2^P-complete whereas 'cautious reasoning' for ground programs is Π_2^P-complete. However, at the best of our knowledge, no optimizations techniques for the efficient evaluation of disjunctive programs enriched with aggregate functions have been appeared in the literature. Hence, with current implementations of stable model engines, the evaluation of queries over large data sets quickly becomes infeasible because of lacking scalability. This calls for suitable optimization methods that help in speeding up the evaluation of queries, and in making Datalog$^{\vee,\mathcal{A}}$ well suited for real applications in data integration settings.

In this paper, we face such efficiency problems and we present an optimization technique, that is able to support Datalog$^\vee$ programs, enriched with aggregate functions. Specifically, we investigate a promising line of research consisting of the extension of deductive database techniques and, specifically, of binding propagation techniques exploited in the Magic-Set method [24, 2, 4, 23, 18, 22], to nonmonotonic logic languages like disjunctive datalog.

1.1 Related Work

The Magic-Set method is one of the most well-known technique for the optimization of positive recursive Datalog programs due to its efficiency and its generality, even though other focused methods such as the supplementary magic set and other special techniques for linear and chain queries have been proposed as well (see, e.g., [15, 24, 21]). Intuitively, the goal of the Magic-Set method (originally defined for non-disjunctive datalog queries only) is to use the constants appearing in the query to reduce the size of the instantiation by eliminating "a priori" a number of ground instances of the rules which cannot contribute to the derivation of the query goal.

After seminal papers [2, 4], the viability of the approach was demonstrated e.g. in [16, 20]. Lateron, extensions and refinements have been proposed, addressing e.g. query constraints in [23], the well-founded semantics in [18], or integration into cost-based query optimization in [22]. The research on variations of the Magic-Set method is still going on. For instance, in [5] a technique for the class of *soft-stratifiable* programs is given, and in [14] an elaborated technique for disjunctive programs is described.

It has been noted (e.g. in [18]) that in the non-disjunctive case, memoing techniques lead to similar computations as evaluations after Magic-Set transformations. Also in the disjunctive case such techniques have been proposed, e.g.

Hyper Tableaux [3], for which similar relations might hold. However, we leave this issue for future research, and follow [18] in noting that an advantage of Magic-Sets over such methods is that the latter may be more easily combined with other database optimization techniques.

An extension of the Magic-Set method to disjunctive programs is due to [14], where the author observes that binding propagation strategies have to be changed for disjunctive rules so that each time a head predicate receives some binding from the query, it eventually propagates this relevant information to all the other head predicates as well as to the body predicates. An algorithm implementing the above strategy has been also proposed in [14]. Moreover, in [8] some fresh and refined ideas for extending the Magic-Set method to disjunctive datalog queries have been provided, by avoiding some major drawbacks that are intrinsic of the method in [14].

1.2 Contribution

In this paper, we continue on the way paved in [8], and we provide an extension of the Magic-Set method to deal with Datalog$^{\vee,\mathcal{A}}$ programs as well (DMS$^{\mathcal{A}}$ algorithm). Specifically, in Section 2, we preliminarily show how to extend Disjunctive Logic Programming by aggregate functions and we formally define the semantics of the resulting language, named Datalog$^{\vee,\mathcal{A}}$.

Then, in Section 3, we show that in order to make such technique work in the presence of both disjunction and aggregate atoms, traditional Sideways Information Passing Strategies (*SIPS*), cf. [4], simulating the data flow occurring in the top-down evaluation of the query, must be modified by imposing some additional constraints. We provide all the details needed for understanding the main ideas exploited in the design of the DMS$^{\mathcal{A}}$ algorithm, which has been fully implemented and integrated in the DLV system [19] – the state-of-the-art implementation of disjunctive datalog. Finally, in Section 4 we draw our conclusions.

2 The Datalog$^{\vee,\mathcal{A}}$ Language

In this section, we provide a formal definition of the syntax and semantics of the Datalog$^{\vee,\mathcal{A}}$ language – an extension of Datalog$^{\vee}$ by set-oriented functions (also called aggregate functions). We assume that the reader is familiar with standard Datalog$^{\vee}$; we refer to atoms, literals, rules, and programs of Datalog$^{\vee}$, as *standard atoms, standard literals, standard rules*, and *standard programs,* respectively. For further background, see [12, 10].

2.1 Syntax

A (Datalog$^{\vee,\mathcal{A}}$) *set* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{\mathit{Vars} : \mathit{Conj}\}$, where *Vars* is a list of variables and Conj is a conjunction of

standard literals. Intuitively, a symbolic set $\{X{:}a(X,Y), p(Y)\}$ stands for the set of X-values making $a(X,Y), p(Y)$ true, i.e., $\{X : \exists Y s.t.\ a(X,Y), p(Y)\ \textit{is true}\}$. Note that also negative literals may occur in the conjunction $Conj$ of a symbolic set.

A *ground set* is a set of pairs of the form $\langle \bar{t} : Conj \rangle$, where $\bar{t}$ is a list of constants and $Conj$ is a ground (variable free) conjunction of standard literals. An *aggregate function* is of the form $f(S)$, where S is a set, and f is a *function name* among #`count`, #`min`, #`max`, #`sum`, #`times`. An *aggregate atom* is $Lg \prec_1 f(S) \prec_2 Rg$, where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{=, <, \leq, >, \geq\}$, and Lg and Rg (called *left guard*, and *right guard*, respectively) are terms. One of "$Lg \prec_1$" and "$\prec_2 Rg$" can be omitted. An *atom* is either a standard (Datalog$^\vee$) atom or an aggregate atom.

A *(Datalog$^{\vee,\mathcal{A}}$) rule* r is a construct

$$\mathtt{a_1}\ \mathtt{v}\ \cdots\ \mathtt{v}\ \mathtt{a_n}\ \mathtt{:\!-}\ \mathtt{b_1}, \cdots, \mathtt{b_m}.$$

where $a_1, \cdots, a_n,\ b_1, \cdots, b_m$ are atoms, and $n \geq 0$, $m \geq 0$. The disjunction $a_1\ \mathtt{v}\ \cdots\ \mathtt{v}\ a_n$ is the *head* of r, while the conjunction $b_1, ..., b_m$ is the *body* of r. A *(Datalog$^{\vee,\mathcal{A}}$) program* is a set of Datalog$^{\vee,\mathcal{A}}$ rules.

For simplicity, and without loss of generality, we assume that the body of each rule contains at most one aggregate atom. A *global* variable of a rule r is a variable appearing in some standard atom of r; a *local* variable of r is a variable appearing solely in an aggregate function in r.

Stratification. A Datalog$^{\vee,\mathcal{A}}$ program P is *aggregate-stratified* if there exists a function $||\ ||$, called *level mapping*, from the set of (standard) predicates of P to ordinals, such that for each pair a and b of (standard) predicates of P, and for each rule $r \in \mathcal{P}$: (i) if a appears in the head of r, and b appears in an aggregate atom in the body of r, then $||b|| < ||a||$, and (ii) if a appears in the head of r, and b occurs in a standard atom in the body of r, then $||b|| \leq ||a||$.

Example 1. Consider the program consisting of a set of facts for predicates a and b, plus the following two rules:

$$\mathtt{q(X)}\ \mathtt{:\!-}\ \mathtt{p(X)}, \#\mathtt{count}\{\mathtt{Y} : \mathtt{a(Y,X)}, \mathtt{b(X)}\} \leq 2.$$
$$\mathtt{p(X)}\ \mathtt{:\!-}\ \mathtt{q(X)}, \mathtt{b(X)}.$$

The program is aggregate-stratified, as the following level mapping $||\ ||$ satisfies the required conditions: $||\mathtt{a}|| = ||\mathtt{b}|| = 1$; $||\mathtt{p}|| = ||\mathtt{q}|| = 2$.
If we add the rule $\mathtt{b(X)}\ \mathtt{:\!-}\ \mathtt{p(X)}$, then no legal level-mapping exists and the program becomes aggregate-unstratified. □

Intuitively, aggregate-stratification forbids recursion through aggregates, which could cause an unclear semantic in some cases. Consider, for instance, the (aggregate-unstratified) program consisting only of rule $\mathtt{p(a)}\ \mathtt{:\!-}\ \#\mathtt{count}\{\mathtt{X} : \mathtt{p(X)}\} = 0$. Neither $\mathtt{p(a)}$ nor $\emptyset$ is an intuitive meaning for the program. We should

probably assert that the above program does not have any answer set (defining a notion of "stability" for aggregates), but then positive programs would not always have an answer set if there is no integrity constraint. In the following we assume that Datalog$^{\vee,\mathcal{A}}$ programs are safe and aggregate-stratified.

2.2 Semantics

Given a Datalog$^{\vee,\mathcal{A}}$ program $\mathcal{P}$, let $U_\mathcal{P}$denote the set of constants appearing in $\mathcal{P}$, $U_\mathcal{P}^\mathcal{N} \subseteq U_\mathcal{P}$ the set of the natural numbers occurring in $U_\mathcal{P}$, and $B_\mathcal{P}$the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_\mathcal{P}$. Furthermore, given a set S, $\overline{2}^S$ denotes the set of all multisets over elements from S. Let us now describe the domains and the meanings of the aggregate functions we consider.
#`count`: defined over $\overline{2}^{U_\mathcal{P}}$, returns the number of the elements in the set.
#`sum`: defined over $\overline{2}^{U_\mathcal{P}^\mathcal{N}}$, returns the sum of the elements in the set.
#`times`: defined over $\overline{2}^{U_\mathcal{P}^\mathcal{N}}$, returns the product of the elements in the set.[5]
#`min` ; #`max`: defined over $\overline{2}^{U_\mathcal{P}} - \emptyset$, returns the minimum/maximum element in the set (if the set contains also strings, the lexicographic ordering is considered). If the argument of an aggregate function does not belong to its domain, then $\bot$ is returned.

A *substitution* is a mapping from a set of variables to the set $U_\mathcal{P}$ of the constants appearing in the program $\mathcal{P}$. A substitution from the set of global variables of a rule r (to $U_\mathcal{P}$) is a *global substitution for r*; a substitution from the set of local variables of a symbolic set S (to $U_\mathcal{P}$) is a *local substitution for S*. Given a symbolic set without global variables $S = \{Vars : Conj\}$, the *instantiation of set S* is the following ground set of pairs *inst(S)*:
$\{\langle \gamma(Vars) : \gamma(Conj)\rangle \mid \gamma$ *is a local substitution for* $S\}$. Given a substitution σ and a Datalog$^{\vee,\mathcal{A}}$ object *Obj* (rule, conjunction, set, etc.), with a little abuse of notation, we denote by $\sigma(Obj)$ the object obtained by replacing each variable X in *Obj* by $\sigma(X)$.

A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation *inst(S)*. The instantiation $Ground(\mathcal{P})$ of a program $\mathcal{P}$ is the set of all possible instances of the rules of $\mathcal{P}$.

Example 2. Consider the following program $\mathcal{P}_1$:

$$\texttt{q(1)} \vee \texttt{p(2,2).} \qquad \texttt{q(2)} \vee \texttt{p(2,1).}$$
$$\texttt{t(X)} \texttt{ :- q(X)}, \#\texttt{sum}\{\texttt{Y : p(X,Y)}\} > \texttt{1.}$$

The instantiation $Ground(\mathcal{P}_1)$ is the following:

$$\texttt{q(1)} \vee \texttt{p(2,2).} \qquad \texttt{q(2)} \vee \texttt{p(2,1).}$$
$$\texttt{t(1)} \texttt{ :- q(1)}, \#\texttt{sum}\{\langle \texttt{1 : p(1,1)}\rangle, \langle \texttt{2 : p(1,2)}\rangle\} > \texttt{1.}$$
$$\texttt{t(2)} \texttt{ :- q(2)}, \#\texttt{sum}\{\langle \texttt{1 : p(2,1)}\rangle, \langle \texttt{2 : p(2,2)}\rangle\} > \texttt{1.}$$

□

[5] #`sum` and #`times` applied over an empty set return 0 and 1, respectively.

An *interpretation* for a Datalog$^{\vee,\mathcal{A}}$ program $\mathcal{P}$ is a set of standard ground atoms $I \subseteq B_{\mathcal{P}}$. The truth valuation $I(A)$, where A is a standard ground literal or a standard ground conjunction, is defined in the usual way. Besides assigning truth values to the standard ground literals, an interpretation provides the meaning also to (ground) sets, aggregate functions and aggregate literals; the meaning of a set, an aggregate function, and an aggregate atom under an interpretation, is a multiset, a value, and a truth-value, respectively. Let $f(S)$ be a an aggregate function. The valuation $I(S)$ of set S w.r.t. I is the multiset of the first constant of the first components of the elements in S whose conjunction is true w.r.t. I. More precisely,

$$I(S) = [\ t_1 \mid \langle t_1, ..., t_n : Conj \rangle \in S \wedge \text{ Conj is true w.r.t. } I\ ]$$

The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of the function f on $I(S)$. (If the multiset $I(S)$ is not in the domain of f, $I(f(S)) = \bot$.)

An aggregate atom $A = Lg \prec_1 f(S) \prec_2 Rg$ is *true w.r.t. I* if: (i) $I(f(S)) \neq \bot$, and, (ii) the relationships $Lg \prec_1 I(f(S))$, and $I(f(S)) \prec_2 Ug$ hold whenever they are present; otherwise, A is false.

Using the above notion of truth valuation for aggregate atoms, the truth valuations of aggregate literals and rules, as well as the notion of model, minimal model, and answer set for Datalog$^{\vee,\mathcal{A}}$ are an trivial extension of the corresponding notions in Datalog$^{\vee}$ [12].

2.3 Querying Datalog$^{\vee,\mathcal{A}}$ Programs

Let $\mathcal{P}$ be a Datalog$^{\vee,\mathcal{A}}$ program and let $\mathcal{F}$ be a set of facts. Then, we denote by $\mathcal{P}_{\mathcal{F}}$ the program $\mathcal{P}_{\mathcal{F}} = \mathcal{P} \cup \mathcal{F}$. Given a query $\mathcal{Q}$ and an interpretation M of $\mathcal{P}$, $\vartheta(\mathcal{Q}, M)$ denotes the set containing each substitution ϕ for the variables in $\mathcal{Q}$ such that $\phi(\mathcal{Q})$ is true in M. The answer to a query $\mathcal{Q}$ over $\mathcal{P}_{\mathcal{F}}$, under the *brave* semantics, denoted by $Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$, is the set $\cup_M \vartheta(\mathcal{Q}, M)$, such that $M \in \mathcal{MM}(\mathcal{P} \cup \mathcal{F})$. The answer to a query $\mathcal{Q}$ over the facts in $\mathcal{F}$, under the *cautious* semantics, denoted by $Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$, is the set $\cap_M \vartheta(\mathcal{Q}, M)$, such that $M \in \mathcal{MM}(\mathcal{P} \cup \mathcal{F}) \neq \emptyset$. If $\mathcal{MM}(\mathcal{P} \cup \mathcal{F}) = \emptyset$, then all substitutions over the universe for variables in $\mathcal{Q}$ are in the cautious answer. Finally, we say that programs $\mathcal{P}$ and P' are *bravely* (resp. *cautiously*) *equivalent* w.r.t. $\mathcal{Q}$, denoted by $\mathcal{P} \equiv_{\mathcal{Q},b} \mathcal{P}'$ (resp. $\mathcal{P} \equiv_{\mathcal{Q},c} \mathcal{P}'$), if for any set $\mathcal{F}$ of facts $Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}}) = Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$ (resp. $Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}}) = Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$).

3 Magic-Set Method for Datalog$^{\vee,\mathcal{A}}$ Programs

In this section we present the Magic-Set algorithm for Datalog$^{\vee,\mathcal{A}}$ programs (short. DMS$^{\mathcal{A}}$), which has been implemented and integrated into the DLV system [19]. Basically, we adopt a strategy for simulating the top-down evaluation

of a query by modifying the original program by means of additional rules, which narrow the computation to what is relevant for answering the query.

The input to the DMS$^{\mathcal{A}}$ algorithm (see Figure 1) is a disjunctive datalog program with aggregate functions $\mathcal{P}$ and a query $\mathcal{Q}$. If the query contains some non-free IDB predicates, it outputs a (optimized) program $\texttt{DMS}^{\mathcal{A}}(\mathcal{Q},\mathcal{P})$ consisting of a set of *modified* and *magic* rules, stored by means of the sets $\mathit{modifiedRules}(\mathcal{Q},\mathcal{P})$ and $\mathit{magicRules}(\mathcal{Q},\mathcal{P})$, respectively. The main steps of the algorithm DMS$^{\mathcal{A}}$ are illustrated by means of the following running example, which is an adaptation of the "Strategic Companies" example in [6].

Example 3. We are given a collection C of companies producing some goods in a set G, such that each company $c_i \in C$ is controlled by a set of other companies $O_i \subseteq C$. A subset of the companies $C' \subset C$ is a *strategic set* set if it is a minimal set of companies producing all the goods in G, such that if $O_i \subseteq C'$ for some $i = 1, \ldots, m$ then $c_i \in C'$ must hold. This scenario can be modelled by means of the following program $\mathcal{P}_{sc}$.

r_1 : $\texttt{sc}(\texttt{C}_1)$ v $\texttt{sc}(\texttt{C}_2)$:- $\texttt{produced\_by}(\texttt{P},\texttt{C}_1,\texttt{C}_2)$.
r_2 : $\texttt{sc}(\texttt{C})$:- $\texttt{controlled\_by}(\texttt{C},\texttt{C}_1,\texttt{C}_2,\texttt{C}_3)$, $\texttt{sc}(\texttt{C}_1)$, $\texttt{sc}(\texttt{C}_2)$, $\texttt{sc}(\texttt{C}_3)$.

Moreover, a company is dominant if it is strategic and produces only products which are not produced by any other strategic company:

r_3 : $\texttt{dominant}(\texttt{C})$:- $\texttt{sc}(\texttt{C})$, $\#\texttt{sum}\{\texttt{P} : \texttt{produced\_by}(\texttt{P},\texttt{C},\texttt{C}_2),\ \texttt{sc}(\texttt{C}_2)\} = 0$.

Finally, given a company $\texttt{c} \in C$, we consider a query $\mathcal{Q}_{sc} = \texttt{dominant}(\texttt{c})$. □

The key idea of the algorithm is to materialize binding information which would be propagated during a top-down computation by suitable *adornments*. These are strings of the letters b and f, denoting bound or free for each argument of a predicate. First, adornments are created for query predicates. To efficiently manage adornments, we exploit a stack S of predicates for storing all the adorned predicates to be used for propagating the binding of the query: At each step, an element is removed from S, and each defining rule is processed at a time.

The computation starts in step *2* by initializing the variable $\mathit{modifiedRules}(\mathcal{Q},\mathcal{P})$ to the empty set — the need of this structure will be clear in a while. Then, the function ***BuildQuerySeeds*** pushes on the stack S the adorned predicates of $\mathcal{Q}$, and stores in $\mathit{magicRules}(\mathcal{Q},\mathcal{P})$ some facts, called *magic seeds*. Each fact in such a variable is the *magic version* of an adorned atom $\texttt{p}^{\alpha}$ pushed in S, denoted by $\boldsymbol{magic}(\texttt{p}^{\alpha})$, obtained by eliminating all arguments labelled f in α.

Example 4. Given the query $\mathcal{Q}_{sc} = \texttt{dominant}(\texttt{c})$ and the program $\mathcal{P}_{sc}$, ***BuildQuerySeeds*** creates $\texttt{magic\_dominant}^{\texttt{b}}(\texttt{c})$. and pushes $\texttt{dominant}^{\texttt{b}}$ onto the stack S. □

Input: A Datalog$^{\vee}$ program $\mathcal{P}$, and a query $\mathcal{Q} = \mathtt{g}_1(\mathtt{t}_1), \ldots, \mathtt{g}_n(\mathtt{t}_n)$.
Output: The optimized program $\mathtt{DMS}^{\mathcal{A}}(\mathcal{Q}, \mathcal{P})$.
var S: **stack** of adorned predicates; $modifiedRules(\mathcal{Q}, \mathcal{P})$,$magicRules(\mathcal{Q}, \mathcal{P})$: **set** of rules;
begin
1. **if** $\mathtt{g}_1(\mathtt{t}_1), \ldots, \mathtt{g}_n(\mathtt{t}_n)$ has some IDB predicate **then**
2. $modifiedRules(\mathcal{Q}, \mathcal{P}) := \emptyset$; $\langle S,\ magicRules(\mathcal{Q}, \mathcal{P})\rangle :=$ ***BuildQuerySeeds***$(\mathcal{Q})$;
3. **while** $S \neq \emptyset$ **do**
4. $p^{\alpha} := S.$**pop()**;
5. **for each** rule $r \in \mathcal{P}$: $\mathtt{p(t)\ v\ p}_1\mathtt{(t}_1\mathtt{)\ v\ \ldots\ v\ p}_n\mathtt{(t}_n\mathtt{)\ :-\ q}_1\mathtt{(s}_1\mathtt{)}, \ldots, \mathtt{q}_m\mathtt{(s}_m\mathtt{)}$ **do**
6. $r_a :=$ ***Adorn***(r_s, p^{α}, S);
7. $magicRules(\mathcal{Q}, \mathcal{P}) := magicRules(\mathcal{Q}, \mathcal{P}) \ \bigcup$ ***Generate***(r_a);
8. $modifiedRules(\mathcal{Q}, \mathcal{P}) := modifiedRules(\mathcal{Q}, \mathcal{P}) \ \bigcup \{$***Modify***$(r_a)\}$;
9. **end for**
10. **end while**
11. $\mathtt{DMS}^{\mathcal{A}}(\mathcal{Q}, \mathcal{P}) := magicRules(\mathcal{Q}, \mathcal{P}) \ \cup\ modifiedRules(\mathcal{Q}, \mathcal{P})$;
12. **return** $\mathtt{DMS}^{\mathcal{A}}(\mathcal{Q}, \mathcal{P})$;
13. **end if**
end.

Fig. 1. Magic-Set Method for Datalog$^{\vee,\mathcal{A}}$ Programs.

3.1 Adornment

The query adornments are then used to propagate their information into the body of the rules defining it, simulating a top-down evaluation. And, in fact, the core of the technique (steps *4-9*) consists of removing an adorned predicate $\mathtt{p}^{\alpha}$ from the stack S in step *4*, and in propagating its binding in each (disjunctive) rule r in $\mathcal{P}$ of the form

$$r: \ \mathtt{p(t)\ v\ p}_1\mathtt{(t}_1\mathtt{)\ v\ \ldots\ v\ p}_n\mathtt{(t}_n\mathtt{)\ :-\ q}_1\mathtt{(s}_1\mathtt{)},\ \ldots,\ \mathtt{q}_m\mathtt{(s}_m\mathtt{)}.$$

with $\mathtt{n} \geq 0$, having an atom $\mathtt{p(t)}$ in the head (step *5*).

Obviously various strategies can be pursued concerning the order of processing the body atoms and the propagation of bindings. These are referred to as Sideways Information Passing Strategies (*SIPS*), cf. [4]. Any SIPS must guarantee an iterative processing of all body atoms in r, and simulates the data flow occurring in the top-down evaluation of the query, by iteratively processing all the predicates in r.

Roughly speaking, a SIPS act as follows. Let $\mathtt{q}$ be an atom that has not yet been processed, then its adorned version is created by assuming constants and variables occurring in already considered atoms to be bound, which is denoted by $\mathtt{v} \rightarrow_{\mathtt{X}} \mathtt{q}$, where $\mathtt{X}$ is the set of the variables assumed to be bound which propagate their values into $\mathtt{q}$, and $\mathtt{v}$ is the set of the predicates in which these variables occur. The formal definition of SIPS is provided below.

Definition 1. Let r be a rule having $\mathtt{p}$ in the head, and let $\mathtt{p}^{\alpha}$ be an adornment. A *SIPS* for r is a labelled bipartite graph $\langle V_1 \cup V_2, E\rangle$, where V_1 is the set of subset of $B(r) \cup \{\mathtt{p}^{\alpha}\}$, $V_2 \in B(r)$, and E is a set of arcs satisfying the following conditions:

1. each arc is of the form $v \rightarrow_X s$, where $v \in V_1$ and $s \in V_2$, where X is a non-empty set of variables such that (i) each variable in X appears in s and in either a bound argument position of $\mathtt{p}^{\alpha}$ or a positive body literal of v, and

(ii) for each literal in v there exists a sequence of literals $v = l_0, l_1, ..., l_m = s$ with l_i and l_{i+1} sharing at least a common argument.

2. there exists a total order of $B(r) \cup \{\mathtt{p}^\alpha\}$ in which
   (a) $\mathtt{p}^\alpha$ precedes all members of $B(r)$,
   (b) any literal which does not appear in the graph follows every literal that appears in the graph, and
   (c) for each arc $v \rightarrow_X s$, if $u \in v$ the u precedes s. □

It is well know that if we are able to construct a SIPS for a given rule r and a predicate $\mathtt{p}^\alpha$, then we can use its edges for simulating the data flow from the head to the body of a rule, and, hence, for deriving the adornment of the rule, which is, in fact, performed in the step *6*.

Example 5. Consider the rule $\mathtt{path}(\mathtt{X},\mathtt{Y})$:- $\mathtt{path}(\mathtt{X},\mathtt{Z})$, $\mathtt{path}(\mathtt{Z},\mathtt{Y})$. together with query $\mathtt{path}(1,5)$?. Then, the adornment of the query predicate, i.e., $\mathtt{path}^{\mathtt{bb}}(1,5)$, passes its binding information to $\mathtt{path}(\mathtt{X},\mathtt{Z})$ through $\mathtt{path}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y}) \rightarrow_{\{\mathtt{X}\}} \mathtt{path}(\mathtt{X},\mathtt{Z})$, which causes the generation of the adorned predicate $\mathtt{path}^{\mathtt{bf}}(\mathtt{X},\mathtt{Z})$. Then, we apply $\{\mathtt{path}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y}),\ \mathtt{path}^{\mathtt{bf}}(\mathtt{X},\mathtt{Z})\} \rightarrow_{\{\mathtt{X},\mathtt{Y},\mathtt{Z}\}} \mathtt{path}(\mathtt{Z},\mathtt{Y})$, generating $\mathtt{path}^{\mathtt{bb}}(\mathtt{Z},\mathtt{Y})$. The resulting adorned rule is $\mathtt{path}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$:- $\mathtt{path}^{\mathtt{bf}}(\mathtt{X},\mathtt{Z})$, $\mathtt{path}^{\mathtt{bb}}(\mathtt{Z},\mathtt{Y})$. □

We point out that, for each rule, it is possible to derive different SIPS, associated to all the possible permutations of the atoms appearing in the body. The choosing of a strategy does not matter in the case of positive programs, but it represents a serious issue in the case of Datalog$^{\vee,\mathcal{A}}$ programs, as shown in the following section.

3.2 Binding Propagation in Datalog$^{\vee,\mathcal{A}}$ Programs

Aggregate Atoms. Let us first consider the binding propagation in the presence of aggregate atoms. We recall that an aggregate atom has the form $L_g \leq f\{Vars : Conj\} \leq U_g$, where $Vars$ are variables local w.r.t. the function f, while $Conj$ is a conjunction of literals. All the variables occurring in predicates of $Conjs$ that are not in $Vars$ are said *global* variables.

Since $Conjs$ might contain some variables that are used into other predicates of the rule, we can exploit these variables for propagating the binding into the aggregate atom, too. Then, in the adornment step, literals in $Conj$ can be treated as they were part of the rule; nonetheless some further attention is needed for ensuring the correctness of the SIPS implemented. In fact, literals in $Conj$ have not to be used for propagating bindings to other literals, and, hence, they should be considered at the end of the adornment process. To this aim we extend any standard SIPS, by introducing the additional constraint of preferring for binding propagation aggregate atoms only if there are no other atoms to be processed. Moreover, when only aggregate atoms remain to be processed we prefer the ones having the maximum number of bound variables.

Example 6. Consider again Example 3. When $\texttt{dominant}^{\texttt{b}}$ is removed from the stack, we select rule r_3 for its adornment. Then, $\texttt{C}$ is the unique bound variable and might propagate its binding to both $\texttt{sc}(\texttt{C})$ or to $\texttt{sc}(\texttt{C}_2)$ trough the fact $\texttt{produced\_by}(\texttt{P}, \texttt{C}, \texttt{C}_2)$. However, non-aggregate atoms are always processed first, and hence $\texttt{dominant}^{\texttt{b}}(\texttt{c})$, passes its binding information to $\texttt{sc}(\texttt{C})$ through $\texttt{dominant}^{\texttt{b}}(\texttt{C}) \rightarrow_{\{\texttt{C}\}} \texttt{sc}(\texttt{C})$. Then, we apply $\{\texttt{sc}^{\texttt{b}}(\texttt{C}), \texttt{produced\_by}(\texttt{P}, \texttt{C}, \texttt{C}_2)\} \rightarrow_{\{\texttt{C},\texttt{C}_2\}} \texttt{sc}(\texttt{C}_2)$, generating $\texttt{sc}^{\texttt{b}}(\texttt{C}_2)$. The resulting adorned rule is

$$r_{3_a} : \texttt{dominant}^{\texttt{b}}(\texttt{C}) \texttt{ :- } \texttt{sc}^{\texttt{b}}(\texttt{C}),\ \#\texttt{sum}\{\texttt{P} : \texttt{produced\_by}(\texttt{P}, \texttt{C}, \texttt{C}_2),\ \texttt{sc}^{\texttt{b}}(\texttt{C}_2)\} = \texttt{0}.$$

and the adorned predicate $\texttt{sc}^{\texttt{b}}$ is pushed on the stack S. □

Disjunctive Programs. Let us now consider the case of disjunctive programs without aggregate functions. Then, as first observed in [14], while in nondisjunctive programs bindings are propagated only head-to-body, any sound rewriting for disjunctive programs has to propagate bindings also head-to-head in order to preserve soundness. Roughly, suppose that a predicate $\texttt{p}$ is relevant for the query, and a disjunctive rule $\texttt{r}$ contains $\texttt{p}(\texttt{X})$ in the head. Then, besides propagating the binding from $\texttt{p}(\texttt{X})$ to the body of $\texttt{r}$ (as in the nondisjunctive case), a sound rewriting has to propagate the binding also from $\texttt{p}(\texttt{X})$ to the other head atoms of $\texttt{r}$. Consider, for instance, a Datalog$^{\vee}$ program $\mathcal{P}$ containing rule $\texttt{p}(\texttt{X})\ \texttt{v}\ \texttt{q}(\texttt{Y})\ \texttt{:-}\ \texttt{a}(\texttt{X}, \texttt{Y}),\ \texttt{r}(\texttt{X}).$ and the query $\texttt{p}(\texttt{1})?$. Even though the query propagates the binding for the predicate $\texttt{p}$, in order to correctly answer the query, we also need to evaluate the truth value of $\texttt{q}(\texttt{Y})$, which indirectly receives the binding through the body predicate $\texttt{a}(\texttt{X}, \texttt{Y})$. For instance, suppose that the program contains facts $\texttt{a}(\texttt{1}, \texttt{2})$, and $\texttt{r}(\texttt{1})$; then atom $\texttt{q}(\texttt{2})$ is relevant for query $\texttt{p}(\texttt{1})?$ (i.e., it should belong to the magic set of the query), since the truth of $\texttt{q}(\texttt{2})$ would invalidate the derivation of $\texttt{p}(\texttt{1})$ from the above rule, because of the minimality of the semantics.

It follows that, while propagating the binding, the head atoms of disjunctive rules must be all adorned as well. We achieve this by defining an extension of any non-disjunctive SIPS to the disjunctive case. The constraint for such a disjunctive SIPS is that head atoms (different from $\texttt{p}(\texttt{t})$) cannot provide variable bindings, they can only *receive* bindings (similarly to negative literals in standard SIPS). So they should be processed only once all their variables are bound or do not occur in yet unprocessed body atoms.[6] Moreover they cannot make any of their free-variables bound.

The function ***Adorn*** produces an adorned disjunctive rule from an adorned predicate and a suitable unadorned rule by employing the refined SIPS, pushing all newly adorned predicates onto S. Hence, in step *6* the rule r_a is of the form

$$r_a : \texttt{p}^{\alpha}(\texttt{t})\ \texttt{v}\ \texttt{p}_1^{\alpha_1}(\texttt{t}_1)\ \ldots\ \texttt{p}_\texttt{n}^{\alpha_\texttt{n}}(\texttt{t}_\texttt{n})\ \texttt{:-}\ \texttt{q}_1^{\beta_1}(\texttt{s}_1), \ldots, \texttt{q}_\texttt{m}^{\beta_\texttt{m}}(\texttt{s}_\texttt{m}).$$

[6] Recall that the safety constraint guarantees that each variable of a head atom also appears in some positive body-atom.

Example 7. Consider again Example 3. When $\mathtt{sc}^{\mathtt{b}}$ is removed from the stack, we first select rule r_1 and the head predicate $\mathtt{sc}(\mathtt{C}_1)$. Then, the adorned version is

$$r'_{1_a} : \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1) \ \mathtt{v} \ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2) \mathtt{:-} \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$

Next r_1 is processed again, this time with head predicate $\mathtt{sc}(\mathtt{C}_2)$, producing

$$r''_{1_a} : \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2) \ \mathtt{v} \ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1) \mathtt{:-} \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$

and finally, processing r_2 we obtain

$$r_{2_a} : \mathtt{sc}^{\mathtt{b}}(\mathtt{C}) \mathtt{:-} \mathtt{controlled\_by}(\mathtt{C}, \mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3), \ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1), \ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2), \ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_3). \ \square$$

3.3 Generation

The algorithm uses the adorned rule r_a for generating and collecting the *magic rules* in step 7, which simulate the top-down evaluation scheme. Since r_a is in general a disjunctive rule with aggregate atoms, ***Generate*** first produces a non-disjunctive intermediate rule, say r'_a by moving head atoms into the body and by replacing each aggregate atom, say $L_g \leq f\{Vars : Conj\} \leq U_g$, by the conjunction $Conj$.

Then, for each adorned atom $\mathtt{p}$ in the body of an adorned rule r'_a, a magic rule r_m is generated such that (i) the head of r_m consists of $\boldsymbol{magic}(\mathtt{p})$, and (ii) the body of r_m consists of the magic version of the head atom of r'_a, followed by all of the predicates of r'_a which can propagate the binding on $\mathtt{p}$.

Example 8. In the program of Example 6, from the rule r'_3, we first derive the following standard rule

$$\mathtt{dominant}^{\mathtt{b}}(\mathtt{C}) \mathtt{:-} \mathtt{sc}^{\mathtt{b}}(\mathtt{C}), \ \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}, \mathtt{C}_2), \ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2).$$

and, then, the magic rules

$$\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}) \mathtt{:-} \mathtt{magic\_dominant}^{\mathtt{b}}(\mathtt{C}), \ \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}, \mathtt{C}_2), \ \mathtt{st}^{\mathtt{b}}(\mathtt{C}_2).$$
$$\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_2) \mathtt{:-} \mathtt{magic\_dominant}^{\mathtt{b}}(\mathtt{C}), \ \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}, \mathtt{C}_2), \ \mathtt{st}^{\mathtt{b}}(\mathtt{C}_2).$$

Similarly, by looking at Example 7, from the rule r'_{1_a} first its non-disjunctive intermediate rule

$$\mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1) \mathtt{:-} \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2), \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$

is produced, from which the magic rule

$$\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_2) \mathtt{:-} \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_1), \ \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$

is generated. Similarly, from the rule r''_{1_a} we obtain

$$\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_1) \mathtt{:-} \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_2), \ \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$

and finally r_{2_a} gives rise to the following rules

$$\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_1) \mathtt{:-} \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}), \ \mathtt{controlled\_by}(\mathtt{C}, \mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3).$$
$$\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_2) \mathtt{:-} \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}), \ \mathtt{controlled\_by}(\mathtt{C}, \mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3).$$
$$\mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_3) \mathtt{:-} \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}), \ \mathtt{controlled\_by}(\mathtt{C}, \mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3).$$

$\square$

3.4 Modifications

In step *8* the *modified rules* are generated and collected. These rules represent the rewriting of the original program in which the instantiation of body predicates is limited by the magic predicates. Specifically, the function ***Modify*** constructs a rule of the following form

$$\mathtt{p}(\mathtt{t}) \vee \mathtt{p}_1(\mathtt{t}_1) \vee \ldots \vee \mathtt{p}_n(\mathtt{t}_n) \text{ :- } \textbf{\textit{magic}}(\mathtt{p}^{\alpha}(\mathtt{t})),\ \textbf{\textit{magic}}(\mathtt{p}_1^{\alpha_1}(\mathtt{t}_1)), \ldots,\ \textbf{\textit{magic}}(\mathtt{p}_n^{\alpha_n}(\mathtt{t}_n)),\ \mathtt{q}_1(\mathtt{s}_1), \ldots, \mathtt{q}_m(\mathtt{s}_m).$$

Finally, after all the adorned predicates have been processed the algorithm outputs the program $\mathtt{DMS}^{\mathcal{A}}(\mathcal{Q}, \mathcal{P})$.

Example 9. In our running example, we derive the following set of modified rules:

$$r'_{1_m} : \mathtt{sc}(\mathtt{C}_1) \vee \mathtt{sc}(\mathtt{C}_2) \text{ :- } \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_1),\ \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_2),\ \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$
$$r''_{1_m} : \mathtt{sc}(\mathtt{C}_2) \vee \mathtt{sc}(\mathtt{C}_1) \text{ :- } \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_2),\ \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}_1),\ \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}_1, \mathtt{C}_2).$$
$$r_{2_m} : \mathtt{sc}(\mathtt{C}) \text{ :- } \mathtt{magic\_sc}^{\mathtt{b}}(\mathtt{C}),\ \mathtt{controlled\_by}(\mathtt{C}, \mathtt{C}_1, \mathtt{C}_2, \mathtt{C}_3),\ \mathtt{sc}(\mathtt{C}_1),\ \mathtt{sc}(\mathtt{C}_2),\ \mathtt{sc}(\mathtt{C}_3).$$
$$r_{3_m} : \mathtt{dominant}(\mathtt{C}) \text{ :- } \mathtt{magic\_dominant}^{\mathtt{b}}(\mathtt{C}),\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}),\ \#\mathtt{sum}\{\mathtt{P} : \mathtt{produced\_by}(\mathtt{P}, \mathtt{C}, \mathtt{C}_2),\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2)\} = 0.$$

where r'_{1_m} (resp. r''_{1_m}, r_{2_m}, r_{3_m}) is derived by adding magic predicates and stripping off adornments for the rule r'_{1_a} (resp. r''_{1_a}, r_{2_a}, r_{3_a}). Thus, the optimized program $\mathtt{DMS}^{\mathcal{A}}(\mathcal{Q}_{sc}, \mathcal{P}_{cs})$ comprises the above modified rules as well as the magic rules in Example 8, and the magic seed $\mathtt{magic\_dominant}^{\mathtt{b}}(\mathtt{c})$. □

We conclude the exposition of this algorithm by stressing that the rewriting computed throughout its application is, in fact, an equivalent rewriting of the input program, in the sense provided by the following proposition.

Theorem 1 (Soundness of the DMS$^{\mathcal{A}}$ Algorithm). *Let $\mathcal{P}$ be a Datalog$^{\vee}$ program, let $\mathcal{Q}$ be a query. Then, $\mathtt{DMS}^{\mathcal{A}}(\langle \mathcal{Q}, \mathcal{P} \rangle) \equiv_{\mathcal{Q},b} \mathcal{P}$ and $\mathtt{DMS}^{\mathcal{A}}(\langle \mathcal{Q}, \mathcal{P} \rangle) \equiv_{\mathcal{Q},c} \mathcal{P}$ hold.*

4 Conclusions

Motivated by the application in data integration settings, we have presented a technique for the optimization of (partially) bound queries that extends the Magic-Set method to the case of disjunctive programs with aggregate operators. The technique has been fully implemented into the DLV system.

We point out that our investigation can be of a great interest in several other applicative domains. In fact, aggregate functions in logic programming languages appeared already in the 80s, when their need emerged in deductive databases like LDL. Currently, they are supported in the Smodels system, besides DLV, and their importance in knowledge representation tasks is widely recognized, since they can be simulated only by means of inefficient and unnatural encodings of

the problems. As an example, suppose that a user wants to know if the sum of the salaries of the employees working in a team exceeds a given budget. To this end, the user should first order the employees defining a successor relation. Then she should define a *sum* predicate, in a recursive way, which computes the sum of all salaries, and compare its result with the given budget. This approach has two drawbacks: (1) It is bad from the KR perspective, as the encoding is not natural at all; (2) It is inefficient, as the (instantiation of the) program is quadratic (in the cardinality of the input set of employees).

Concerning future work, our objective is to extend the Magic-Set method to the case of disjunctive programs with constraints and unstratified negation, such that it can be fruitfully applied on arbitrary DLV programs. We believe that the framework developed in this paper is general enough to be extended to these more involved cases.

Acknowledgments

The research was supported by the European Commission under the INFOMIX project (IST-2001-33570).

References

1. O. Arieli, M. Denecker, B. Van Nuffelen, and M. Bruynooghe. Database repair by signed formulae. In *Proc. of FoIKS'04*, volume 2942 of *LNCS*, pp. 14–30. Springer, February 2004.
2. F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. of PODS'86*, pp. 1–16, 1986.
3. P. Baumgartner, U. Furbach, and I. Niemelä. Hyper Tableaux. In *Proc. of JELIA'96*, number 1126 in LNCS, pp. 1–17. Springer, 1996.
4. C. Beeri and R. Ramakrishnan. On the power of magic. *JLP*, 10(1–4):255–259, 1991.
5. A. Behrend. Soft stratification for magic set based query evaluation in deductive databases. In *Proc. of PODS'03*, pp. 102–110. ACM Press, 2003.
6. M. Cadoli, T. Eiter, and G. Gottlob. Default Logic as a Query Language. *TKDE*, 9(3):448–463, 1997.
7. A. Calì, D. Lembo, and R. Rosati. Query rewriting and answering under constraints in data integration systems. In *Proc. 18th Int'l Joint Conference on Artificial Intelligence (IJCAI 2003)*, pp. 16–21, 2003.
8. C. Cumbo, W. Faber, and G. Greco. Improving Query Optimization for Disjunctive Datalog. In *Proc. of the Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003*, pp. 252–262, 2003.
9. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proc. 18th Int'l Joint Conference on Artificial Intelligence (IJCAI 2003)*, pp. 847–852, 2003.
10. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. *Logic-Based Artificial Intelligence*, pp. 79–103. Kluwer, 2000.

11. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *TODS*, 22(3):364–418, September 1997.
12. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
13. G. Greco, S. Greco, and E. Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. In P. Codognet, editor, *Proc. 17th Int'l Conference on Logic Programming (ICLP 2001)*, LNCS 2237, pp. 348–364. Springer, 2001.
14. S. Greco. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. *IEEE TKDE*, 15(2):368–385, March/April 2003.
15. S. Greco, D. Saccà, and C. Zaniolo. The PushDown Method to Optimize Chain Logic Programs (Extended Abstract). In *ICALP'95*, pp. 523–534, 1995.
16. A. Gupta and I.S. Mumick. Magic-sets Transformation in Nonrecursive Systems. In *Proc. of PODS'92*, pp. 354–367, 1992.
17. T. Janhunen, I. Niemelä, P. Simons, and J.-H. You. Partiality and Disjunctions in Stable Model Semantics. In *Proc. of KR'00*, April 12-15, Breckenridge, Colorado, USA, pp. 411–419. Morgan Kaufmann.
18. D.B. Kemp, D. Srivastava, and P.J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theoretical Computer Science*, 146:145–184, July 1995.
19. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *to appear* in ACM TOCL.
20. I.S. Mumick, S.J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proc. of SIGMOD'00*, pp. 247–258, 1990.
21. R. Ramakrishnan, Y. Sagiv, J.D. Ullman, and M.Y. Vardi. Logical Query Optimization by Proof-Tree Transformation. *JCSS*, 47(1):222–248, 1993.
22. P. Seshadri, J.M. Hellerstein, H. Pirahesh, T.Y.C. Leung, R. Ramakrishnan, D. Srivastava, P.J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proc. of SIGMOD'96*, pp. 435–446. ACM Press, June 1996.
23. P.J. Stuckey and S. Sudarshan. Compiling query constraints. In *Proc. of PODS'94*, pp. 56–67. ACM Press, May 1994.
24. J. D. Ullman. *Principles of Database and Knowledge Base Systems.*