DOOD'93

# Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}$++ Approach

Carlo Zaniolo
Computer Science Department, University of California
Los Angeles, California 90024
zaniolo@cs.ucla.edu

Natraj Arni and Kayliang Ong
Microelectronics and Computer Technology Corporation
Austin, Texas 78759
natraj@mcc.com and kayliang@mcc.com

**Abstract**

The problem of allowing non-monotonic constructs, such as negation and aggregates, in recursive programs represents a difficult challenge faced by current research in deductive databases. In this paper, we present a solution that combines generality with efficiency, as demonstrated by its implementation in the new $\mathcal{LDL}$++ system. A novel and general treatment of set aggregates, allowing for user-defined aggregates, is also presented.

## 1 Introduction

The area of non-monotonic reasoning has benefited significantly from research in deductive databases, as demonstrated by the introduction of the concept of stratified negation and stratified set aggregates [16]. This concept is significant since it removes several of the limitations and problems of Prolog's negation-by-failure, and it is conducive to efficient implementation, as demonstrated by systems such as *Glue-Nail*, $\mathcal{LDL}$ and *CORAL* [15, 22, 3]. However, as experience was gained with real-life applications [23], it became clear that stratification is too restrictive and there remain many important applications where negation and set aggregates are needed: such applications, range from processing Bill of Materials to finding the shortest path in a graph [5, 6].

Therefore, during the last five years, a substantial research effort has been devoted to solving the non-stratification issue. This endeavor has produced significant progress on the theoretical front, with the introduction of concepts such as locally stratified programs [16], well-founded models [12], and stable models [7], but it has not yet begotten a solution that is both general and efficient. Indeed our experience with deductive database applications [24], suggests that a practical solution must satisfy three difficult requirements, inasmuch as it must

- have a formal logic-based semantics
- have a simple and intuitive constructive semantics
- be amenable to efficient implementation.

Thus, in addition to requiring formal semantics and efficient implementation, any practical proposal must also stress the importance of having a simple concrete semantics: i.e., one that can be easily mastered by the application programmer, without a need for understanding demanding formalisms. For instance, a notion such as stratified programs can be mastered by a programmer, who can make full use of it without having to understand its formal perfect-model semantics. Furthermore, a compiler can verify efficiently that a given program satisfies the stratification condition, and then support such programs by a bottom-up computation. However, a familiarity with the abstract formalisms of logic and non-monotonic semantics is required to understand and make effective usage of notions such as well-founded models or stable models in writing programs. Furthermore, no simple syntactic check exists for deciding whether a program has a well-founded model or a stable model; when such models exist their computation can be very expensive. Notions such as modularly stratified programs, represent a first step toward efficient execution [20] but leave other issues unresolved—such as compile-time decidability, and uniform treatment of negation and set aggregates.

In this paper, we take a different approach— the synthesis of our experience with concrete $\mathcal{LDL}$ applications [23], and classical optimization problems [5, 6]. We focus on syntactically decidable subclasses of non-monotonic recursive programs which capture the expressive power of inflationary fixpoint semantics [14]. Therefore, we introduce a special class of locally stratified programs that is easy for a compiler to recognize and implement using the fixpoint-based computation of deductive DBs. This class, called XY-stratified programs is first introduced in Section 2, which focuses on programs with negated goals in recursive rules. In Section 3, we extend these notions to recursive rules with aggregates, as demanded by most of the critical applications. In Section 4, we concentrate on the implementation problem, and other practical issues, such as termination. In Section 5 we discuss the generality of the proposed approach by showing that classical problems requiring non-stratified negation and aggregates can be expressed naturally by programs in this class. We also show that the magic-set method and similar re-writing techniques used to propagate constraints in deductive databases, are applicable to XY-stratified programs as well.

We assume our reader to be familiar with the basic concepts pertaining to Datalog and logic rules, including the concept of locally stratified programs and the *iterated fixpoint procedure* to compute the perfect model of these programs [16, 18]. Given a program $P$, a set of rules of $P$ defining a maximal set of mutually recursive predicates will be called a *recursive clique* of $P$.

## 2 XY-Programs

We begin with a simple example that computes the nodes Y of a graph g reachable from a given node a, along with least numbers of arcs needed to reach Y from a:

**Example 1** *Reachable nodes:*

$$\begin{aligned}
&\mathtt{r_0} : \mathtt{delta(nil, Y)} \leftarrow && \mathtt{g(a, Y).}\\
&\mathtt{r_1} : \mathtt{delta(s(I), Y)} \leftarrow && \mathtt{delta(I, X), g(X, Y),}\\
& && \neg\mathtt{all(I, Y).}\\
&\mathtt{r_2} : \mathtt{all(s(I), Y)} \leftarrow && \mathtt{all(I, Y), delta(s(I), _).}\\
&\mathtt{r_3} : \mathtt{all(I, Y)} \leftarrow && \mathtt{delta(I, Y).}
\end{aligned}$$

This program presents several unusual traits. A most obvious one is the presence of terms such as nil, I, s(I) in the first arguments of the recursive predicates. These arguments will be called *stage arguments*, and their usage is for counting as in the recursive definition of integers: nil stands for zero and s(I) stands for I+1. [1]

The intuitive meaning of the program of Example 1 is quite obvious: it implements a fixpoint computation of a transitive closure, through the use of two predicates: delta contains the new values, and all is the union of all values computed so far. In rule $r_1$ all is used for checking that no previous (therefore shorter) path exists to this node.

The formal semantics of the program also supports its intuitive semantics. Because of its particular syntactic form, the program is locally stratified. Each stratum contains all atoms with the same number of occurrences of the successor function symbol "s" in their stage arguments. The strata are also ordered according to the number of occurrences of s in the stage argument.

For the program in Example 1, the bottom stratum has the form:

delta(nil, ...), all(nil, ...), delta(a, ...), all(a, ...)

the next stratum consists of all atoms of the form:

delta(s(nil), ...), all(s(nil), ...), delta(s(a), ...), all(s(a), ...)

and so on. As we shall see later, this particular syntactic form of the recursive rules, w.r.t. the stage arguments, makes it simple for a compiler to detect the occurrence of such a locally stratified program. Furthermore, this type of program can be implemented efficiently using a modified fixpoint computation. It is well-known that the perfect model of such a program is characterized by a transfinite computation called iterated fixpoint[16]. This proceeds as follows:

[1] In this paper we are using function symbols (interpreted and non-interpreted) to express more realistic applications. However, function symbols are not required for a formal treatment: an ordered domain of adequate size and a successor relationship is all it is needed. These can be expressed in pure Datalog (with the choice construct), thus allowing a formal analysis of the data complexity and expressive power of our programs.

the least fixpoint for the bottom stratum is first computed. Then, once the least fixpoint is computed for the $n$-th stratum, the least fixpoint is computed for the $n+1$-th stratum. This transfinite computation of perfect models simplifies dramatically for the program at hand.

The computation of the bottom stratum simply copies all the `g` facts describing our graph. Then, the fixpoint at the lowest stratum of recursive atoms is reached after firing rule $r_0$ followed by $r_3$ thus obtaining all the perfect-model atoms with a stage value of `nil`. The fixpoint for the stratum `s(nil)` is reached by firing rule $r_1$ followed by $r_2$ and $r_3$. Then, the higher strata are inductively generated by firing these three rules in the same order. Therefore, the general transfinite procedure to compute perfect models here reduces to the customary fixpoint iteration. Moreover, various improvements can be made to this fixpoint computation to ensure that it executes efficiently. In fact, while rule $r_2$ seems to suggest that a complete copying of the old relation is needed at each step, no such operation is needed in reality. In fact, at each step of the computation, the only instances of rules that can produce new atoms are those instantiated with stage values from the current stratum: values from the old strata are not used and can be discarded. Thus, if we keep the values of the stage variable (denoting the stratum at which we are working) in a separate memory cell, all that is needed to perform the copying operation is to increase the value of this variable by one. Then, the only computation actually performed at each iteration is the firing of rules $r_1$ and $r_3$; thus the transfinite computation of this perfect model reduces to the usual seminaive fixpoint computation of a transitive closure.

As we formalize these ideas, we find that atoms with the same stage values for the general case can be partitioned into finer strata.

Given a recursive clique, $Q$, the first arguments of the recursive predicates of a rule $r$ in $Q$ will be called the *stage arguments* of $r$[2]. We have the following definitions:

**Definition 1** *Let $Q$ be a recursive clique and $r$ be a recursive rule of $Q$. Then $r$ is called an*

- **X-rule** *if all the stage arguments of $r$ are equal to a simple variable, say $J$, which does not appear anywhere else in $r$;*

- **Y-rule** *if (i) some positive goal of $r$ has as stage argument a simple variable $J$, (ii) the head of $r$ has stage argument $s(J)$, (iii) all the remaining stage arguments are either $J$ or $s(J)$ and (iv) $J$ does not appear anywhere else in $r$.*

In Example 1, $r_3$ is an X-rule, while $r_1$ and $r_2$ are Y-rules. A recursive clique $Q$ will be said to be an *$XY$-clique when all its recursive rules are either $X$-rules or $Y$-rules.*

[2]This is only a matter of convention. Alternatively, we could let the last arguments of recursive predicates be our stage arguments.

**Priming:** $\mathtt{p}'(...)$ *will be called the primed version of an atom* $\mathtt{p}(...)$. *Given an XY-clique, $Q$, its primed version $Q'$, is constructed by priming certain occurrences of recursive predicates in recursive rules as follows:*

- *X-rules: all occurrences of recursive predicates are primed,*
- *Y-rules: the head predicate is primed, and so is every goal with stage argument equal to that of the head.*

*The primed version of our example is as follows:*

$$
\begin{aligned}
\mathtt{r}_0 : \mathtt{delta(nil,Y)} \leftarrow\ & \mathtt{g(a,Y)}.\\
\mathtt{r}_1 : \mathtt{delta'(s(I),Y)} \leftarrow\ & \mathtt{delta(I,X),g(X,Y)},\\
& \neg\mathtt{all(I,Y)}.\\
\mathtt{r}_2 : \mathtt{all'(s(I),Y)} \leftarrow\ & \mathtt{all(I,Y),delta'(s(I),\_)}.\\
\mathtt{r}_3 : \mathtt{all'(I,Y)} \leftarrow\ & \mathtt{delta'(I,Y)}.
\end{aligned}
$$

**Definition 2** *An XY-clique $Q$ will be said to be* XY-stratified *when*

- *The primed version of $Q$ is non-recursive*
- *All exit rules have as stage argument the same constant.*

A program will be said to be *XY-stratified* when every recursive rule that contains the set aggregation construct "< >" in its head or a negated recursive goal in its body belongs to an XY-stratified clique.

The dependency graph for a primed clique provides a very simple syntactic test on whether a program is $XY$-stratified. Furthermore, these programs are amenable to efficient implementation as proven in the following discussion.

The primed version $Q'$ of an XY-stratified clique defines a non-recursive program: thus, the dependency graph of $Q'$ contains no cycle. Thus there exists a topological sorting of the nodes of $Q'$ which obeys stratification, and such that the unprimed predicate names precede the primed ones. For Example 1, for instance, the following is a topological sorting:

$$\mathtt{all} < \mathtt{delta} < \mathtt{delta}' < \mathtt{all}' \ .$$

The tail of such a sorting, restricted to the primed predicate names will be called a *primed sorting* of $Q$. For the previous example, we obtain the following primed sorting:

$$\mathtt{delta}' < \mathtt{all}' \ .$$

Now, we can partition the atoms in Herbrand Base $B_Q$ of the *original* $Q$ into classes according to their predicate name and their stage arguments as follows:

- there is a distinct class, say $\sigma_0$, containing all instances of non-recursive predicates in $Q$—i.e., those without a stage argument.

- all atoms with the same recursive predicate name and the same number of function symbols `s` in the stage argument belong to the same equivalence class $\sigma_{n,p}$, with $n$ denoting the number of `s` function symbols in the stage argument of $p$. Thus, $all(s(nil),...)$, and $all(s(a),...$, belong to the same class (stratum) in our example.

The partition $\Sigma$ of $B_Q$ so constructed can be totally ordered, by letting $\sigma_0$ be the bottom stratum in $\Sigma$, and then letting $\sigma_{n,p} \prec \sigma_{m,q}$ if

- $n < m$, or
- if $n = m$ but $p'$ precedes $q'$ in the primed sorting of the clique.

The totally ordered $\Sigma$ so constructed will be called the *stage layering* of $B_Q$. For Example 1, let $C$s denote either $nil, a$ or 1 and the $V$s denote elements in $Q$'s Herbrand Universe; then $\Sigma$ has the following form:

$$\{\mathtt{g}(\mathtt{V}_1,\mathtt{V}_2)\} \prec$$
$$\{delta(C_1,V_3,V_4)\} \prec \{all(C_2,V_5,V_6\} \prec$$
$$\{delta(s(C_3),V_7,V_8)\} \prec \{all(s(C_4),V_9,V_{10})\} \prec ...$$

Then, we have the following theorem:

**Theorem 1** *Each XY-stratified clique $Q$ can be locally stratified according to a stage layering of $B_Q$. Then, for every instance $r$ of each rule in $Q$, the head of $r$ belongs to a layer strictly higher than the layers for the goals in $r$ (strict stratification).*

Proof: If $g$ is an unprimed goal $g$ for $r$, then the stage argument of the head of $r$ $h(r)$ contains one more `s` than $g$. If $g$ is primed, then $h(r)$ is also primed and its name follows the name of $g$ in the primed ordering. $\Box$

Since the stratification is strict, in computing the iterated fixpoint, *the saturation for each stratum is reached in one step.* Therefore, the compiler can resequence the rules according to the primed sorting of their head names. Then, having derived all atoms with stage value $J$ a *single pass through the rules of $Q$ ordered according to the primed sorting* computes all the atoms with stage value $s(J)$.

Therefore, if $p'$ is the $k$-th predicate name in the primed sorting, let $T_k$ denote the *immediate consequence operator* [13, 18] for the recursive rules in $Q$ defining $p$. The *composite consequence operator* $\Gamma_Q$, will be defined as follows:

$$\Gamma_Q(I) = T_n(T_{n-1}...(T_1(I))...))$$

where $I$ denotes subset of $Q$'s Herbrand Base $B_Q$, and $n \geq 1$. In Example 1, $\Gamma_Q$ is the composition of the immediate consequence operator for rule $r_1$ with that for the two rules $r_2$ and $r_3$.

For $\Gamma_Q$ to be applied correctly, all the atoms with lower values of stage variables must have been produced. Consider now the exit rules of $Q$, and let $T_0$ be the immediate consequence operator for these rules. By the second condition of XY-stratification, all ground atoms in $T_0(\emptyset)$ share the same stage argument. However, additional ground atoms with the same stage value might be obtained by firing the $X$-rules (only one pass through these rules is needed if we follow the topological stratification.) Therefore, with $p_k$ the $k$-th predicate name in the primed sorting, let $T_k^X$ denote the immediate consequence operator for X-rules with head name $p_k$, if any such rules exists, and the identity transformation otherwise. Thus, we can define the *composite consequence operator for the X-rules*, $\Gamma_Q^X$ as follows:

$$\Gamma_Q^X(I) = T_n^X(T_{n-1}^X ... (T_1^X(I))...))$$

Thus, the ground atoms with same stage arguments as the exit rules are: $\Gamma_Q^X(T_0(\emptyset))$.

Therefore, for Example 1, $\Gamma_Q^X$ is the immediate consequence operator for rule $r_3$, thus, all atoms with a stage argument of *nil* can be computed by firing $r_0$ followed by $r_3$.

Now let us define, as customary, $T^0(I) = I$ and, by induction, $T^{n+1}(I) = T(T^n(I)) \cup T^n(I)$. Then $T^\omega(I)$ denotes the union of all $T^k(I)$, for $k \geq 0$, i.e., the result of applying the operator $T$ up to the the first ordinal, beginning from $I$. Then, we can state our key result:

**Theorem 2** *Let $Q$ be a $XY$-stratified clique, with composite consequence operator $\Gamma_Q$ and composite consequence operator for X-rules $\Gamma_Q^X$, then*

- *$Q$ is locally stratified,*
- *Let $M_Q$ be the perfect model of $Q$. Then $M_Q = \Gamma_Q^\omega(M_{nil})$, where $M_{nil} = \Gamma_Q^X(T_0(\emptyset))$ and $T_0$ is the immediate consequence operator for the exit rules of $Q$*

*Proof: It follows directly from the the previous discussion.* □

Thus the perfect model of an XY-stratified clique can be constructed as in a positive program: by a fixpoint iteration to the first ordinal. Computation of $XY$-stratified programs proceeds as that for stratified programs: all the non-recursive predicates in the recursive XY-clique must be saturated before the recursive rules in the clique are computed.

# 3 Aggregates, Conditionals and Choice

Many important applications, such as BOM applications, and graph problems, require the use of set aggregates in non-stratified in programs. For instance, Floyd's least paths algorithm can be written as follows:

**Example 2** *Floyd's Algorithm.*

$$
\begin{array}{ll}
\mathtt{delta(nil, X, X, 0).} & \\
\mathtt{delta(s(J), X, Z, min(< C >))} \leftarrow & \mathtt{delta(J, X, Y, C1),} \\
& \mathtt{g(Y, Z, C2), C = C1 + C2,} \\
& \mathtt{if(all(J, X, Z, C3)\ then\ C3 > C).} \\
\mathtt{all(s(J), X, Z, C)} \leftarrow & \mathtt{all(J, X, Z, C), \neg delta(s(J), X, Z, \_).} \\
\mathtt{all(J, X, Z, C)} \leftarrow & \mathtt{delta(J, X, Z, C).}
\end{array}
$$

In $\mathcal{LDL}$, set aggregates are not built-in but defined by the user. All the user needs to do is to specify the rules defining (i) the aggregate for singleton set `single` and (ii) aggregate for $S = S_1 \cup S_2$ knowing the aggregates of the disjoint $S_1$ and $S_2$ `multi` . For instance, the definition of `min` is

$$
\begin{array}{ll}
\mathtt{single(min, X, X).} & \\
\mathtt{multi(min, Y1, Z1, Z)} \leftarrow & \mathtt{if(Y1 < Z1\ then\ Z = Y1\ else\ Z = Z1).}
\end{array}
$$

Therefore, we now need to provide a formal semantics for such extensible aggregates: whereas previous approaches have focused on a fixed set of primitives [10, 19, 13]. Here we propose a semantics that is simple, general and conducive to efficient implementation, as demonstrated by the $\mathcal{LDL}$++ system [2]. Our approach builds on the choice construct, where formal logic-based semantics, usability in programming and computational efficiency have successfully converged— a second example of such a convergence after stratification.

The idea of `choice` was introduced in [11] to express non-determinism in a declarative fashion. Thus, a construct such as `choice((X), (Y))` is used to denote that the functional dependency $X \rightarrow Y$ must hold in the models defining the meaning of this program, whereby the following program assigns to each student an advisor from the same area:

$$
\begin{array}{ll}
\mathtt{st\_ad(St, Ad)} \leftarrow & \mathtt{major(St, Area), faculty(Ad, Area),} \\
& \mathtt{choice((St), (Ad)).}
\end{array}
$$

Thus, while the result of joining `major` and `faculty` normally results in a many-to-many relation between students and professors (sharing the same area), the `st_ad` relation is restricted to be many-to-one, as a result of the presence of `choice((St), (Ad))`. In $\mathcal{LDL}$, this very powerful construct was disallowed in recursion, inasmuch as the functional-dependency based semantics proposed in [11] suffers from technical problems such as a lack of justifiability property and its unsuitability to efficient implementation due to its static nature [21]. These problems were solved by using, instead, a semantics based on the use of negation and stable models [8]. For instance, the meaning of the rule above is defined by its rewriting into the following non-stratified program:

**Example 3** *An advisor for each student by non-stratified negation.*

$$
\begin{array}{ll}
\mathtt{st\_ad(St, Ad)} \leftarrow & \mathtt{major(St, Area), faculty(Ad, Area),} \\
& \mathtt{chosen(St, Ad).}
\end{array}
$$

$$\texttt{chosen(St,Ad)} \leftarrow \quad \texttt{major(St,Area),faculty(Ad,Area),}$$
$$\neg\texttt{diffChoice(St,Ad)}.$$

$$\texttt{diffChoice(St,Ad)} \leftarrow \quad \texttt{chosen(St,}\overline{\texttt{Ad}}\texttt{),Ad} \neq \overline{\texttt{Ad}}.$$

Programs so rewritten always have one or more stable model, and in each such a model the `chosen` atoms obey the *FD*: $\texttt{St} \rightarrow \texttt{Ad}$ [21]. The new semantics agrees with the original semantics for non-recursive programs, but solves the problems of the old semantics in programs where `choice` occurs in a recursive predicate. For instance, say that an undirected graph is stored as pairs of edges, `g(Y,X)`, `g(X,Y)` then a spanning tree for this graph, starting from the source node `a`, can be computed as follows:

**Example 4** *Finding the spanning tree of a graph.*

$$\texttt{st(nil,a)}$$
$$\texttt{st(X,Y)} \leftarrow \quad \texttt{st(Z,X),g(X,Y),Z} \neq \texttt{Y,choice((Y),(X))}.$$

Now, while the declarative semantics of choice rests on the theoretical foundations of non-monotonic logic, a programmer need not be cognizant of the notion of stable models to make effective use of choice in his/her program; indeed, there is an equivalent operational semantics based on the notion of memorizing the old values of choice and checking each candidate new value for violations of the *FD* constraints. This operational semantics is also the basis for efficient implementations of the construct in $\mathcal{LDL}$++.

The semantics-by-rewriting approach used for choice can be generalized to set aggregates. Assume, for instance that there exists only one aggregate rule, as follows:

**Example 5** *The sum of Y-values, after grouping by X-values.*

$$\texttt{a}_0 : \texttt{p(X,sum(}< \texttt{Y} >\texttt{))} \leftarrow \quad \texttt{body(X,Y)}.$$

The intuitive meaning of this rule, is that, for each X-value in `body(X,Y)`, the sum of Y-values associated with this `X` must be computed and entered as second argument in `P`. This sum can be computed as follows:

**Example 6** *Aggregates Defined Using Choice.*

$$\texttt{a}_1 : \texttt{aggr(OP,0,X,nil)} \leftarrow \quad \texttt{body(X,\_)}.$$

$$\begin{aligned}\texttt{a}_2 : \texttt{aggr(OP,s(J),X,Z)} \leftarrow \quad & \texttt{aggr(OP,J,X,W),body(X,Y),}\\ & \texttt{choice((X,s(J)),(Y)),choice((X,Y),(s(J))),}\\ & \texttt{single(OP,Y,SY),}\\ & \texttt{if(J = 0 then Z = SY}\\ & \qquad\quad \texttt{else multi(OP,SY,W,Z))}.\end{aligned}$$

$$\texttt{a}_3 : \texttt{single(sum,X,X)}.$$
$$\texttt{a}_4 : \texttt{multi(sum,Y1,Z1,Z)} \leftarrow \quad \texttt{Z = Z1 + Y1}.$$

Now observe that the two choice goals in rule $a_2$ imply that a new value of Y is selected at each step (for each X), evaluated using `single`, and (except for the first step, $\mathtt{J} = \mathtt{0}$) the result finally combined with previous partial results via `multi`. In our example `multi` simply executes the sum of the two operands.

Therefore, the two choice goals cause the recursive predicate to iterate through all tuples of `body` and performs the tallies specified by `single` and `multi`. Thus, `aggr` is a built-in function that remains the same for every set-aggregate, while `single` and `multi` are specified in the application program. (For instance, to count the elements of a set, `single` would have to return the value: 1. If we wanted to compute a max, `single` operates as in `sum`, but `multi` must select the larger of the two elements.)

The formal semantics of program of Example 5 can therefore be defined by rewriting $a_0$ into the following rule:

$$\mathtt{a_5 : p(X, S)} \leftarrow \quad \mathtt{aggr(sum, J, X, S)}, \neg\mathtt{aggr(sum, s(J), X, \_)}.$$

which simply states that, for each $X$, the proper aggregate value is obtained for the last value of $Y$—i.e., when the two choice goals in rule $a_2$ fail due to the fact that every element in $\mathtt{body(X, Y)}$ has already been considered.

These definitions can be extended to the general case, by reducing the situation of multiple variables and multiple goals to this basic template. When there are several rules with aggregate constructs in their heads, we assume that there exists a distinct $\mathtt{aggr_j}$ for each such rule, $\mathtt{r_j}$.

The criteria for restricting the usage of aggregates in recursion follow directly from these definitions. There is no restriction in using choice, since positive programs with choice goal always have a stable model [8]. Thus, to determine the semantic well-formedness of recursive programs, we only need to be concerned with the negation introduced by rule $a_5$ above: the original program is XY-stratified with respect to aggregates, iff the program obtained by re-writing the aggregates as described above is semantically well-formed.

The if-then-else construct is the final non-monotonic logic based construct of $\mathcal{LDL}$ and $\mathcal{LDL}$++ whose formal semantics is defined through negation. The second rule in the definition of the aggregate operator `min` is viewed as a shorthand notation for the following pair of rules:

$$\mathtt{multi(min, Y1, Z1, Z)} \leftarrow \quad \mathtt{Y1 < Z1, Z = Y1}.$$
$$\mathtt{multi(min, Y1, Z1, Z)} \leftarrow \quad \neg\mathtt{(Y1 < Z1)}, \ \ \mathtt{Z = Z1}.$$

Thus, programs containing if-then-else constructs are stratified iff their expansion using negation is.

# 4 $\mathcal{LDL}$++ Implementation

XY-Stratified programs have the property that tuples generated at each iteration are only dependent on tuples generated at the last iteration. This property offers several opportunities for optimized implementation, insofar as tuples generated in iterations before the last one can be discarded, or, in some cases, they can be re-used with their old values. After considering several design tradeoffs for $\mathcal{LDL}$++, the approach described in this section was selected for the reason that it combines effective optimization with simplicity of implementation.

At compilation time, X-rules and Y-rules are reordered by the compiler and, thus, the programmer needs not write the rules in any particular order. Such reordering is always possible since the program satisfies the XY-stratification requirements.

At execution time, XY-stratified cliques are implemented by using two temporary relations for each recursive predicate. Each of these relations is used to store tuples at each alternate iteration of the saturation process. Thus for each recursive predicate `p` we keep two temporary relations $\mathtt{R_0(p)}$ and $\mathtt{R_1(p)}$.

At level 0, the exit rules and X-rules are executed once with tuples taken from and added to relations $\mathtt{R_0}$. In levels above the 0-level, both X-rules and Y-rules are executed in their primed sorting order. The recursive goals in the Y-rules can either be primed (stage argument equal to `s(J)`) or not (stage argument equal to `J`). At level 1 (and other odd levels, 3, 5, ... ) the unprimed recursive goals in Y-rules take their tuples from the relation $\mathtt{R_0}$. Primed goals in the Y-rules and recursive goals in the X-rules consume tuples from the relation $\mathtt{R_1}$. All new tuples generated by either kind of rule are stored into the $\mathtt{R_1}$ relations. At the end of an odd-level iteration, the old values in the $\mathtt{R_0}$ relation are no longer needed, and they are actually overwritten in the next step. At the even levels, 2, 4, ..., the roles of the two relations are switched: the unprimed values are taken from the $\mathtt{R_1}$ copies and the new values are written into the $\mathtt{R_0}$ copies.

In summary, the compilation of XY-Stratified programs proceeds as follows:

**Compiling XY-stratified Cliques**

**C1.** Detect the XY-cliques and recognize X-rules, Y-rules and exit rules

**C2.** Reorder the recursive X-rules and Y-rules based on the primed sorting

**C3.** Check that all exit rules share the same stage constant.

As mentioned earlier, the computation of the stage arguments is factored out from the computation of the rules. Thus the compiler simply projects out the stage argument from the recursive predicate in each rule. Then, the execution of these rules takes place as follows:

**Executing XY-stratified Cliques**

**Step 1.** The stage variable is assigned the stage constant from the exit rules.

**Step 2.** Fire the X-rules (without the stage argument) once

**Step 3.** Fire the recursive rules (without stage arguments) in the order arranged by the compiler

**Step 4.** If saturation is not reached, increase the value of the stage variable and resume from **Step 3**.

### 4.1 Copy and Deletion Rules

Toward making the implementation more efficient, we can recognize two classes of recursive rules in XY-Stratified program that lend themselves to optimization. These are the *copy* rules and the *deletion* rules—rules other than these will be referred to as regular rules.

The following rule, from of Example 1, is a *copy rule*:

$$\mathtt{r_2 : all(s(I), Y) \leftarrow \quad all(I, Y), delta(s(I), \_).}$$

The only computation required to implement such a rule, basically, is the evaluation of the `delta` goal. If this goal evaluates to false, no further computation is needed. If the `delta` goal evaluates to true, then we need to set $\mathtt{R_0^{all}}$ ($\mathtt{R_1^{all}}$) equal to $\mathtt{R_1^{all}}$ ($\mathtt{R_0^{all}}$) when we are operating in the *even (odd)* stratum. This can be achieved by simply letting the pointers for the two relations point at the same table. In general, the processing of predicate, say $q$ defined by copy rules is similar to that required by the seminaive fixpoint as follows:

- the compiler reorders the rules to ensure that the copy rule defining say the predicate $q$ precedes all the remaining rules defining $q$[3].
- As in the computation of seminaive fixpoint $q$ consists of two parts: the old tuples $\mathbf{q}$ and the new ones $\delta q$. New tuples generated by the rules following the copy rule, are added to $\delta q$ after checking they are not contained in $\mathbf{q} \cup \delta q$ (duplicate check). A shared index structure gives access to both the tuples in $\mathbf{q}$ and those in $\delta q$.
- After all the rules for $q$ have been executed $\mathbf{q}$ still contains all the values of $q(I, ...)$ while the values of $q(s(I), ...)$ must be retrieved from $\mathbf{q} \cup \delta q$. At the end of the cycle, $\mathbf{q}$ is reset to $\mathbf{q} \cup \delta q$ while $\delta q$ is reset to empty.

In general, copy rules are Y-rules, where the first goal and the head are identical except for the stage argument, and all other goals are evaluated to either true or false for the given stage argument, independent of the variables in the first goal. Thus, copy rules have the following form:

$$\begin{array}{ll} \mathtt{p(s(I), X) \leftarrow} & \mathtt{p(I, X),} \\ & \mathtt{q_1(I, Z_1), ..., q_n(I, Z_n), r_1(s(I), W_1), ..., r_m(s(I), W_m),} \\ & \mathtt{t_1(Y_1), ..., t_l(Y_l).} \end{array}$$

where $\mathtt{q_1}$, ..., $\mathtt{q_n}$, $\mathtt{r_1}$, ..., $\mathtt{r_m}$, $\mathtt{t_1}$, ..., $\mathtt{t_l}$, are predicates that always evaluate to either true or false for the given stage argument. (This is, for instance, the

[3] Observe that, in principle, at least, we could have several copy rules; but once a copy rule actually fires, the rest can be ignored.

case when the variables $\mathtt{X}$ are disjoint from those in every $\mathtt{Z_1, ..., Z_n}$, $\mathtt{W_1, ..., W_m}$ and $\mathtt{Y_1, ..., Y_l}$. A compile-time recognition of other cases is harder).

Consider now a rule having the following format:

**Example 7** *A deletion rule.*

$$\mathtt{p(s(I), X)} \leftarrow \quad \mathtt{p(I, X), \neg d(X).}$$

This rule can be implemented by deleting from $\mathtt{p}$ the tuples that satisfy $\mathtt{d}$: the cost of this operation is proportional to the cardinality of $\mathtt{d(X)}$, which in many cases is much less than that of $\mathtt{p(I, X)}$. This can be achieved with the same scheme as that employed in the execution of the copy rules; i.e. by placing the deletion rule before the regular rules, but after the copy rules [4] and then deleting the atoms specified by the negative goals in the rules. In general, a deletion rule has the following form:

$$\begin{aligned} \mathtt{p(s(I), X)} \leftarrow \quad & \mathtt{p(I, X), \neg d(A).} \\ & \mathtt{q_1(I, Z_1), ..., q_n(I, Z_n), r_1(s(I), W_1), ..., r_m(s(I), W_m)} \\ & \mathtt{t_1(Y_1), ..., t_l(Y_l).} \end{aligned}$$

where $\mathtt{A}$ is a subset of $\mathtt{X}$, and $\mathtt{q_1}$, ..., $\mathtt{q_n}$, $\mathtt{r_1}$, ..., $\mathtt{r_m}$ $\mathtt{t_1}$, ..., $\mathtt{t_l}$, are predicates that evaluate to either to true or false for the given stage argument, thus the set of variables $\mathtt{X}$ must be disjoint from those in every $\mathtt{Z_1, ..., Z_n}$, $\mathtt{W_1, ..., W_m}$ and $\mathtt{Y_1, ..., Y_l}$.

Deletion rules are more problematic than the copy rules because the tuples they delete in the old relation may be needed in later rules. Consider the following example:

**Example 8** *Problematic deletion rule.*

$$\begin{aligned} \mathtt{r_1 : p(s(I), X)} \leftarrow \quad & \mathtt{p(I, X), \neg g(X).} \\ \mathtt{r_2 : q(s(I), X)} \leftarrow \quad & \mathtt{p(I, X), X > Y, p(s(I), X).} \end{aligned}$$

Based on the primed sorting, $r_2$ must follow $r_1$ because it uses both $\mathtt{p(s(I), X)}$ and $\mathtt{p(I, X)}$. Thus, if the deletion rule is implemented as suggested above, tuples in the old relation will be clobbered even when the recursive goal $\mathtt{p(I, X)}$ is consuming from it ! In such a case, we have to execute $r_1$ like a normal Y-rule. For instance, multiple deletion rules for the same predicates must be treated as they were regular rules.

In general, the compilation of deletion rules should also be combined with the process of primed sorting of the predicate names. Let $\mathtt{p}$ be the predicate name of the head of a candidate deletion rule $r$: if there exists a primed sorting, such that $\mathtt{p(I)}$ is not used in rules after $r$, then $r$ is treated as a deletion rule. Otherwise $r$ is implemented as a regular rule.

[4] As a special case, whenever any copy rule is fired, all the deletion rules must be ignored.

## 4.2 Choice and Termination in XY-stratified Programs

An important consideration in the practical deployment of XY-stratified programs is termination. Strictly speaking, saturation occurs only when all the recursive rules fail to deliver any new value. Thus for programs containing copy rules, such as the one above, the old values of p are copied into a set of new values and the fixpoint computation would never terminate. One solution to this problem is that of Example 1, where we have added a second goal to rule $r_2$ to ensure failure. However, the $\mathcal{LDL}$++ system is also capable of detecting two common situations when the computation is stopped before the computation of the perfect model has completed.

One such situation involves the use of choice constructs in the rules using the tuples produced by the XY-stratified clique. Take for instance Floyd's algorithm of Example 6. Assume that the recursive clique is called by the following rule:

$$\texttt{dist(X,Y,C)} \leftarrow \quad \texttt{all(J,X,Y,C)}, \neg\texttt{delta(J,\_,\_)}, \texttt{choice((),(J))}.$$

The compiler recognizes that $J$ is bound to a stage argument, and that the only $J$ value of interest is the lowest value $J'$ for which the rule is first satisfied. Once the system detects a second value $J \neq J'$ it immediately terminates the computation of the XY-stratified clique.

The previous example illustrates a useful application of choice outside the XY-stratified clique. Choice is also very useful inside an XY-clique, but it can be used only with major restrictions. These follow from the fact that choice requires the memorization of old values, while the implementation of XY-stratified clique is based on only one level of stage argument being needed at any time. Therefore, *only choice goals where the stage variable J appears in the first argument of choice (the left side of the FD) are allowed in recursive rules.*[5] Even so curtailed, choice can play an important role. For instance, Prim's minimum spanning tree algorithm can be expressed as follows:

**Example 9** *Greedy Algorithms: Prim's Minimum Spanning Tree*

```
prim(0,nil,a).

leastf(s(J),least(< C >)) ←  fringe(J,X,Y,C).

prim(J,X,Y) ←  leastf(J,C),fringe(J,X,Y,C),choice((J),(X,Y)).

solved(s(I),Y) ←  solved(I,Y).
solved(J,Y) ←     prim(J,_,Y).

fringe(s(J),X,Y,C) ←  fringe(J,X,Y,C),¬prim(s(J),_,Y).
fringe(J,X,Z,C) ←     prim(J,X,Y),g(Y,Z,C),¬solved(J,Y).
```

[5]Definition 1 will thus be modified to reflect this relaxation

The meaning of the recursive rules in this example is as follows: (1) find all the least-cost arcs in fringe, (2) choose one such arc and add it to prim (3) update the list of solved nodes (i.e., nodes with an incoming arc), (4) update the fringe, (5) remove from fringe arcs entering the new node, (6) add to fringe the arcs departing from the new node added to prim, provided that they do not lead back to solved nodes.

The results produced by this program will used by a calling goal such as $\mathtt{prim}(\_, \mathtt{X}, \mathtt{Y})$ which ignores the values of the stage arguments. This enables the $\mathcal{LDL}$++ compiler to apply a second important criterion for termination. In fact if the content of recursive relations (without stage argument) at step $I+1$ is identical to that at step $I$, then it will remain identical for every $J > I$; since the value of stage argument is not used by the calling goal, the computation can be stopped. Checking equality of contents is impractical, but the following sufficient condition can be tested efficiently: During the last iteration *(i) regular rules (i.e., those that neither copy rules nor deletion rules) have produced no atoms, and (ii) deletion rules, if any, have actually not deleted any tuple.* When both conditions occurred, the relations at the next step will be identical to those at this step; thus computation can be stopped for the XY-stratified clique. [6]

In summary, we have two powerful mechanisms for ensuring termination of XY-stratified cliques via external calls: an explicit one that uses choice on the stage variable, and an implicit one that is based on the existential usage of the stage variable in the calling goal.

# 5 XY-Stratified Programs

Let us begin with a Bill of Materials application where the given relation, g(X,Y, Q), states that X uses subpart Y in quantity Q. Our application lists and counts the elementary components contained in each part. We solve the problem inductively. Our exit rule finds elementary parts (i.e., those that have no subcomponents), and aggregates them for their immediate super-parts. Then, at step $I+1$, we propagate the partial tallys of elementary components, computed at step I. These partial tallys are finally summed up by the final_count rule, outside the clique.

**Example 10** *BOM: for each part* X *counting its basic components* Y*.*

$$\begin{array}{ll} \mathtt{rgc}(0,\mathtt{X},\mathtt{Y},\mathtt{sum}(<\mathtt{Q}>)) \leftarrow & \mathtt{g}(\mathtt{X},\mathtt{Y},\mathtt{Q}), \neg\mathtt{g}(\mathtt{Y},\_,\_). \\ \mathtt{rgc}(\mathtt{I}+1,\mathtt{X},\mathtt{Y},\mathtt{sum}(<\mathtt{Q}>)) \leftarrow & \mathtt{g}(\mathtt{X},\mathtt{W},\mathtt{Q2}), \mathtt{rgc}(\mathtt{I},\mathtt{W},\mathtt{Y},\mathtt{Q1}), \mathtt{Q}=\mathtt{Q1}*\mathtt{Q2}. \end{array}$$

$$\mathtt{final\_count}(\mathtt{X1},\mathtt{Y1},\mathtt{sum2}(<(\mathtt{I},\mathtt{Q})>)) \leftarrow \quad \mathtt{rgc}(\mathtt{I},\mathtt{X1},\mathtt{Y1},\mathtt{Q}).$$

[6] In Example 9, relation prim becomes empty after all nodes have been solved and fringe is empty as well. Thus the computation stops independent on whether the compiler treats the first fringe rule as a deletion rule, or not. However, consider Floyd's algorithm of Example 2, assuming that the calling goal is $\mathtt{delta}(\_, \mathtt{X}, \mathtt{Y}, \mathtt{C})$. Here, if the first all rule is recognized as a deletion rule, computation terminates as soon as delta becomes empty.

In this XY-stratified clique, we have used `I+1` instead of $s(I)$ in the stage arguments. The two notations are obviously equivalent and the $\mathcal{LDL}$++ compiler recognizes these programs as being XY-Stratified.

The `sum2` predicate, in the non-recursive rule, adds up all the values of `Q` without eliminating duplicate values of `I`. Therefore it can be defined as follows:

```
single(sum2, (I, Q), Q).
multi(sum2, Y1, Z1, Z) ←   Z = Z1 + Y1.
```

This computation terminates after saturation is reached.

**Example 11** *Dijkstra's single-source least-cost-path algorithm.*

```
cand(nil, a, 0).
cand(s(J), Y, min(< C >)) ←   least(J, C1), cand(J, X, C1), g(X, Y, C2),
                              ¬alleast(J, Y, _),
                              if(cand(J, Y, C3) then C3 > C).
cand(s(J), Y, C) ←   cand(J, Y, C), ¬least(J, Y).
alleast(s(J), Y, C) ←   alleast(J, Y, C).
alleast(s(J), Y, C) ←   least(J, Y, C).
least(J, min(< C >)) ←   cand(J, _, C).
```

The termination behavior of this greedy program, w.r.t. an external calling goal, is similar to that of Prim's algorithm.

**Corporations and Shares.** Corporation A owns an `SP` percentage of the shares of corporation B; if it owns more than 50% of such shares then A controls B. When A controls B, then it (indirectly) owns all the shares owned by B. We need to compute all these indirect ownerships.

**Example 12** *Transitive Control of Corporations*

```
delta(0, C1, C2, SP) ←   own(C1, C2, SP).
delta(s(J), C1, C3, sum2(< (C2, SP) >)) ←   control(J, C1, C2),
                                            own(C2, C3, SP).
shares(s(J), C1, C3, SP) ←   shares(J, C1, C3, SP), ¬delta(J, C1, C3, _).
shares(J, C1, C3, SP) ←      delta(J, C1, C3, Q1),
                             if(shares(J, C1, C3, Q2)then SP = Q1 + Q2,
                             else SP = Q1).
control(J, C1, C2) ←   shares(J, C1, C2, SP), SP > 0.5.
```

In this example as well, the value of the stage argument is of no consequence outside the clique, and the special termination test previously described is applicable.

## 5.1 Magic Set Method

The magic set method, and other compilation methods designed to propagate constraints into the recursive predicates, remain applicable to XY-stratified programs. Take for instance the BOM program, Example 10. If we are only interested in finding the elementary components used by a given part, say `frame`, we can ask the following query: ?`final_count(frame, Y, Q)`. Thus, this binding is propagated through the non-recursive rule of Example 10, and the recursive predicate `rgc` is called with adornment $\mathtt{rgc}^{\mathtt{fbff}}$. This binding is propagated through the goal $\mathtt{g(X, W, Q2)}$ in the recursive rule, returning the same adornment: $\mathtt{rgc}^{\mathtt{fbff}}$. Thus the magic-set method is applied yielding:

**Example 13** *BOM: after the magic set transformation.*

$$
\begin{array}{ll}
\mathtt{m.rgc(\$Y1).} & \%\ \mathtt{\$Y1\ denotes\ the\ bound\ value\ of\ Y1} \\
\mathtt{m.rgc(W)} \leftarrow & \mathtt{m.rgc(X), g(X, W, Q2).}
\end{array}
$$

$$
\begin{array}{ll}
\mathtt{rgc(0, X, Y, sum(<Q>))} \leftarrow & \mathtt{m.rgc(X), g(X, Y, Q), \neg g(Y, \_, \_).} \\
\mathtt{rgc(I+1, X, Y, sum(<Q>))} \leftarrow & \mathtt{m.rgc(W),} \\
 & \mathtt{g(X, W, Q2), rgc(I, W, Y, Q1), Q = Q1 * Q2.}
\end{array}
$$

Observe that the application of the magic-set method adds no semantic complication. The recursive clique remains XY-stratified. Alternatively one could be interested in finding the usage of a given elementary component by different parts, and, e.g., issue the query ?`final_count(X, bolt, Q)`. This second query will be recognized, in the special class of left/right linear queries, and compiled by specializing the recursive clique with the instantiation of `X` with '`bolt`'.

# 6 Conclusions

The problem of finding efficient solutions to logic-based languages and specifications is one of the most enduring challenges of computing. Researchers in deductive databases have experienced the difficulty of this problem in dealing with non-stratified negation and aggregates, where the many elegant solutions presented in the past have not yielded the level of efficiency and generality that is required by complex real-life applications. Different remedies have been proposed to overcome this problem. An obvious remedy is that of escaping to a procedural language for operations that cannot be performed efficiently in a logic-based language. In particular, Nail-Glue provides a procedural shell that minimizes the "impedance-mismatch" problem between the procedural language and the logic-based one [15].

A more radical suggestion proposed is to abandon the minimal-model and least-fixpoint semantics of programs in favor of an operational semantics based on the concept of *inflationary fixpoint* [14]. This idea has an obvious appeal for deductive database people, insofar as (i) it represents a natural abstraction of the bottom-up computation used to implement systems such as $\mathcal{LDL}$ or Nail! (ii) it is easy for a programmer to understand and powerful in terms of expressive power (iii) it actually coincides with the minimal model/least-fixpoint semantics for monotonic programs. On the downside however, this approach repeats the past mistakes made by Prolog designers. Having opted for a top-down depth-first approach to the implementation of Horn clauses, Prolog designers elected to adopt this as THE actual semantics for the language and the basis of non-monotonic and impure constructs, such as the cut, assert and negation. The result is that Prolog is completely inadequate in dealing with classical fixpoint-oriented problems such as those discussed in this paper. Of course, the adoption of an inflationary-fixpoint as the actual semantics of a deductive database language would lead to symmetric problems—such as a loss of data independence and the ability of optimizing programs that lend themselves to a top-down execution.

The approach proposed in this paper combines the advantages of inflationary fixpoint and those of declarative logic. In fact, XY-stratified Datalog programs always compute in polynomial time and every problem of polynomial data complexity can be expressed using XY-stratified programs (assuming ordered domains [14] or using choice). Also, equivalence-preserving compilation methods, such as the magic-set method, remain applicable. On the other hand, the formal logic-based semantics enhances the uses of the language as a formal methods tools—as demonstrated e.g., by its recent use in formalizing updates and rules in active databases [25].

In terms of technology evolution, it is interesting to underscore the similarities and differences between $\mathcal{LDL}$ and $\mathcal{LDL}$++. The basic non-monotonic constructs (i.e., negation, conditionals, aggregates and choice) have remained the same and there is upward compatibility from the older system to the new system. But the power of the language has been extended while its underlying formal semantics have been simplified and unified. In $\mathcal{LDL}$, set aggregates and choice were given a separate semantic definition (e.g., choice was defined in terms of Functional Dependencies). But in $\mathcal{LDL}$++ the semantics of these constructs, which are seemingly different in terms of syntax, and operational meaning, have been all reduced to the same semantic basis—i.e., negation. Choice is formally defined using negation in [8], and set aggregates were defined in this paper using choice and negation. This latter definition solves several problems left open by previous approaches [10, 19, 13].

# References


[1] A.V. Aho, J.E. Hopcropt, and J.D. Ullmann. *The Design and analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] N. Arni, K. Ong, S. Tsur and C. Zaniolo. *The $\mathcal{LDL}$++ System: Rationale, Technology and Applications.* Submitted for publication.

[3] Chimenti, D. et al., "The $\mathcal{LDL}$ System Prototype," *IEEE Journal on Data and Knowledge Engineering*, vol. 2, no. 1, pp. 76-90, March 1990.

[4] M.P. Consens and A.O. Mendelzon. Low complexity aggregation in graphlog and datalog. In *Proceedings of the third International Conference on Database Theory*, 1990.

[5] S. Ganguly, S. Greco, and C. Zaniolo. *Minimum and Maximum Predicates in Logic Programming.* In *Proceedings of the Tenth ACM Symposium on Principles of Database Systems*, pages 154–113, 1991.

[6] S. Ganguly, S. Greco, and C. Zaniolo. *Propagation of Extrema Predicates into Recursive Logic Programs.* In *Unpublished Manuscript*, 1992.

[7] M. Gelfond and V. Lifschitz. The stable model semantics of logic programming. In *Proceedings of the Fifth Intern. Conference on Logic Programming*, pages 1070–1080, 1988.

[8] F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo. Nondeterminism in deductive databases. In *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, 1991.

[9] S. Greco, C. Zaniolo, and S. Ganguly. *Greedy by Choice.* In *Proceedings of the Eleventh ACM Symposium on Principles of Database Systems*, pages 105–163, 1992.

[10] D.B. Kemp and P.J. Stackey. Semantics of Logic Programs with Aggregates. In *Proceedings 1991 Int. Logic Programming Symposium*, pages 338–401, 1991.

[11] R. Krishnamurthy and S. Naqvi. "Non-deterministic choice in Datalog," In *Proceedings 3rd Int.Conference on Data and Knowledge Bases*, 1988.

[12] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.

[13] A. Van Gelder. The Well-Founded Semantics of Aggregation. In *Proceedings of the Eleventh ACM Symposium on Principles of Database Systems*, pages 127–138, 1992.

[14] P.G. Kolaitis and C.H. Papadimitriou, Why not negation by fixpoint?, *JCSS*, 43(1), 125-144, 1991.

[15] Phipps, G., M.A., Derr and K. A. Ross, "Glue-Nail: a Deductive Database System," *Proc. 1991 ACM–SIGMOD Conference on Management of Data*, pp. 308-317 (1991).

[16] T. Przymusinski. On the declarative and procedural semantics of stratified deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan-Kaufman, Los Altos, CA, 1988.

[17] A. Przymusinska and T. Przymusinski. Weakly Perfect Model Semantics for Logic Programs. In *Proceedings of the Fifth Intern. Conference on Logic Programming*, pages 1106–1122, 1988.

[18] S. A. Naqvi, S. Tsur *"A Logical Language for Data and Knowledge Bases"*, W. H. Freeman, 1989.

[19] K.A. Ross, and Y. Sagiv. Monotonic Aggregation in Deductive Databases. In *Proceedings of the Eleventh ACM Symposium on Principles of Database Systems*, pages 127–138, 1992.

[20] K.A. Ross, Modular Stratification and Magic Sets for Datalog Programs, In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 160–171, 1990.

[21] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 205–217, 1990.

[22] Ramakrishan, R., Srivastava, D. and Sudarshan, S., "*CORAL*: A Deductive Database Programming Language," Proc. VLDB'92 Int. Conf, pp. 238-250, 1992.

[23] Tsur S., 'Deductive Databases in Action,' *Proc. 10th, ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 205-218, 1990.

[24] Zaniolo, C., Intelligent Databases: Old Challenges and New Opportunities, *Journal of Intelligent Information Systems*, 1, 271-292 (1992).

[25] Zaniolo, C., A Unified Semantics for Deductive Databases and Active Databases, *Procs. Workshop on Rules in Database Systems*, Edinburgh, U.K., 1993.