Chapter 22

NONMONOTONIC REASONING IN LDL^{++}

Haixun Wang
Computer Science Department
University of California at Los Angeles
hxwang@cs.ucla.edu

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

Abstract Deductive database systems have made major advances on efficient support for nonmonotonic reasoning. A first generation of deductive database systems supported the notion of stratification for programs with negation and set aggregates. Stratification is simple to understand and efficient to implement but it is too restrictive; therefore, a second generation of systems seeks efficient support for more powerful semantics based on notions such as well-founded models and stable models. In this respect, a particularly powerful set of constructs is provided by the recently enhanced $\mathcal{LDL}$++ system that supports (i) monotonic user-defined aggregates, (ii) XY-stratified programs, and (iii) the nondeterministic choice constructs under stable model semantics. This integrated set of primitives supports a terse formulation and efficient implementation for complex computations, such as greedy algorithms and data mining functions, yielding levels of expressive power unmatched by other deductive database systems.

Keywords: Deductive Databases, Nonmonotonic Reasoning, Stratification, Monotonic Aggregation

1 INTRODUCTION

The $\mathcal{LDL}$ system (Chimenti et al., 1990) developed at MCC in the 80's is representative of a generation of Deductive Database Systems (Minker 1996) that sought to integrate logic-based rule languages into relational databases, in order to provide efficient and scalable support for sophisticated queries on large data sets and knowledge bases. Therefore, most of these systems

(Ramakrishnan and Ullman 1995) followed a (bottom-up) fixpoint approach for both abstract and operational semantics, rather than the (top-down) SLD-resolution of Prolog, since the former was considered more conducive to mapping into relational algebra and secondary-store implementation techniques of database systems than the stack-oriented implementation techniques of Prolog (Zaniolo et al., 1997). Indeed, this line of work led to implementation techniques, such as differential fixpoint and magic sets that are now included in most commercial database systems and SQL3 standards (Filkenstein et al., 1996); also it leads to the ability of the $\mathcal{LDL}/\mathcal{LDL}$++ system to compile rules and predicates into SQL when these are supportable on external databases. These techniques have already been documented extensively (Chimenti et al., 1990, Ramakrishnan et al., 1993, Abiteboul et al., 1995, Ramakrishnan and Ullman 1995, Zaniolo et al., 1997) and will not be further discussed here.

In this paper, we will instead discuss the nonmonotonic reasoning techniques that were developed later, largely in response to the experience of deploying these systems in real life applications. Indeed, it was found that the stratification condition obeyed by $\mathcal{LDL}$ and most systems of this generation was too restrictive and prevents the efficient expression of quintessential database queries, such as Bill of Materials (BoM) queries and the optimized transitive-closure queries needed to compute efficiently least-cost paths in a graph.

Many new notions and techniques were therefore proposed and added to deductive database systems to support negation and set aggregates in nonstratified logic programs (Ramakrishnan and Ullman 1995). In particular, $\mathcal{LDL}$ was followed by $\mathcal{LDL}$++ which, with the recent extensions implemented at UCLA (Zaniolo et al. 1998), provides a powerful and comprehensive set of nonmonotonic constructs; these extensions have been used in advanced applications such as AI planning (Brogi et al., 1997) and data mining (Bonchi et al., 1999).

The issues posed by nonmonotonicity can be illustrated by the notion of stratification (Przymusinski 1988) which, while lacking in expressive power, is rich in the other important qualities that are desirable in nonmonotonic constructs. Indeed, stratification (i) has a formal declarative semantics, (ii) has good usability, since it is easy for users to understand, (iii) is amenable to efficient implementation (using an iterated fixpoint procedure (Przymusinski 1988, Zaniolo et al., 1997)), and (iv) is applicable to set-aggregates as well as negation.

The problem of generalizing stratified programs without compromising the four properties described above proved to be a real challenge. The notions of well-founded models (Van Gelder et al., 1990), and stable models (Gelfond and Lifschitz) provide a sound basis for formal semantics; but then, there remain the issues of performance (since finding stable models is $\mathcal{NP}$-hard (Schilpf et al., 1992)), and also the issue of usability. The usability issue follows from the complexity of the notions involved, and the fact that the semantic well-formedness of a program cannot be checked at compile time, since the existence of stable models or well-founded models for a given program depends on the database content besides the rules[1].

Some Deductive Database Systems (Ramakrishnan and Ullman 1995) provide support for modular stratification (Ross 1994), which is more powerful than stratification but fail to solve the usability issue. On the other hand, the notion of XY-stratification supported in $\mathcal{LDL}$++ solves both the issue of expressive power and that of usability (Zaniolo et al., 1993).

A second line of attack, rather than attempting to solve the nonmonotonicity conundrum, tries to circumvent it by observing that in many applications of interest, e.g., greedy algorithms (Ganguly et al., 1995), aggregates are used in a monotonic way. This idea of monotone aggregation was pursued in a general setting in (Ross et al., 1997), where several interesting examples were studied in which aggregates are monotonic in lattices different from the standard set-containment lattice used for Horn clauses. Unfortunately, the lack of viable techniques to detect these special lattices (Van Gelder, 1993) prevented the implementation of this idea in actual DDB systems. This situation has changed recently with the introduction in $\mathcal{LDL}$++ of user-defined aggregates (UDAs) that support *early returns*. In fact, we discovered (Zaniolo and Wang, 1999) that UDAs that only use early returns (no final returns) are monotonic in the standard lattice of set containment; thus, they can be used freely in recursive $\mathcal{LDL}$++ programs, or SQL queries—unlike aggregates with final returns that are nonmonotonic and can only be used in XY-stratified programs.

Our presentation is organized as follows. In the next section, we introduce the construct of choice, and focus on programs with negation that have nondeterministic but monotonic stable-model semantics. Then, in Section 3, we use choice to define the semantics of aggregates and provide syntactic characterization for monotonic and nonmonotonic user-defined aggregates. Finally, in Section 4, we describe the notion of XY-stratification for recursive rules containing negation and nonmonotonic aggregates.

2 MONOTONIC NONDETERMINISM

Say that our database contains relations `student(Name, Major, Year)`, and `professor(Name, Major)`. In fact, let us take a toy example that only has the following facts:

```
student('JimBlack', ee, senior).        professor(ohm, ee).
                                        professor(bell, ee).
```

Now, the rule is that the major of a student must match his/her advisor's major area of specialization. Then, eligible advisors can be computed as follows:

```
elig_adv(S, P) ←  student(S, Majr, Year), professor(P, Majr).
```

This yields

[1]Unlike Prolog where the programmer sees both facts and rules, only the rules are normally available to the programmer of a deductive database system (along with the database schema that contains type information for the tables but not their content).

$$\texttt{elig\_adv}(\texttt{'JimBlack'}, \texttt{ohm}).$$
$$\texttt{elig\_adv}(\texttt{'JimBlack'}, \texttt{bell}).$$

But, since a student can only have one advisor, the goal $\texttt{choice}((\texttt{S}), (\texttt{P}))$ must be added to our rule to force the selection of a unique advisor for each student:

Example 1 Computation of unique advisors by a choice rule

$$\begin{aligned} \texttt{actual\_adv}(\texttt{S}, \texttt{P}) \leftarrow\ & \texttt{student}(\texttt{S}, \texttt{Majr}, \texttt{Yr}),\ \texttt{professor}(\texttt{P}, \texttt{Majr}), \\ & \texttt{choice}((\texttt{S}), (\texttt{P})). \end{aligned}$$

The goal $\texttt{choice}((\texttt{S}), (\texttt{P}))$ can also be viewed as enforcing a *functional dependency* (FD) $\texttt{S} \rightarrow \texttt{P}$ on the results produced by the rule; thus, in $\texttt{actual\_adv}$, the second column (professor name) is functionally dependent on the first one (student name). As in FDs, $\texttt{S}$ and $\texttt{P}$ will be called the left side and the right side of this choice goal. The right side of a choice goal cannot be empty, but its left side can be empty, denoting that all tuples produced must share the same values for the right side attributes.

The result of this rule is *nondeterministic*: it can either return a singleton relation containing the tuple $(\texttt{'JimBlack'}, \texttt{ohm})$, or one containing the tuple $(\texttt{'JimBlack'}, \texttt{bell})$.

A program where the rules contain choice goals is called a *choice program*. The semantics of a choice program P can be defined by transforming P into a program with negation, $foe(P)$, called the *first order equivalent* of P. Now, $foe(P)$ exhibits a multiplicity of stable models, each obeying the FDs defined by the choice goals; each such stable model corresponds to an alternative set of answers for P and is called a *choice model* for P. The first order equivalent of Example 1 is as follows:

Example 2 *The first order equivalent for Example 1*

$$\begin{aligned} \texttt{actual\_adv}(\texttt{S}, \texttt{P}) \leftarrow\ & \texttt{student}(\texttt{S}, \texttt{Majr}, \texttt{Yr}),\ \texttt{professor}(\texttt{P}, \texttt{Majr}), \\ & \texttt{chosen}(\texttt{S}, \texttt{P}). \\ \texttt{chosen}(\texttt{S}, \texttt{P}) \leftarrow\ & \texttt{student}(\texttt{S}, \texttt{Majr}, \texttt{Yr}),\ \texttt{professor}(\texttt{P}, \texttt{Majr}), \\ & \neg\texttt{diffChoice}(\texttt{S}, \texttt{P}). \\ \texttt{diffChoice}(\texttt{S}, \texttt{P}) \leftarrow\ & \texttt{chosen}(\texttt{S}, \texttt{P}'),\ \texttt{P} \neq \texttt{P}'. \end{aligned}$$

This can be read as a statement that a professor will be assigned to a student whenever a different professor has not been assigned to the same student. In general, $foe(P)$ is defined as follows:

Definition 1 Let P denote a program with choice rules: its first order equivalent $foe(P)$ is obtained by the following transformation. Consider a choice rule r in P:

$$r : A \leftarrow B(Z),\ choice((X_1), (Y_1)),\ \ldots,\ choice((X_k), (Y_k)).$$

where,

(i) $B(Z)$ denotes the conjunction of all the goals of r that are not choice goals, and

(ii) X_i, Y_i, Z, $1 \leq i \leq k$, denote vectors of variables occurring in the body of r such that $X_i \cap Y_i = \emptyset$ and $X_i, Y_i \subseteq Z$.

Then, $foe(P)$ is constructed P as follows:

1. Replace r with a rule r' obtained by substituting the choice goals with the atom $chosen_r(W)$:

$$r' : A \leftarrow B(Z),\ chosen_r(W).$$

   where $W \subseteq Z$ is the list of all variables appearing in choice goals, i.e., $W = \bigcup_{1 \leq j \leq k} X_j \cup Y_j$.

2. Add the new rule

$$chosen_r(W) \leftarrow B(Z),\ \neg diffChoice_r(W).$$

3. For each choice atom $choice((X_i),(Y_i))$ $(1 \leq i \leq k)$, add the new rule

$$diffChoice_r(W) \leftarrow chosen_r(W'),\ Y_i \neq Y_i'.$$

   where (i) the list of variables W' is derived from W by replacing each $A \notin X_i$ with a new variable A' (i.e., by priming those variables), and (ii) $Y_i \neq Y_i'$ is true if $A \neq A'$, for some variable $A \in Y_i$ and its primed counterpart $A' \in Y_i'$.

Properties of Choice Models

Theorem 1 Let P be a positive program with choice rules. Then the following properties hold (Giannotti et al., 1991):

- $foe(P)$ has one or more total stable models.
- The *chosen* atoms in each stable model of $foe(P)$ obey the FDs defined by the choice goals.

Observe that the $foe(P)$ of a program with choice does not have total well-founded models; in fact, for our Example 1, the well-founded model yields undefined values for advisors. Therefore, the choice can express nondeterministic semantics, which can be also expressed by stable models but not well-founded models. On the other hand, the choice model avoids the exponential complexity which is normally encountered in programs with nondeterministic semantics, such as disjunctive logic programs (Eiter et al., 2000). Indeed, the computation of stable models is $\mathcal{NP}$-hard (Schilpf et al., 1992), but the computation of choice models for positive programs can be performed in polynomial time with respect to the size of the database. This, basically, is due to the monotonic nature of the choice construct that yields a simple fixpoint computation for programs with choice (Giannotti et al., 2000). We show next that the use of choice rules in positive programs preserve their monotonic properties. A program P can be viewed as consisting of two separate components: an extensional component (i.e., the database facts), denoted $edb(P)$, and an intensional one (i.e., the rules), denoted $idb(P)$. Then, a positive choice program defines a monotonic multi-valued mapping from $edb(P)$ to $idb(P)$, as per the following theorem proven in (Giannotti et al., 2000):

Theorem 2 Let P and P' be two positive choice programs where $idb(P') = idb(P)$ and $edb(P') \supseteq edb(P)$. Then, if M is a choice model for P, then, there exists a choice model M' for P' such that $M' \supseteq M$.

Thus positive choice programs represent a class of logic programs that are very well-behaved from both semantic and computational aspects. The same can be said for choice programs with stratified negation that are defined next.

Definition 2 Let P be a choice program with negated goals. Then, P is said to be stratified when the program obtained from P by removing its choice goals is stratified.

The stable models for a stratified choice program P can be computed using an *iterated choice fixpoint* procedure that directly extends the iterated fixpoint procedure for programs with stratified negation (Przymusinski 1988, Zaniolo et al., 1997); this is summarized next. Let P_i, denote the rules of P (whose head is) in stratum i, and let $P_i{}^*$ be the union of P_j, $j \leq i$. Now, if M_i is a stable model for $P_i{}^*$, then every stable model for $M_i \cup P_{i+1}$ is a stable model for the program P^*_{i+1}. Therefore, the stable models of stratified choice programs can be computed by modifying the iterated fixpoint procedure used for stratified programs so that choice models (rather than the least models) are computed for strata containing choice rules (Giannotti et al., 1999).

Programs with Choice

Choice significantly extends the power of Datalog by allowing *nondeterministic* queries; moreover using choice in Datalog we can express important *deterministic* queries that cannot be expressed in stratified Datalog (Giannotti et al., 2000). This can be illustrated by the following choice program that places the elements of a relation `d(Y)` into a chain, thus establishing a random total order on these elements.

Example 3 Linear sequencing of the elements of a set. The elements of the set are stored by means of facts of the form `d(Y)`.

```
chain(nil,nil).
chain(X,Y) ←    chain(_,X), d(Y),
                choice((X),(Y)), choice((Y),(X)).
```

Here `chain(nil,nil)` is the root of a chain linking all the elements of `d(Y)`— thus inducing a total order on elements of `d`. Because of the ability of choice programs to order the elements of a set, Datalog with choice and stratified negation is P-time complete and can, for instance, express the parity query that determines if a relation has an even number of elements (Abiteboul et al., 1995). This query cannot be expressed in Datalog with stratified negation unless we assume that the underlying universe is totally ordered—an assumption that violates the data independence principle of *genericity* (Abiteboul et al., 1995).

The expressive power of choice was studied in (Giannotti et al., 2000) where it was shown that it is more powerful than other nondeterministic constructs, such as the witness operator (Abiteboul et al., 1995) and the original version of choice proposed in (Krishnamurthy and Naqvi, 1988); e.g., these other operators can neither order domains nor express the parity query (Giannotti et al., 1991).

Example 4 Rooted spanning tree. We are given an undirected graph where an edge joining two nodes, say x and y, is represented by the pair $g(x, y)$ and $g(y, x)$. Then, a spanning tree in this graph, starting from the source node a, can be constructed by the following program:

$$
\begin{array}{ll}
\texttt{st(root, a).} & \\
\texttt{st(X, Y)} \leftarrow & \texttt{st(\_, X),}\ \texttt{g(X, Y),}\ \texttt{Y} \neq \texttt{a,}\ \texttt{Y} \neq \texttt{X,} \\
& \texttt{choice((Y), (X)).}
\end{array}
$$

To illustrate the presence of multiple total choice models for this program, take a simple graph consisting of the following arcs:

$$
\begin{array}{ll}
\texttt{g(a, b).} & \texttt{g(b, a).} \\
\texttt{g(b, c).} & \texttt{g(c, b).} \\
\texttt{g(a, c).} & \texttt{g(c, a).}
\end{array}
$$

After the exit rule adds st(root, a), the recursive rule could add st(a, b) and st(a, c) along with the two tuples chosen(a, b) and chosen(a, c) in the chosen table. No further arc can be added after those, since the addition of st(b, c) or st(c, b) would violate the FD that follows from choice((Y), (X)) enforced through the chosen table. However, since st(root, a) was produced by the first rule (the exit rule), rather than the second rule (the recursive choice rule), the table chosen contains no tuple with second argument equal to the source node a. Therefore, to avoid the addition of st(c, a) or st(b, a), the goal $\texttt{Y} \neq \texttt{a}$ was added to the recursive rule.

By examining all possible solutions, we conclude that this program has three different choice models, for which we list only the st-atoms, below:

1. st(a, b), st(b, c).
2. st(a, b), st(a, c).
3. st(a, c), st(c, b).

3 AGGREGATES IN LOGIC

The expressive power of choice can be put to use to provide an inductive definition of aggregates in logic. Say for instance that we want to define the aggregate **avg** that returns the average of Y-values that satisfy d(Y). Using the notation first used in $\mathcal{LDL}$ (Chimenti et al., 1990), and then in *CORAL* (Ramakrishnan et al., 1993) and $\mathcal{LDL}$++, this computation can be specified by the following rule:

$$\texttt{p(avg}\langle\texttt{Y}\rangle\texttt{)} \leftarrow \texttt{d(Y).}$$

A logic-based equivalent for this rule is

$$\mathtt{p(Y)} \leftarrow \quad \mathtt{results(avg, Y).}$$

where $\mathtt{results(avg, Y)}$ is derived from $\mathtt{d(Y)}$ by (i) the chain rules, (ii) the `cagr` rules and (iii) the `return` rules. The chain rules are those of Example 3, that place the elements of $\mathtt{d(Y)}$ into an order-inducing chain. Then, the `cagr` rules perform the inductive computation by calling the `single` and `multi` rules as follows:

$$\begin{aligned}
\mathtt{cagr(AgName, Y, New)} \leftarrow \quad & \mathtt{chain(nil, Y), Y \neq nil, single(avg, Y, New).} \\
\mathtt{cagr(AgName, Y2, New)} \leftarrow \quad & \mathtt{chain(Y1, Y2), cagr(AgName, Y1, Old),} \\
& \mathtt{multi(AgName, Y2, Old, New).}
\end{aligned}$$

Thus, the `cagr` rules are used to memorize the previous results, and to apply (i) `single` to the first element of $\mathtt{d(Y)}$ (i.e., for the pattern $\mathtt{chain(nil, Y)}$) and (ii) `multi` to the successive elements. The return rules are as follows:

$$\begin{aligned}
\mathtt{results(AgName, Yield)} \leftarrow \quad & \mathtt{chain(Y1, Y2), cagr(AgName, Y1, Old),} \\
& \mathtt{ereturn(AgName, Y2, Old, Yield).} \\
\mathtt{results(AgName, Yield)} \leftarrow \quad & \mathtt{chain(X, Y), \neg chain(Y, _),} \\
& \mathtt{cagr(AgName, Y, Old),} \\
& \mathtt{freturn(AgName, Y, Old, Yield).}
\end{aligned}$$

Therefore we first compute `chain`, and then `cagr` that applies the `single` and `multi` to every element in the chain. Concurrently, the first `results` rule produces all the results that can be generated by the application of `ereturn` to each element of the chain. The final returns are however computed by the second `results` rule by the application of `freturn` once the last element in the chain (i.e., the element without successors) is detected. This is the only rule using negation; in the absence of `freturn` this rule can be removed yielding a positive choice program that is monotonic by Theorem 2. Thus every aggregate with only early returns is *monotonic with respect to set-containment* and can be used freely in recursive rules.

To define a new aggregate, the user must write the `single`, `multi`, `ereturn` and `freturn` rules; the remaining rules are built in the system. For instance, the computation of `avg` requires the accumulation of the count and the sum of the elements visited so far. Thus $\mathtt{single(avg, \ldots)}$ initializes the count to `1` and the sum to the value of the actual element `Y`, while $\mathtt{multi(avg, \ldots)}$increases the count by one and the sum by the new value. Therefore, for `avg` the user can write the following rules:

$$\begin{aligned}
& \mathtt{single(avg, Y, cs(1, Y)).} \\
& \mathtt{multi(avg, Y, cs(Cnt, Sum), cs(Cnt1, Sum1))} \leftarrow \\
& \qquad\qquad\qquad\qquad \mathtt{Cnt1 = Cnt + 1, Sum1 = Sum + Y.}
\end{aligned}$$

Then, the user writes the `freturn` rule that upon visiting the final element in $\mathtt{d(Y)}$ produces the ratio of sum over count, as follows:

$$\mathtt{freturn(avg, Y, cs(Cnt, Sum), Val)} \leftarrow \quad \mathtt{Val = Sum/Cnt.}$$

To compute the average salary of employees grouped by `Dno`, the user can write:

$$\mathtt{avgsal(Dno, avg\langle Sal\rangle)} \leftarrow \quad \mathtt{emp(Eno, Sal, Dno).}$$

Each aggregate is defined by its own single, multi, ereturn and freturn rules, where the unique aggregate name is used as the first argument in the head of these rules to eliminate interference between the rules defining different aggregates. Also, for the general situation where there are group-by attributes, such as `Dno` in the rule above, then these must be added as additional arguments to the predicates of the equivalent rules (Zaniolo and Wang, 1999).

Applications of User Defined Aggregates (UDAs)

From the above discussion it follows that traditional SQL aggregates can be characterized as shorthand for logic programs with negation under stable model semantics. As shown below, the same is true for new UDAs. For instance, say that from a set of pairs such as $\mathtt{(Name, YearOfBirth)}$ as input, we want to return the `Name` of the youngest person (i.e., the person born in the latest year). This computation, which cannot be expressed directly as an aggregate in SQL, is easy to express by the UDA `youngest`, below (in $\mathcal{LDL}$++, a vector of n arguments $(\mathtt{X}_1, \ldots, \mathtt{X}_n)$ is basically treated as an n-argument function with a default name).

$$\begin{array}{ll}
\mathtt{single(youngest, (N, Y), (N, Y)).} & \\
\mathtt{multi(youngest,\ (N, Y), (N1, Y1), (N, Y))} \leftarrow & \mathtt{Y \geq Y1.} \\
\mathtt{multi(youngest,\ (N, Y), (N1, Y1), (N1, Y1))} \leftarrow & \mathtt{Y \leq Y1.} \\
\mathtt{freturn(youngest, (N, Y), (N1, Y1), N1).} &
\end{array}$$

User-defined aggregates provide a simple solution to a number of complex problems in deductive databases; due to space limitations, we consider here few simple examples, and more examples can be found in (Zaniolo et al. 1998).

The power of UDAs is significantly enhanced by *early returns*. A first important application of early returns is online aggregation (Hellerstein et al., 1997). This can be used to provide good estimates for the value of an aggregate long before the whole data set is visited. For instance, say that we want to see the average value obtained so far every 100 elements. Then, we can add the following `ereturn`:

$$\begin{array}{l}
\mathtt{ereturn(avg, X, (Sum, Count), Avg)} \leftarrow \\
\qquad\qquad\qquad \mathtt{Count\ mod\ 100 = 0, Avg = Sum/Count.}
\end{array}$$

As second example, let us consider the well-known problem of coalescing after temporal projection in temporal databases (Zaniolo et al., 1997). For instance in Example 5, below, after projecting out from the employee relation the salary column, we might have a situation where the same `Eno` appears in tuples where their valid-time intervals overlap; then these intervals must be coalesced. Here, we use closed intervals represented by the pair (`From`, `To`) where `From` is the start-time, and `To` is the end-time. Under the assumption that tuples are sorted by increasing start-time, then we can use a special `coales` aggregate to perform the task in one pass through the data.

Example 5 *Coalescing overlapping intervals sorted by start time.*

$$\mathtt{empProj(Eno, coales\langle(From, To)\rangle)} \leftarrow \quad \mathtt{emp(Eno, \_, \_, (From, To))}.$$

$$\begin{array}{ll}
\mathtt{single(coales, (Frm, To), (Frm, To)).} & \\
\mathtt{multi(coales, (Nfr, Nto), (Cfr, Cto), (Cfr, Nto))} \leftarrow & \\
\qquad\qquad\qquad\qquad \mathtt{Nfr <= Cto, Nto > Cto.} & \\
\mathtt{multi(coales, (Nfr, Nto), (Cfr, Cto), (Cfr, Cto))} \leftarrow & \mathtt{Nto <= Cto.} \\
\mathtt{multi(coales, (Nfr, Nto), (Cfr, Cto), (Cfr, Cto))} \leftarrow & \mathtt{Cto < Nfr.}
\end{array}$$

$$\begin{array}{ll}
\mathtt{ereturn(coales, (Nfr, Nto), (Cfr, Cto), (Cfr, Cto))} \leftarrow & \mathtt{Cto < Nfr.} \\
\mathtt{freturn(coales, _, LastInt, LastInt).} &
\end{array}$$

In the multi rules, when the new interval (`Nfr`, `Nto`) overlaps the current interval (`Cfr`, `Cto`) (i.e., when `Nfr` $\leq$ `Cto` given that `Cfr` $\leq$ `Nfr` by the assumption that input intervals are ordered by their start time), then we coalesce the two into an interval that begins in `Cfr` and ends with the larger of `Nto` and `Cto`. Otherwise, the current interval is returned while the new interval becomes the current one.

A rule r whose head contains aggregates is called an *aggregate rule*. A program P is said to be stratified w.r.t. aggregates when for each aggregate rule r in P, the stratum of its head predicate is strictly higher than the stratum of each body predicate. The previous program is stratified w.r.t. `coales` which is nonmonotonic since it uses both early returns and final returns.

Therefore, nonmonotonic UDAs in stratified programs are very useful in many applications. However, stratified programs are too restrictive for many advanced applications. A first solution to this problem consists in using a form of locally stratified programs called XY-stratified programs, which will be discussed in Section 3. The second solution consists in using monotonic aggregates that eliminate the need for stratification all together. As discussed in the next section, monotonic aggregates can be used freely in recursion, to express many important algorithms (Wang and Zaniolo, 2000).

Monotone Aggregation

We next define a continuous count that returns the current count after each new element (thus, it does not have a freturn since that would be redundant).

$$\begin{array}{ll}
\texttt{single(mcount, Y, 1).} & \\
\texttt{multi(mcount, Y, Old, New)} \leftarrow & \texttt{New} = \texttt{Old} + \texttt{1}. \\
\texttt{ereturn(mcount, Y, Old, New)} \leftarrow & \texttt{New} = \texttt{Old} + \texttt{1}.
\end{array}$$

The next two examples are derived from (Ross et al., 1997).

Join the Party. Some people will come to the party no matter what, and their names are stored in a `sure(Person)` relation. But others will join only after they know that at least $K = 3$ of their friends will be there. Here, `friend(P, F)` denotes that F is P's friend.

$$\begin{array}{ll}
\texttt{willcome(P)} \leftarrow & \texttt{sure(P)}. \\
\texttt{willcome(P)} \leftarrow & \texttt{c_friends(P, K)}, \texttt{K} \geq \texttt{3}. \\
\texttt{c_friends(P, mcount}\langle\texttt{F}\rangle\texttt{)} \leftarrow & \texttt{willcome(F), friend(P, F)}.
\end{array}$$

Consider now a computation of these rules on the following database.

```
friend(jerry, mark).    sure(mark).
friend(penny, mark).    sure(tom).
friend(jerry, jane).    sure(jane).
friend(penny, jane).
friend(jerry, penny).
friend(penny, tom).
```

Then, the basic semi-naive computation yields:

`willcome(mark), willcome(tom), willcome(jane),`

`c_friends(jerry, 1), c_friends(penny, 1), c_friends(jerry, 2),`

`c_friends(penny, 2), c_friends(penny, 3), willcome(penny),`

`c_friends(jerry, 3), willcome(jerry).`

This example illustrates how the standard semi-naive computation can be applied to queries containing monotone user-defined aggregates. Another interesting example is transitive ownership and control of corporations.

Company Control. Say that `owns(C1, C2, Per)` denotes the percentage of shares that corporation `C1` owns of corporation `C2`. Then, `C1` controls `C2` if it owns more than, say, 50% of its shares. In general, to decide whether `C1` controls `C3` we must also add the shares owned by corporations such as `C2` that are controlled by `C1`. This yields the transitive control rules defined with the help of a continuous sum aggregate that returns the partial sum for each new element:

$$\begin{array}{ll} \texttt{control(C,C)} \leftarrow & \texttt{owns(C,\_,\_)}. \\ \texttt{control(Onr,C)} \leftarrow & \texttt{towns(Onr,C,Per)}, \texttt{Per} > 50. \\ \texttt{towns(Onr,C2,msum}\langle\texttt{Per}\rangle\texttt{)} \leftarrow & \texttt{control(Onr,C1)}, \texttt{owns(C1,C2,Per)}. \end{array}$$

$$\begin{array}{ll} \texttt{single(msum,Y,Y)}. & \\ \texttt{multi(msum,Y,Old,New)} \leftarrow & \texttt{New} = \texttt{Old} + \texttt{Y}. \\ \texttt{ereturn(msum,Y,Old,New)} \leftarrow & \texttt{New} = \texttt{Old} + \texttt{Y}. \end{array}$$

Thus, every company controls itself, and a company `C1` that has transitive ownership of more than 50% of `C2`'s shares controls `C2` . In the last rule, `towns` computes transitive ownership with the help of `msum` that adds up the shares of controlling companies. Observe that any pair (`Onr`, `C2`) is added at most once to `control`, thus the contribution of `C1` to `Onr`'s transitive ownership of `C2` is only accounted once.

Bill-of-Materials (BoM) Applications. BoM applications represent an important application area that requires aggregates in recursive rules. Say, for instance that `assembly(P1,P2,QT)` denotes that `P1` contains part `P2` in quantity `QT`. We also have elementary parts described by the relation `basic_part(Part,Price)`. Then, the following program computes the cost of a part as the sum of the cost of the basic parts it contains.

$$\begin{array}{l} \texttt{part\_cost(Part,0,Cst)} \leftarrow \qquad \texttt{basic\_part(Part,Cst)}. \\ \texttt{part\_cost(Part,mcount}\langle\texttt{Sb}\rangle\texttt{,msum}\langle\texttt{MCst}\rangle\texttt{)} \leftarrow \\ \qquad\qquad \texttt{part\_cost(Sb,ChC,Cst)}, \texttt{prolfc(Sb,ChC)}, \\ \qquad\qquad \texttt{assembly(Part,Sb,Mult)}, \texttt{MCst} = \texttt{Cst} * \texttt{Mult}. \end{array}$$

Thus, the key condition in the body of the second rule is that a subpart `Sb` is counted in `part_cost` only when all `Sb`'s children have been counted. This occurs when the number of `Sb`'s children counted so far by `mcount` is equal to the out-degree of this node in the graph representing `assembly`. This number is kept in the prolificacy table, `prolfc(Part,ChC)`, which can be computed as follows:

$$\begin{array}{ll} \texttt{prolfc(P1,count}\langle\texttt{P2}\rangle\texttt{)} \leftarrow & \texttt{assembly(P1,P2,\_)}. \\ \texttt{prolfc(P1,0)} \leftarrow & \texttt{basic\_part(P1,\_)}. \end{array}$$

4 BEYOND STRATIFICATION

Several deductive database systems have addressed the need to go beyond stratification by supporting the notion of modular stratification (Ross 1994). Unfortunately, this approach suffers from poor usability, since the existence of a modular stratification for a program can depend on its extensional information (i.e., its fact base) and, in general, cannot be checked without executing the program. This should be contrasted with the concept of basic stratification, which instead provides a simple criterion for the programmer to follow and for the compiler to use when validating the program and optimizing its execution. Therefore, $\mathcal{LDL}$++ uses the notion of XY-stratified programs that preserves the compilability and usability benefits of stratified programs while achieving the expressive power of well-founded models (Kemp et al., 1995). XY-stratified programs are locally stratified explicitly by a temporal arguments: i.e., they can be viewed as Datalog$_{1S}$ programs, which are known to provide a powerful tool for temporal reasoning (Baudinet et al., 1994, Zaniolo et al., 1997), or Statelog programs that were used to model active databases (Lausen et al., 1998). The deductive database system Aditi (Vaghanin et al., 1994, Kemp and Ramamohanarao, 1998) also supports the closely related concept of explicitly locally stratified programs, which where shown to be as powerful as well-founded models since they can express their alternating fixpoint computation (Kemp et al., 1995).

For instance, the ancestors of `marc`, with the number of generations that separates them from `marc`, can be computed using the following program which models the differential fixpoint computation:

Example 6 *Computing ancestors of Marc and their remoteness from Marc using differential fixpoint.*

$$
\begin{array}{ll}
r_1 : \texttt{delta\_anc(0, marc)}. & \\
r_2 : \texttt{delta\_anc(J+1, Y)} \leftarrow & \texttt{delta\_anc(J, X), parent(Y, X)}, \\
 & \neg\texttt{all\_anc(J, Y)}. \\
r_3 : \texttt{all\_anc(J+1, X)} \leftarrow & \texttt{all\_anc(J, X)}. \\
r_4 : \texttt{all\_anc(J, X)} \leftarrow & \texttt{delta\_anc(J, X)}.
\end{array}
$$

This program is locally stratified by the first arguments in `delta_anc` and `all_anc` that serve as temporal arguments (thus $+1$ is a postfix successor function symbol, much in the same way as $s(J)$ denotes the successor of J in Datalog$_{1S}$ (Zaniolo et al., 1997)). The zero stratum consists of atoms of nonrecursive predicates such as `parent` and of atoms that unify with `all_anc(0,X)` or `delta_anc(0,X)`, where `X` can be any constant in the universe. The k^{th} stratum consists of atoms of the form `all_anc(k,X)`, `delta_anc(k,X)`. This program is locally stratified (Przymusinski 1988) since the heads of recursive rules belong to strata that are one above those of their goals. Alternatively, we can view this program as a compact representation for the stratified programs obtained by instantiating the temporal argument to integers that are then viewed as part of the predicate names.

Also observe that the temporal arguments in every rule are either the same as those in the head of the rule or one less. Then, there are two kinds of rules in our example: (i) an X-rule (i.e., a horizontal rule) where the temporal argument in each goal is the same as that in the head, (ii) a Y-rule (i.e., a vertical rule) where the temporal argument in some goal is one less than that in the head. Formally, let P be a set of rules defining mutually recursive predicates, where each recursive predicate has a distinguished temporal argument and every rule in P is either an X-rule or a Y-rule. Then, P will be said to be an XY-program. For instance, the program in Example 6 is an XY-program, where r_4 and r_1 are X-rules, while r_2 and r_3 are Y-rules.

A simple test can now be used to decide whether an XY-program P is locally stratified. The test begins by labeling all the head predicates in P with the prefix 'new'. Then, the body predicates with the same temporal argument as the head are labeled with the prefix 'new', and the others are labeled with the prefix 'old'. Finally, the temporal arguments are dropped from the program. The resulting program is called the *bistate version* of P and is denoted P_{bis}.

Example 7 *The bistate version of the program in Example 6*

$$
\begin{array}{ll}
\texttt{new\_delta\_anc(marc)}. & \\
\texttt{new\_delta\_anc(Y)} \leftarrow & \texttt{old\_delta\_anc(X)}, \texttt{parent(Y,X)}, \\
 & \neg\texttt{old\_all\_anc(Y)}. \\
\texttt{new\_all\_anc(X)} \leftarrow & \texttt{new\_delta\_anc(X)}. \\
\texttt{new\_all\_anc(X)} \leftarrow & \texttt{old\_all\_anc(X)}.
\end{array}
$$

Now we have that (Zaniolo et al., 1993, Zaniolo et al., 1997):

Definition 3 Let P be an XY-program. P is said to be XY-stratified when P_{bis} is a stratified program.

Theorem 3 Let P be an XY-stratified program. Then P is locally stratified.

The program of Example 7 is stratified with the following strata: $\texttt{S}_0 = \{\texttt{parent}, \texttt{old\_all\_anc}, \texttt{old\_delta\_anc}\}$, $\texttt{S}_1 = \{\texttt{new\_delta\_anc}\}$, and $\texttt{S}_2 = \{\texttt{new\_all\_anc}\}$. Thus, the program in Example 6 is locally stratified.

For an XY-stratified program P, the iterated fixpoint of Algorithm (Zaniolo et al., 1993) becomes quite simple; basically it reduces to a repeated computation over the stratified program P_{bis}. However, since the temporal arguments have been removed from this program, we need to

1. store the temporal argument as an external fact $\texttt{counter(T)}$,
2. add a new goal $\texttt{counter}(\texttt{I}_r)$ to each exit rule r in P_{bis}, where $\texttt{I}_r$ is the variable from the temporal arguments of the original rule r, and
3. For each recursive predicate $\texttt{q}$ add the rule:

$$\texttt{q(J,X)} \leftarrow \quad \texttt{new\_q(X)}, \texttt{counter(J)}.$$

The program so constructed will be called the *synchronized* bistate version of P, denoted *syncbi*(P). For instance, to obtain the synchronized version of the program in Example 7, we need to change the first rule to

$$\texttt{new\_delta\_anc(marc)} \leftarrow \quad \texttt{counter(0)}.$$

since the temporal argument in the original exit rule was the constant 0. Then, we must add the following rules:

$$
\begin{array}{ll}
\texttt{delta\_anc(J,X)} \leftarrow & \texttt{new\_delta\_anc(X)}, \texttt{counter(J)}. \\
\texttt{all\_anc(J,X)} \leftarrow & \texttt{new\_all\_anc(X)}, \texttt{counter(J)}.
\end{array}
$$

Then, the iterated fixpoint computation for an XY-stratified program can be implemented by the following procedure:

Procedure 4 *Computing a stable model of an XY-stratified program P Add the fact* `counter(0)`. *Then forever repeat the following two steps;*

1. *Compute the stable model of syncbi(P).*
2. *For each recursive predicate* `q`*, replace* `old_q` *with* `new_q`*, computed in the previous step. Then, increase the value of* `counter` *by one.*

Since *syncbi*(*P*) is stratified, we can then use the iterated fixpoint computation to compute its stable model.

XY-stratified programs have unique computational advantages that are exploited in the $\mathcal{LDL}$++ implementation. For instance, the replacement of `old_q` with `new_q` described in the last step of the procedure becomes a zero-cost operation when it is properly implemented (e.g., by switching the pointers to the relations). A second improvement concerns *copy rules*, such as the last rule in Example 6. For instance r_3 in Example 6 is a copy rule that copies the new values of `all_anc` from its old values. Observe that the body and the head of this rule are identical, except for the prefixes `new` or `old`, in its bistate version (Example 7). Thus, in order to compute `new_all_anc`, we first execute the copy rule by simply setting the pointer to `new_all_anc` to point to `old_all_anc`—a zero-cost operation. Rule r_4 that adds tuples to `new_all_anc` must be executed after r_3.

In writing XY-stratified programs, the user must also be concerned with termination conditions, since e.g., rule such as r_3 in Example 6 will keep generating the same results even after `delta` becomes empty. One solution to this problem is to add the goal `delta_anc(J, _)` to rule r_3; once no new arcs are found, then because of the negated goal in the second rule, there is no `new_delta_anc(J, _)` atom and the computation stops. Alternatively, these recursive rules might be called by a goal such as `delta_anc(J, Y)` in which case the $\mathcal{LDL}$++ system will stop as soon as no result is produced for some `J` value.

The next example solves the coalescing problem of Example 5 without assuming that tuples are sorted by their start-time. Therefore, we use the XY-stratified program of Example 8, which iterates over two basic computation steps. The first step is defined by the `overlap` rule. This determines pairs of distinct overlapping intervals, where the first interval contains the start of the second interval. The second step consists of deriving a new interval that begins at the start of the first interval, and ends at the later of the two endpoints. Finally, a rule `final_e_hist` returns the intervals that do not overlap other intervals (after eliminating the temporal argument).

Example 8 *Coalescing overlapping periods into maximal periods after a projection*

```
e_hist(0, Eno, Frm, To) ←    emp_dep_sal(0, Eno, _, _, Frm, To).
overlap(J + 1, Eno, Frm1, To1, Frm2, To2) ←
                             e_hist(J, Eno, Frm1, To1),
                             e_hist(J, Eno, Frm2, To2),
                             Frm1 ≤ Frm2, Frm2 ≤ To1,
                             distinct(Frm1, To1, Frm2, To2).
e_hist(J, Eno, Frm1, To) ←   overlap(J, Eno, Frm1, To1, Frm2, To2),
                             select_larger(To1, To2, To).
```

```
final_e_hist(J + 1, Eno, Frm, To) ←  e_hist(J, Eno, Frm, To),
                                     ¬overlap(J + 1, Eno, Frm, To, _, _).

          distinct(Frm1, To1, Frm2, To2) ←  To1 ≠ To2.
          distinct(Frm1, To1, Frm2, To2) ←  Frm1 ≠ Frm2.
          select_larger(X, Y, X) ←    X ≥ Y.
          select_larger(X, Y, Y) ←    Y > X.
```

As demonstrated by these examples, XY-stratified programs allow an efficient logic-based expression of procedural algorithms—in fact the alternating fixpoint used in the computation of well-founded models can also be expressed using these programs (Kemp et al., 1995). Also observe that, for the examples used here, the bistate program is nonrecursive, as in the original definition of XY-stratification (Zaniolo et al., 1993). In general, by making the computation of the recursive predicate explicit as it was done for the `anc` example, it is possible to rewrite a program with a recursive bistate graph into one where the bistate program is nonrecursive.

Aggregates and Choice in XY-stratified Programs

The notion of XY-stratification can be extended easily to programs with choice and aggregates (Zaniolo et al., 1997).

Let P be a choice program with negated goals. Then P is said to be *XY-stratified* w.r.t. negation when the following two conditions hold:

- The program obtained from P by removing its choice goals is XY-stratified w.r.t. negation, and
- If r is a recursive choice rule in P, then *some* choice goal of r contains the variable from r's temporal arguments in its left side.

Choice programs stratified w.r.t. negation always have (total) stable models (Giannotti et al., 1999) and there is a simple procedure for computing a stable model for such a program. The procedure begins with constructing the bistate version P_{bis} of a program P by (i) dropping the temporal variable from the choice goals, and (ii) applying the standard transformation to the rest of the rule. The synchronized version $syncbi(P)$ is constructed as previously described. Finally, Procedure 4 can be applied without modification.

XY-stratified programs with choice provide a very powerful modeling tool that has, e.g., been used to model the AI planning problem (Brogi et al., 1997) and active database rules (Zaniolo, 1997).

Let us consider now the situation of a recursive program P with aggregates. There is no restriction on P if it contains only monotonic aggregates and no negation. But recursive programs with nonmonotonic aggregates, or with negated goals and monotonic aggregates must satisfy the XY-stratification requirements discussed next. Say that given a program P with aggregates, we transform it into an equivalent choice program P' by the transformation described in Section 2. For P' to be an XY-stratified choice program, P must satisfy the following requirements:

- The bistate version of P must be stratified w.r.t. negation and nonmonotonic aggregates, and
- For each recursive aggregate rule, the temporal variable must be contained in the group-by attributes.

After checking these simple conditions, the $\mathcal{LDL}$++ compiler proceeds with the usual computation of *syncbi*(P) as previously described.

Floyd's algorithm to compute the least-cost path between pairs of nodes in a graph can be expressed by the following XY-stratified program, where $\mathtt{g(X,Y,C)}$ denotes an arc from $\mathtt{X}$ to $\mathtt{Y}$ of cost $\mathtt{C}$:

Example 9 *Floyd's least-cost paths between all node pairs.*

$$\begin{array}{ll}
\mathtt{delta(0,X,Y,C)} \leftarrow & \mathtt{g(X,Y,C)}. \\
\mathtt{new(J+1,X,Z,C)} \leftarrow & \mathtt{delta(J,X,Y,C1), all(J,Y,Z,C2), C = C1 + C2}. \\
\mathtt{new(J+1,X,Z,C)} \leftarrow & \mathtt{all(J,X,Y,C1), delta(J,Y,Z,C2), C = C1 + C2}. \\
\multicolumn{2}{l}{\mathtt{newmin(J,X,Z,min < C >)} \leftarrow \ \mathtt{new(J,X,Z,C)}.} \\
\mathtt{discard(J,X,Z,C)} \leftarrow & \mathtt{newmin(J,X,Z,C1), all(J,X,Z,C2), C1 \geq C2}. \\
\mathtt{delta(J,X,Z,C)} \leftarrow & \mathtt{newmin(J,X,X,C)}, \neg\mathtt{discard(J,X,Z,)}. \\
\mathtt{all(J+1,X,Z,C)} \leftarrow & \mathtt{all(J,X,Z,C)}, \neg\mathtt{delta(J+1,X,Z,_)}. \\
\mathtt{all(J,X,Z,C)} \leftarrow & \mathtt{delta(J,X,Z,C)}.
\end{array}$$

This example uses a nonmonotonic min aggregate to select the least cost pairs among those just generated (observe that the temporal variable `J` is among the group-by attributes in this rule). The next rule derives the new `delta` pairs by discarding from `new` those that are larger than existing pairs in `all`. The new `delta` is then used to update `all` and compute new pairs using the new rules.

XY-stratified programs can naturally model AI planning problems, since preconditions can simply be expressed by rules, choice can be used to select among applicable actions, and frame axioms can be expressed by XY-stratified rules that describe changes from the old state to the new state (Brogi et al., 1997).

5 CONCLUSION

The constructs described in this paper are fully operational in the version of $\mathcal{LDL}$++ implemented at UCLA (Zaniolo et al. 1998) and can express and support effectively complex queries and knowledge-based applications. This claim is supported by the applications developed in $\mathcal{LDL}$++ (Tsur, 1991), which include knowledge-based applications, such as ontology-based cooperation of heterogeneous agents in heterogeneous information systems (Tsur, 1991), and datamining applications (Shen et al., 1996, Bonchi et al., 1999). Furthermore, these constructs have a declarative formal semantics based on the concept of total stable models and can therefore be used to characterize other computational notions, such as AI planning (Brogi et al., 1997) and active database rules (Zaniolo, 1997). Finally, using these constructs, many elegant greedy algorithms that were not easily expressed in Prolog can be expressed in $\mathcal{LDL}$++, using rules that provide a natural logic-based abstraction for these algorithms (Greco and Zaniolo, 1998).

The concepts of choice, XY-stratification and aggregates can be viewed as syntactic devices to harness the power of stable models in order to achieve user-friendliness and efficient computation. XY-stratification is used to express *deterministic* computations, including the computation of well-founded models. Nondeterministic computations can instead be expressed by the choice construct, which also provides a simple characterization of monotonic aggregates (defined using only early return rules) and nonmonotonic aggregates (those using final return rules). All these constructs can be freely mixed in programs, provided that negation and nonmonotonic aggregates are used in stratified or XY-stratified programs. Programs that obey this condition always have total stable models computable in polynomial time, but also have clear intuitive meaning that can be easily mastered by users, who need not understand the subtleties of stable models.

Many of these advantages are unique to $\mathcal{LDL}$++; in particular, the well-founded model semantics used in several systems (Ramakrishnan et al., 1993, Rao et al., 1997) is not conducive to nondeterministic computations or user-defined aggregates. On the other hand, other form of nondeterminism, such as disjunctive logic programs, are prone to computational problems, whereas $\mathcal{LDL}$++ programs assure efficient polynomial-time execution. In fact, we have implemented user-defined aggregates in a commercial database system (including monotonic aggregates in recursive queries) and achieved performance levels comparable to those of the aggregates built in by the vendor (Wang and Zaniolo, 2000).

Acknowledgements

The authors would like to thank the referees for the improvements they have suggested.

References

Abiteboul, S., Hull, R. and Vianu, V. (1995). *Foundations of Databases.* Addison-Wesley, Reading, MA, 1995.

Baudinet, M., Chomicki, J., and Wolper, P. (1994). *Temporal Deductive Databases*, Chapter 13 of Temporal Databases: Theory, Design, and Implementation, A. Tansel et al. (eds), pp. 294-320, Benjamin/Cummings, 1994.

Brogi, A, Subrahmanian, V. S. and Zaniolo, C. (1997). The Logic of Totally and Partially Ordered Plans: A Deductive Database Approach, *Annals of Mathematics and Artificial Intelligence* 19(1-2): 27-58 (1997).

Eiter, T. et al. (2000). Declarative Problem-Solving Using the DLV System, In this volume.

Bonchi, F. et al. (1999). "Applications of $\mathcal{LDL}$++ to Datamining: A Classification-Based Methodology for Planning Audit Strategies in Fraud Detection", *Proc. Fifth ACM SIGKDD Int. Conference on Knowledge Discovery and Data Mining, KDD'99* 175-184, ACM, 1999.

Chimenti, D. et al. (1990). The LDL System Prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 76-90 (1990).

Finkelstein, S. J., et al.(1996) Expressing Recursive Queries in SQL, ISO WG3 report X3H2-96-075, March 1966.

Hellerstein, J. M., Haas, P.J. and Wang, H.J. (1997). "Online Aggregation". *Proc. ACM SIGMOD Int. Conference on Management of Data*, 171-182, ACM, 1997.

Ganguly, S., Greco, S. and Zaniolo, C. (1995). "Extrema Predicates in Deductive Databases," *JCSS* 51(2): 244-259 (1995)

Gelfond, M. and Lifschitz, V. (1988). The Stable Model Semantics for Logic Programming. *Proc. Joint International Conference and Symposium on Logic Programming*, R. A. Kowalski and K. A. Bowen, eds., pp. 1070–1080, MIT Press, 1988.

Giannotti, F. et al. (1981). Non-Determinism in Deductive Databases. *DOOD'91*, C. Delobel, M. Kifer, Y. Masunaga (Eds.), pp. 129-146, Springer, 1991.

Giannotti, F., et al. (1999). On the Effective Semantics of Nondeterministic, Nonmonotonic, Temporal Logic Databases, *Proc. 12th Int. Workshop, Computer Science Logic*, pp. 58-72, LNCS Vol. 1584, Springer, 1999.

Giannotti, F., Pedreschi, D. and Zaniolo, C. (2000). "Semantics and Expressive Power of Non-Deterministic Constructs in Deductive Databases," *Journal of Computer and System Sciences*, to appear.

Greco, S. and Zaniolo, C. (1998). Greedy Algorithms in Datalog with Choice and Negation, *Proc. 1998 Joint Int. Conference & Symposium on Logic Programming, JCSLP'98*, pp. 294-309, MIT Press, 1998.

Kemp, D., Ramamohanarao, K., and Stuckey, P. (1995). ELS Programs and the Efficient Evaluation of Non-Stratified Programs by Transformation to ELS. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases: DOOD'95*, T. W. Ling, A. O. Mendelzon, L. Vieille (Eds.): pp. 91–108, Springer, 1995.

Kemp, D. and Ramamohanarao, K. (1998). Efficient Recursive Aggregation and Negation in Deductive Databases. *TKDE* 10(5): 727-745 (1998).

Krishnamurthy, R., Naqvi, S. (1988) Non-Deterministic Choice in Datalog. *Proc. 3nd Int. Conf. on Data and Knowledge Bases*, pp. 416-424, Morgan Kaufmann, Los Altos (1988).

Minker, J. (1996). Logic and Databases: A 20 Year Retrospective. *Proc. International Workshop on Logic in Databases (LID'96)*, D. Pedreschi and C. Zaniolo (eds.), pp. 5–52, Springer-Verlag, 1966.

Przymusinski, T.C. (1998). On the Declarative and Procedural Semantics of Stratified Deductive Databases. In J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming*, pp. 193–216, Morgan Kaufman, San Francisco, CA, 1988.

Lausen, G., Ludaescher, B. and May, W. (1998). On Logical Foundations of Active Databases, *In Logics for Databases and Information Systems*, J. Chomicki and G. Saake (Eds.), pp. 389-422 Kluwer Academic Publishers, pp. 375–398, 1998.

Rao, P., et al. (1997). XSB: A System for Efficiently Computing WFS. Proc. Fourth Int. Conference on Logic Programming and Non-Monotonic Reasoning, *LPNMR'97*, J. Dix, U. Furbach, A. Nerode (Eds.), pp. 431-441, Springer 1997

Ramakrishnan, R., et al. (1993). Implementation of the CORAL Deductive Database System. *Proc. International ACM SIGMOD Conference on Management of Data*, pp. 167–176, 1993.

Ramakrishnan, R., and Ullman J.D. (1995). A survey of deductive database systems. *JLP*, 23(2): 125-149 (1995)

Ross, K.A. (1994). Modular Stratification and Magic Sets for Datalog Programs with Negation. *Journal of ACM* 41(6):1216–1266, 1994.

Ross, K.A. and Sagiv, Y. (1997). "Monotonic Aggregation in Deductive Database", *JCSS*, 54(1), 79-97 (1997).

Shen, W., et al. (1996). Metaqueries for Data Mining, Chapter 15 of *Advances in Knowledge Discovery and Data Mining*, U. M. Fayyad et al (eds.), pp. 201-217, MIT Press, 1996.

Schlipf, J.S. (1992). A Survey of Complexity and Undecidability Results in Logic Programming, *Proc. Workshop on Structural Complexity and Recursion-Theoretic Methods in Logic Programming*, 1993, pp.143–164

Vaghani, J., et .al (1994) The Aditi Deductive Database System. *The VLDB Journal*, 3(2), pp. 245-288 (1994).

Van Gelder, A., Ross, K.A. and Schlipf, J.S. (1990). The Well-Founded Semantics for General Logic Programs. *Journal of ACM* 38:620–650, 1991.

Van Gelder, A. (1990). Foundations of Aggregations in Deductive Databases *Proc. of Int. Conf. On Deductive and Object-Oriented databases, DOOD'93*, S. Ceri, K. Tanaka, S. Tsur (Eds.), pp. 13-34, Springer, 1993.

Tsur, S. (1991). Deductive Databases in Action, *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages*, pp. 142–154, 1991.

Zaniolo, C., Arni, N. Ong, K. (1993). Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}$++ Approach. *DOOD'93*, S. Ceri, K. Tanaka, S. Tsur (Eds.), pp. 204–221, Springer, 1993.

Wang, H. and Zaniolo, C. (2000). User-Defined Aggregates in Object-Relational Database Systems. *International Conference on Database Engineering*, pp. 111-121, IEEE Press, 2000.

Zaniolo, C. and Wang, H. (1999). Logic-Based User-Defined Aggregates for the Next Generation of Database Systems. In *The Logic Programming Paradigm: Current Trends and Future Directions.* K.R. Apt, V. Marek, M. Truszczynski, D.S. Warren (eds.), Springer Verlag, pp. 401-424, 1999.

Zaniolo, et al. (1997) *Advanced Database Systems*, Morgan Kaufmann Publishers, 1997.

Zaniolo, C. (1997). The Nonmonotonic Semantics of Active Rules in Deductive Databases. *DOOD* 1997, F. Bry, R. Ramakrishnan, K. Ramamohanarao (Eds.) pp. 265-282, Springer, 1997.

Zaniolo, C. et al. (1998) $\mathcal{LDL}$++ Documentation and Web Demo, 1988: `http://www.cs.ucla.edu/ldl`