The Deductive Database System $\mathcal{LDL}$++

FAIZ ARNI

InferData Corporation, 8200 N. MoPac Expressway, Austin, TX 78759, USA
(*e-mail:* farni@inferdata.com)

KAYLIANG ONG

Trilogy Inc., 5001 Plaza on the Lake, Austin, TX 78746, USA
(*e-mail:* kayliang.ong@trilogy.com)

SHALOM TSUR

BEA Systems, 2315 N. First Street, San Jose, CA 95131, USA
(*e-mail:* dicktsur@bea.com)

HAIXUN WANG

IBM T. J. Watson Research Center, 30 Saw Mill River Rd., Hawthorne, NY 10532, USA
(*e-mail:* haixun@us.ibm.com)

CARLO ZANIOLO

Computer Science Department, University of California, Los Angeles, CA, 90095, USA
(*e-mail:* zaniolo@cs.ucla.edu)

Abstract

This paper describes the $\mathcal{LDL}$++ system and the research advances that have enabled its design and development. We begin by discussing the new nonmonotonic and nondeterministic constructs that extend the functionality of the $\mathcal{LDL}$++ language, while preserving its model-theoretic and fixpoint semantics. Then, we describe the execution model and the open architecture designed to support these new constructs and to facilitate the integration with existing DBMSs and applications. Finally, we describe the lessons learned by using $\mathcal{LDL}$++ on various tested applications, such as middleware and datamining.

1 Introduction

The $\mathcal{LDL}$++ system, which was completed at UCLA in the summer of 2000, concludes a research project that was started at MCC in 1989 in response of the lessons learned from of its predecessor, the $\mathcal{LDL}$ system. The $\mathcal{LDL}$ system, which was completed 1988, featured many technical advances in language design (Naqvi & Tsur, 1989), and implementation techniques (Chimenti *et al.*, 1990). However, its deployment in actual applications (Tsur, 1990a; Tsur, 1990b) revealed many problems and needed improvements, which motivated the design of the new $\mathcal{LDL}$++ system. Many of these problems were addressed in the early versions of the $\mathcal{LDL}$++ prototype that were built at MCC in the period 1990–1993; but other problems, particularly limitations due to the stratification requirement, called for advances on nonmonotonic semantics, for which solutions were discovered and incorporated into the system over time—till the last version (Version 5.1) completed at UCLA in the summer of 2000.

In this paper, we will concentrate on the most innovative and distinctive features of $\mathcal{LDL}$++, which can be summarized as follows:

- Its new language constructs designed to extend the expressive power of the language, by allowing negation and aggregates in recursion, while retaining the declarative semantics of Horn clauses,
- Its execution model designed to support (i) the new language constructs, (ii) data-intensive applications via tight coupling with external databases, and (iii) an open architecture for extensibility to new application domains,
- Its extensive application testbed designed to evaluate the effectiveness of deductive database technology on data intensive applications and new domains, such as middleware and data mining.

2 The Language

A challenging research objective pursued by $\mathcal{LDL}$++ was that of extending the expressive power of logic-based languages beyond that of $\mathcal{LDL}$ while retaining a fully declarative model-theoretic and fixpoint semantics. As many other deductive database systems designed in the 80s (Minker, 1996), the old $\mathcal{LDL}$ system required programs to be stratified with respect to nonmonotonic constructs such as negation and set aggregates (Ramakrishnan & Ullman, 1995). While stratification represented a major step forward in taming the difficult theoretical and practical problems posed by nonmonotonicity in logic programs, it soon became clear that it was too restrictive for many applications of practical importance. Stratification makes it impossible to support efficiently even basic applications, such as Bill of Materials and optimized graph-traversals, whose procedural algorithms express simple and useful generalizations of transitive closure computations. Thus, deductive database researchers have striven to go beyond stratification and allow negation and aggregates in the recursive definitions of new predicates. $\mathcal{LDL}$++ provides a comprehensive solution to this complex problem by the fully integrated notions of (i) `choice`, (ii) User Defined Aggregates (UDAs), and (iii) XY-stratification. Now, XY-stratification generalizes stratification to support negation and (nonmonotonic) aggregates in recursion. However, the choice construct (used to express functional dependency constraints) defines mappings that, albeit nondeterministic, are monotonic and can thus be used freely in recursion. Moreover, this construct makes it possible to provide a formal semantics to the notion of user-defined aggregates (UDAs), and to identify a special class of UDAs that are monotonic (Zaniolo & Wang, 1999); therefore, the $\mathcal{LDL}$++ compiler recognizes monotonic UDAs and allows their unrestricted usage in recursion. In summary, $\mathcal{LDL}$++ provides a two-prong solution to the nonmonotonicity problem, by (i) enlarging the class of logic-based constructs that are monotonic (with constructs such as choice and monotonic aggregates), and (ii) supporting XY-stratification for hard-core nonmonotonic constructs, such as negation and nonmonotonic aggregates.

These new constructs of $\mathcal{LDL}$++ are fully integrated with all other constructs, and easy to learn and use. Indeed, a user needs not to know abstract semantic concepts, such as stable models or well-founded models; instead, the user only needs to follow simple syntactic rules—the same rules that are then checked by the compiler. In fact, the semantic well-formedness of $\mathcal{LDL}$++ programs can be checked at compile time—a critical property of stratified programs that was lost in later extensions, such as modular stratification (Ross, 1994). These new constructs are described next.

2.1 Functional Constraints

Say that we have a database containing the relations `student(Name, Major, Year)` and `professor(Name, Major)`. In fact, let us take a toy example that only has the following facts[1]

```
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

```
elig_adv('JimBlack', ohm).
elig_adv('JimBlack', bell).
```

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

Example 2.1
Computation of unique advisors by a choice rule

```
actual_adv(S, P) ←  student(S, Majr, Yr), professor(P, Majr),
                    choice((S), (P)).
```

The goal `choice((S), (P))` can also be viewed as enforcing a *functional dependency* (FD) `S` → `P` on the results produced by the rule; thus, in `actual_adv`, the second column (professor name) is functionally dependent on the first one (student name). Therefore, we will refer to `S` and `P`, respectively, as the left side and the right side of this FD, and of the choice goal defining it. 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.

[1] We follow the standard convention of using upper case initials to denote variables; lower case initials and strings enclosed in quotes denote constants.

The result of the rule of Example 2.1 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 2.1 is as follows:

Example 2.2
The first order equivalent for Example 2.1

```
actual_adv(S,P) ←    student(S,Majr,Yr), professor(P,Majr),
                     chosen(S,P).
chosen(S,P) ←        student(S,Majr,Yr), professor(P,Majr),
                     ¬diffChoice(S,P).
diffChoice(S,P) ←    chosen(S,P′), P ≠ P′.
```

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 2.1
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 from 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) $Y_i \neq Y_i'$ denotes the inequality of the vectors; i.e., $Y_i \neq Y_i'$ is true when for some variable $A \in Y_i$ and its primed counterpart $A' \in Y_i'$, $A \neq A'$.

Monotonic Nondeterminism

Theorem 2.1
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 2.1, the well-founded model yields undefined values for advisors. Therefore, the choice construct can express nondeterministic semantics, which can be also expressed by stable models, but not by well-founded models. On the other hand, the choice model avoids the exponential complexity which is normally encountered under stable model semantics. Indeed, the computation of stable models is $\mathcal{NP}$-hard (Schlipf, 1993), 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.*, 2001b). Indeed, the use of choice rules in positive programs preserves 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.*, 2001b):

Theorem 2.2
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 $M' \supseteq M$.

Two concrete semantics are possible for choice programs: one is an all-answers semantics, and the other is the semantics under which any answer will do—don't care nondeterminism. While an all-answers semantics for choice is not without interesting applications (Greco & Saccà, 1997), the single-answer semantics was adopted by $\mathcal{LDL}$++, because this is effective at supporting *DB-PTime* problems (Abiteboul *et al.*, 1995). Then, we see that Theorem 2 allows us to compute results incrementally as it is done in differential fixpoint computations; in fact, to find an answer, a program with choice can be implemented as an ordinary program, where the choice predicates are memorized in a table; then newly derived atoms that violate the choice FDs are simply discarded, much in the same way as duplicate atoms are discarded during a fixpoint computation. Thus positive choice programs represent a class of logic programs that are very well-behaved from both a semantic and a computational viewpoint. The same can be said for choice programs with stratified negation that are defined next.

Definition 2.2
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.*, 1998).

The Power of Choice

The expressive power of choice was studied in (Giannotti *et al.*, 2001b), where it was shown that stratified Datalog with choice can express all computations that are polynomial in the size of the database (i.e., DB-PTIME queries (Abiteboul *et al.*, 1995)). Without choice, DB-PTIME cannot be expressed in stratified Datalog, unless a predefined total order is assumed for the universe, an assumption that would violate the *genericity* principle (Abiteboul *et al.*, 1995). In terms of computational power, non-determinism and order fulfill a similar function (Abiteboul *et al.*, 1995); in fact, the application of choice can also be viewed as non-deterministically and incrementally generating a possible order on the universe—an order that is made explicit by the predicate `chain` discussed in Example 2.4.

Before moving to Example 2.4, however, we would like to observe that the version of choice supported in $\mathcal{LDL}$++ is more powerful than other nondeterministic constructs, such as the witness operator (Abiteboul *et al.*, 1995), and an earlier version of choice proposed in (Krishnamurthy & Naqvi, 1998) (called static choice in (Giannotti *et al.*, 2001b)). For instance, the following query cannot be expressed in standard Datalog (since the query is nondeterministic) nor it can be expressed by the early version of choice (Krishnamurthy & Naqvi, 1998) or by the witness construct (Abiteboul *et al.*, 1995). These early constructs express nondeterminism in nonrecursive programs, but suffer from inadequate expressive power in recursive programs (Giannotti *et al.*, 2001b). In particular, they cannot express the query in Example 2.3.

Example 2.3

*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:

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

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

```
g(a,b).  g(b,a).
g(b,c).  g(c,b).
g(a,c).  g(c,a).
```

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 `Y` $\neq$ `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)`.

In addition to supporting *nondeterministic* queries, the introduction of the choice extends the power of Datalog for *deterministic* queries. 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; then checks if the last element in the chain is even.

Example 2.4

The odd parity query by arranging the elements of a set in a chain. 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)).
odd(X) ←          chain(nil, X).
odd(Z) ←          odd(X), chain(X, Y), chain(Y, Z).
isodd ←           odd(X), ¬chain(X, Y).
```

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.

The negated goal in the last rule defines the last element in the chain. Observe that the final isodd answer does not depend on the particular chain constructed; it only depends on its length that is equal to the cardinality of the set. Thus stratified Datalog with choice can express deterministic queries, such as the parity query, that cannot be expressed in stratified Datalog without choice (Abiteboul *et al.*, 1995).

The parity 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 benefits of this added expressive power in real-life applications follows from the fact that the chain program used in Example 2.4, above, to compute the odd parity query can be used to stream through the elements of a set one by one, and compute arbitrary aggregates on them. For instance, to count the cardinality of the set d(Y) we can write:

```
mcount(X, 1) ←     chain(nil, X).
mcount(Y, J1) ←    mcount(X, J), chain(X, Y), J1 = J + 1.
count(J) ←         mcount(X, J), ¬chain(X, Y).
```

The negated goal in the last rule qualifies the element(s) X without a successor in the chain, i.e., X for which ¬chain(X, Y) holds for all Ys. Therefore, count is defined by a program containing (and stratified with respect to) negation; thus, if count is then used as a builtin aggregate, the stratification requirement must be enforced upon every program that uses count.

However, if we seek to determine if the base relation d(Y) has more than 14 elements, then we can use the mcount aggregate instead of count, as follows:

```
morethan14 ←   mcount(_, J), J > 14.
```

Now, mcount is what is commonly known as an *online* aggregate (Hellerstein *et al.*, 1997): i.e., an aggregate that produces *early returns* rather than final returns as traditional aggregates. The use of mcount over count offers clear performance benefits; in fact, the computation of morethan14 can be terminated after 14 items, whereas the application of count requires visiting all the items in the chain. From a logical viewpoint, the benefits are even greater, since count is no longer needed and the rule defining it can be eliminated—leaving us with the program defining mcount, which is free of negation. Thus, no restriction is needed when using mcount in recursive programs; and indeed, `mcount` (and `morethan14`) define monotonic mappings in the lattice of set-containment.

In summary, the use of choice led us to (i) a simple and general definition of the concept of aggregates, including user defined aggregates (UDAs), and (ii) the identification of a special subclass of UDAs that are free from the yoke of stratification, because they are monotonic. This topic is further discussed in the next section.

2.2 User Defined Aggregates

The importance of aggregates in deductive databases has been recognized for a long time (Ross & Sagiv, 1997; Van Gelder, 1993; Kemp *et al.*, 1998). In particular, there have been several attempts to overcome the limitations placed on the use of aggregates in programs because of their nonmonotonic nature (Finkelstein, 1996). Of particular interest is the work presented in (Ross & Sagiv, 1997), where it shown that rules with aggregates often define monotonic mappings in special lattices—i.e., in lattices different from the standard set-containment lattice used for T_P. Furthermore, programs with such monotonic aggregates can express many interesting applications (Ross & Sagiv, 1997). Unfortunately, the lattice that makes the aggregate rules of a given program monotonic is very difficult to identify automatically (Van Gelder, 1993); this problem prevents the deployment of such a notion of monotonicity in real deductive database systems.

A new wave of decision support applications has recently underscored the importance of aggregates and the need for a wide range of new aggregates (Han & Kamber, 2001). Examples include rollups and datacubes for OLAP applications, running aggregates and window aggregates in time series analysis, and special versions of standard aggregates used to construct classifiers or association rules in datamining (Han & Kamber, 2001). Furthermore, a new form of aggregation, called online aggregation, finds many uses in data-intensive applications (Hellerstein *et al.*, 1997). To better serve this wide new assortment of applications requiring specialized aggregates, a deductive database system should support User Defined Aggregates (UDAs). Therefore, the new $\mathcal{LDL}$++ system supports powerful UDAs, including online aggregates and monotonic aggregates, in a simple rule-based framework built on formal logic-based semantics.

In $\mathcal{LDL}$++ users can define a new aggregate by writing the `single`, `multi`, and `freturn` rules (however, `ereturn` rules can be used to supplement or replace `freturn` rules). The `single` rule defines the computation for the first element of the set (for instance `mcount` has its second argument set to 1), while `multi` defines the induction step whereby the value of the aggregate on a set of $n+1$ elements is derived from the aggregate value of the previous set with n elements and the value of $(n+1)^{th}$ element itself. A unique aggregate name is used as the first argument in the head of these rules to eliminate any interference between the rules defining different aggregates. For instance, for computing averages we must compute both the count and the sum of the elements seen so far:

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

Then, we write a `freturn` rule that upon visiting the final element in `d(Y)` produces the ratio of sum over count, as follows:

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

After an aggregate is defined by its `single`, `multi`, `ereturn` and/or `freturn` rules, it can be invoked and used in other rules. For instance, our the newly defined `avg` can be invoked as follows:

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

Thus $\mathcal{LDL}$++ uses the special notation of pointed brackets, in the head of rules, to denote the application of an aggregate. This syntax, that has been adopted by other languages (Ramakrishnan *et al.*, 1993), also supports an implicit 'group by' construct, whereby the aggregate arguments in the head are implicitly grouped by the other arguments in the head. Thus, to find the average salary of employees grouped by department a user can write the following rule:

$$\texttt{davg(DeptNo, avg}\langle\texttt{Sal}\rangle\texttt{)} \leftarrow \quad \texttt{employee(Eno, Sal, DeptNo)}.$$

The formal semantics of UDAs was introduced in (Zaniolo & Wang, 1999) and is described in the Appendix: basically, the aggregate invocation rules and the aggregate definition rules are rewritten into an equivalent program that calls on the chain predicate defined as in Example 2.4. (Naturally, for the sake of efficiency, the $\mathcal{LDL}$++ system shortcuts the full rewriting used to define their formal semantics, and implement the UDAs by a more direct implementation.)

$\mathcal{LDL}$++ UDAs have also been extended to support online aggregation (Hellerstein *et al.*, 1997). This is achieved by using `ereturn` rules in the definition of UDAs, to either supplement, or replace `freturn` rules.

For example, the computation of averages normally produces an approximate value long before the whole data set is visited. Then, we might want to see the average value obtained so far every 100 elements. Then, the following rule will be added:

$$\begin{aligned}&\texttt{ereturn(avg, X, (Sum, Count), Avg)} \leftarrow\\&\qquad\qquad\qquad\qquad\texttt{Count mod 100} = 0, \texttt{Avg} = \texttt{Sum/Count}.\end{aligned}$$

Thus the `ereturn` rules produce *early returns*, while the `freturn` rules produce final returns.

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 2.5
Coalescing overlapping intervals sorted by start time.

```
empProj(Eno, coales⟨(From, To)⟩) ←   emp(Eno, _, _, (From, To)).


single(coales, (Frm, To), (Frm, To)).
multi(coales, (Nfr, Nto), (Cfr, Cto), (Cfr, Nto)) ←
                                        Nfr <= Cto, Nto > Cto.
multi(coales, (Nfr, Nto), (Cfr, Cto), (Cfr, Cto)) ←
                                        Nfr <= Cto, Nto <= Cto.
multi(coales, (Nfr, Nto), (Cfr, Cto), (Nfr, Nto)) ←   Cto < Nfr.

ereturn(coales, (Nfr, Nto), (Cfr, Cto), (Cfr, Cto)) ←   Cto < Nfr.
freturn(coales, _, LastInt, LastInt).
```

Since the input intervals are ordered by their start time, the new interval (Nfr, Nto) overlaps the current interval (Cfr, Cto) when Nfr $\leq$ Cto; in this situation, the two intervals are merged into one that begins at Cfr and ends with the larger of Nto and Cto. When, the new interval does not overlap with the current interval, this is returned by the ereturn rule, while the new interval becomes the current one (see the last multi rule).

Let P be a program. A rule r of P whose head contains aggregates is called an *aggregate rule*. Then, P is said to be stratified w.r.t. aggregates when for each aggregate rule r in P, the stratum of r's head predicate is strictly higher than the stratum of each predicate in the head of r. Therefore, the previous program is stratified with respect to coales which is nonmonotonic since it uses both early returns and final returns.

While, programs stratified with respect to aggregates can be used in many applications, more advanced applications require the use of aggregates in more general settings. Thus, $\mathcal{LDL}$++ supports the usage of arbitrary aggregates in XY-stratified programs, which will be discussed in Section 3. Furthermore $\mathcal{LDL}$++ supports the monotonic aggregates that can be used freely in recursion.

Monotone Aggregation

An important result that follows from the formalization of the semantics of UDAs (Zaniolo & Wang, 1999) (see also Appendix), is that UDA defined without final return rules, i.e., no freturn rule, define monotonic mappings, and can thus be used without restrictions in the definition of recursive predicates. For instance, we will next define a continuous count that returns the current count after each new element (thus final returns are here omitted since they are redundant).

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

Monotonic aggregates allow us to express the following two examples taken from (Ross & Sagiv, 1997).

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

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

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

$$\begin{array}{ll}
\mathtt{friend(jerry,mark).} & \mathtt{sure(mark).} \\
\mathtt{friend(penny,mark).} & \mathtt{sure(tom).} \\
\mathtt{friend(jerry,jane).} & \mathtt{sure(jane).} \\
\mathtt{friend(penny,jane).} & \\
\mathtt{friend(jerry,penny).} & \\
\mathtt{friend(penny,tom).} &
\end{array}$$

Then, the basic semi-naive computation yields:

$$\mathtt{willcome(mark),willcome(tom),willcome(jane),}$$

$$\mathtt{c\_friends(jerry,1),c\_friends(penny,1),c\_friends(jerry,2),}$$

$$\mathtt{c\_friends(penny,2),c\_friends(penny,3),willcome(penny),}$$

$$\mathtt{c\_friends(jerry,3),willcome(jerry).}$$

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

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

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

```
single(msum, Y, Y).
multi(msum, Y, Old, New) ←      New = Old + Y.
ereturn(msum, Y, Old, New) ←    New = Old + Y.
```

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. For instance, let us say 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:

```
part_cost(Part, 0, Cst) ←          basic_part(Part, Cst).
part_cost(Part, mcount⟨Sb⟩, msum⟨MCst⟩) ←
                        part_cost(Sb, ChC, Cst), prolfc(Sb, ChC),
                        assembly(Part, Sb, Mult), MCst = Cst ∗ Mult.
```

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 outdegree 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:

```
prolfc(P1, count⟨P2⟩) ←   assembly(P1, P2, _).
prolfc(P1, 0) ←           basic_part(P1, _).
```

Therefore the simple and general solution of the monotonic aggregation problem introduced by $\mathcal{LDL}$++ allows the concise expression of many interesting algorithms. This concept can also be extended easily to SQL recursive queries, as discussed in (Wang & Zaniolo, 2000) where additional applications are also discussed.

2.3 Beyond Stratification

The need to go beyond stratification has motivated much recent research. Several deductive database systems have addressed it 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. The standard notion of stratification is instead much easier to use, since it provides a simple criterion for the programmer to follow and for the compiler to use when validating the program and optimizing its execution. Therefore, $\mathcal{LDL}$++ has introduced 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 argument: thus, 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 as Statelog programs that were used to model active databases (Lausen, 1998b). The deductive database system Aditi (Kemp *et al.*, 1998) also supports the closely related concept of explicitly locally stratified programs, which were 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 separate them from `marc`, can be computed using the following program which models the differential fixpoint computation:

Example 2.6
Computing ancestors of Marc and their remoteness from Marc using differential fixpoint approach.

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

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 the same as $s(J)$ that 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)`. The k^{th} stratum consists of atoms of the form `all_anc(k,X)`, `delta_anc(k,X)`. Thus, the previous 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 the previous program as a compact representation for the stratified program obtained by instantiating the temporal argument to integers and attaching them to the predicate names, thus generating an infinite sequence of unique names.

Also observe that the temporal arguments in rules are either the same as, or one less than, the temporal argument in the head. Then, there are two kinds of rules in our example: (i) X-rules (i.e., a horizontal rules) where the temporal argument in each of their goals is the same as that in their heads, and (ii) Y-rules (i.e., a vertical rules) where the temporal arguments in some of their goals are one less than those in their heads. 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 2.6 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 labelling all the head predicates in P with the prefix 'new'. Then, the body predicates with the same temporal argument as the head are also labelled with the prefix 'new', while the others are labelled 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 2.7
The bistate version of the program in Example 2.6

```
new_delta_anc(marc).
new_delta_anc(Y) ←    old_delta_anc(X), parent(Y, X),
                      ¬old_all_anc(Y).
new_all_anc(X) ←      new_delta_anc(X).
new_all_anc(X) ←      old_all_anc(X).
```

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

Definition 2.3
Let P be an XY-program. P is said to be XY-stratified when P_{bis} is a stratified program.

Theorem 2.3
Let P be an XY-stratified program. Then P is locally stratified.

The program of Example 2.7 is stratified with the following strata: $\mathtt{S}_0$ = {parent, old_all_anc, old_delta_anc}, $\mathtt{S}_1$ = {new_delta_anc}, and $\mathtt{S}_2$ = {new_all_anc}. Thus, the program in Example 2.6 is locally stratified.

For an XY-stratified program P, the general iterated fixpoint procedure (Przymusinski, 1988) used to compute the stable model of locally stratified programs (Zaniolo *et al.*, 1993) becomes quite simple; basically it reduces to a repeated computation over the stratified program P_{bis}. For instance, for Example 2.7 we compute new_delta_anc from old_delta_anc and then new_all_anc from this. Then, the 'old' relations are re-initialized with the content of the 'new' ones so derived, and the process is repeated. Furthermore, since the temporal arguments have been removed from this program, we need to

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

```
q(J, X) ←   new_q(X), 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 2.7, we need to change the first rule to

```
new_delta_anc(marc) ←   counter(0).
```

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

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

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

Procedure 2.1
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.

Since each XY-stratified program is locally stratified (Przymusinski, 1988), it is guaranteed to have a unique stable model, which is also known as its perfect model (Przymusinski, 1988). But the special syntactic structure of XY-stratified programs allows an efficient computation of their perfect models using Procedure 4; moreover, in the actual $\mathcal{LDL}$++ implementation, this computation is further improved with the optimization techniques discussed next. For instance, the replacement of `old_q` with `new_q` described in the last step of Procedure 2.1 becomes an operation of (small) constant cost when it is implemented by switching the pointers to the relations. A second improvement concerns *copy rules*, such as the last rule in Example 2.6. 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 2.7). 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 constant-time operation. Rule r_4 that adds tuples to `new_all_anc` is then executed after r_3.

In writing XY-stratified programs, the user must also be concerned with termination conditions, since e.g., a rule such as r_3 in Example 2.6 could, if left unchecked, keep producing `all_anc` results under a new temporal argument, after `delta` becomes empty. One solution to this problem is for the user to add the goal `delta_anc(J, _)` to rule r_3. Then, the computation `all_anc` stops as soon as no new `delta_anc(J, _)` is generated. Alternatively, our program could be called from a goal such as `delta_anc(J, Y)`. In this case, if r_2 fails to produce any result for a value `J`, no more results can be produced at successive steps, since `delta_anc(J, Y)` is a positive goal of r_2. The $\mathcal{LDL}$++ system is capable of recognizing these situations, and it will terminate the computation of Procedure 2.1 when either condition occurs.

Example 2.8 solves the coalescing problem without relying on tuples being sorted on their start-time—an assumption made in Example 2.5. Therefore, we use the XY-stratified program of Example 2.8, which iterates over two basic computation steps. The first step is defined by the `overlap` rule, which identifies pairs of distinct intervals that overlap, 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 2.8
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. For instance, the alternating fixpoint procedure used in the computation of well-founded models can also be expressed using these programs (Kemp *et al.*, 1995). In general, XY-stratified programs are quite powerful, as demonstrated by fact that these programs (without choice, aggregates, and function symbol) are known to be equivalent to Statelog programs (Lausen *et al.*, 1998a) , which have PSPACE complexity and can express the WHILE queries (Abiteboul *et al.*, 1995). Finally, observe that the bistate programs for the examples used here are nonrecursive. In general, by making the computation of the recursive predicate explicit as it was done for the `anc` example, it is possible to rewrite an XY-stratified program P whose bistate version P_{bis} is recursive into an XY-stratified program P' whose bistate version P'_{bis} is nonrecursive.

Choice and Aggregates in XY-stratified Programs

As described in Section 2.1, choice can be used in stratified programs with no restriction, and its stable model can be computed by an iterated choice fixpoint procedure. Generalizing such notion, the $\mathcal{LDL}$++ system supports the use of choice in programs that are XY-stratified with respect to negation. The following conditions are however enforced to assure the existence of stable models for a given program P (Giannotti *et al.*, 1998):

- 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 r's temporal variable in its left side.

After checking these conditions, the $\mathcal{LDL}$++ compiler constructs *syncbi*(P) by dropping the temporal variable from the choice goals and transforming the rest of the rules as described in the previous section. Then, the program *syncbi*(P) so obtained is a stratified choice program and its stable models can be computed accordingly; therefore, each stable model for the original XY-stratified program P is computed by simply applying Procedure 2.1 with no modification (Zaniolo *et al.*, 1997; Giannotti *et al.*, 1998).

Using the simple syntactic characterization given in Section 2.2, $\mathcal{LDL}$++ draws a sharp distinction between monotonic and nonmonotonic aggregates. No restriction is imposed on programs with only monotonic aggregates and no negation. But recursive programs with nonmonotonic aggregates must satisfy the following conditions (which assure that once the aggregates are expanded as described in Section 2.2 the resulting choice program satisfies the XY-stratification conditions for choice programs discussed in the previous paragraph):

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

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

For instance, the following XY-stratified program with aggregates expresses Floyd's algorithm to compute the least-cost path between pairs of nodes in a graph. Here, `g(X, Y, C)` denotes an arc from `X` to `Y` of cost `C`:

Example 2.9
Floyd's least-cost paths between all node pairs.

```
delta(0, X, Y, C) ←         g(X, Y, C).
new(J + 1, X, Z, C) ←       delta(J, X, Y, C1), all(J, Y, Z, C2), C = C1 + C2.
new(J + 1, X, Z, C) ←       all(J, X, Y, C1), delta(J, Y, Z, C2), C = C1 + C2.
newmin(J, X, Z, min⟨C⟩) ←          new(J, X, Z, C).
discard(J, X, Z, C) ←       newmin(J, X, Z, C1), all(J, X, Z, C2), C1 ≥ C2.
delta(J, X, Z, C) ←         newmin(J, X, X, C), ¬discard(J, X, Z, ).
all(J + 1, X, Z, C) ←       all(J, X, Z, C), ¬delta(J + 1, X, Z, _).
all(J, X, Z, C) ←           delta(J, X, Z, C).
```

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

By supporting UDAs, choice, and XY-stratification $\mathcal{LDL}$++ provides a powerful, fully integrated framework for expressing logic-based computation and modelling. In addition to express complex computations (Zaniolo *et al.*, 1998), this power has been used to model the AI planning problem (Brogi *et al.*, 1997), database updates, and active database rules (Zaniolo, 1997). For instance, to model AI planning, 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).

3 The System

The main objectives in the design of the $\mathcal{LDL}$++ system, were (i) strengthening the architecture of the previous $\mathcal{LDL}$ system (Chimenti *et al.*, 1990), (ii) improving the system's usability and the application development turnaround time, and (iii) provide efficient support for the new language constructs.

While the first objective could be achieved by building on and extending the general architecture of the predecessor $\mathcal{LDL}$ system, the second objective forced us to depart significantly from the compilation and execution approach used by the $\mathcal{LDL}$ system. In fact, the old system adhered closely to the set-oriented semantics of relational algebra and relational databases; therefore, it computed and accumulated all partial results before returning the whole set to the user. However, our experience in developing applications indicated that a more interactive and incremental computation model was preferable: i.e., one where users see the results incrementally as they are produced. This allows developers to monitor better the computation as it progresses, helping them debugging their programs, and, e.g., allowing them to stop promptly executions that have fallen into infinite loops.

Therefore, $\mathcal{LDL}$++ uses a pipelined execution model, whereby tuples are generated one at a time as they are needed (i.e., lazily as the consumer requests them, rather than eagerly). This approach also realizes objective (iii) by providing better support for new constructs, such as choice and on-line aggregation, and for intelligent backtracking optimization (discussed in the next section).

The $\mathcal{LDL}$++ system also adopted a shallow-compilation approach that achieves faster turnaround during program development and enhances the overall usability; this approach also made it easier to support on-line debugging and meta-level extensions. The previous $\mathcal{LDL}$ system was instead optimized for performance; thus, it used a deep-compilation approach where the original program was translated into a (large) C program—whose compilation and linking slowed the development turnaround time. The architecture of the system is summarized in the next section; additional information, a web demo, and instructions on downloading for noncommercial use can be found in (Zaniolo *et al.*, 1998).

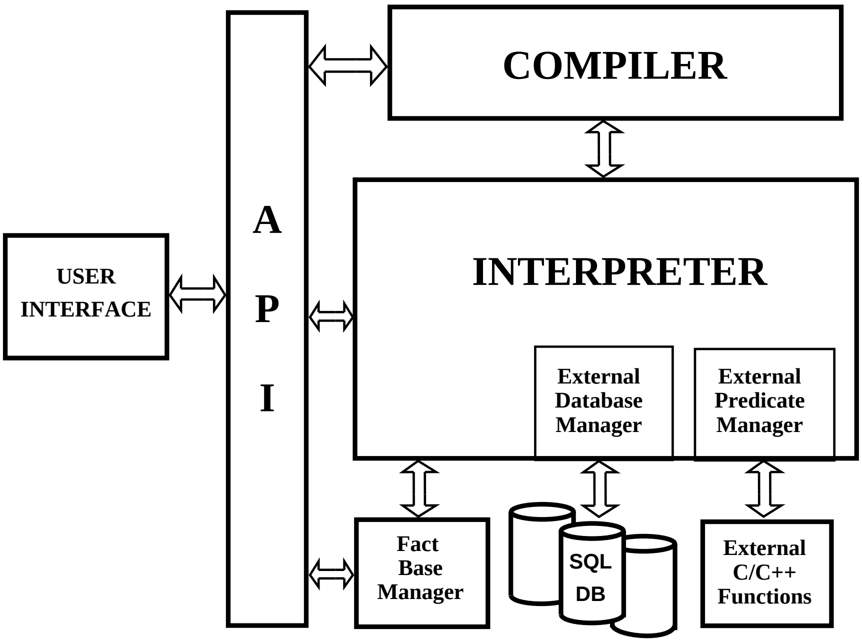

Fig. 1. $\mathcal{LDL}$++ Open Architecture

3.1 Architecture

The overall architecture of the $\mathcal{LDL}$++ system and its main components are shown in Figure 1. The major components of the system are:

The Compiler The compiler reads in $\mathcal{LDL}$++ programs and constructs the *Global Predicate Connection Graph* (PCG). For each query form, the compiler partially evaluates the PCG, transforming it into a network of objects that are executed by the interpreter. The compiler is basically similar to that of the old system (Chimenti *et al.*, 1990), and is responsible for checking the safety of queries, and rewriting the recursive rules using techniques such the Magic Sets method (Bancilhon *et al.*, 1986), and the more specialized methods for left-linear and right-linear rules (Ullman, 1989). These rewriting techniques result in an efficient execution plan for queries.

The Database Managers The $\mathcal{LDL}$ experience confirmed the desirability supporting access to (i) an internal (fast-path) database and (ii) multiple external DBMSs in a transparent fashion. This led to the design of a new system where the two types of database managers are fully integrated.

The internal database is shown in Figure 1 as Fact Base Manager. This module supports the management and retrieval of $\mathcal{LDL}$++ complex objects, including sets and lists, and of temporary relations obtained during the computation. In addition to supporting users' data defined by the schema as internal relations, the interpreter relies on the local database to store and manage temporary data sets. The internal database is designed as a virtual-memory record manager: thus its internal organization and indexing schemes are optimized for the situation where the pages containing frequently used data can reside in main memory. Data is written back onto disk at the commit point of each update transaction; when the transaction aborts the old data is instead restored from disk.

The system also supports an external database manager, which is designed to optimize access to external SQL databases; this is described in Section 3.3.

Interpreter The interpreter receives as input a graph of executable objects corresponding to an $\mathcal{LDL}$++ query form generated by the compiler, and executes it by issuing get-next, and other calls, to the local database. Similar calls are also issued by the External Database Manager and the External Predicate Manager to, respectively, external databases, and external functions or software packages that follow the C/C++ calling conventions. Details on the interpreter are presented in the next section.

User Interface All applications written in C/C++ can call the $\mathcal{LDL}$++ system via a standard API; thus applications written in $\mathcal{LDL}$++ can be embedded in other procedural systems.

One such application is a line-oriented command interpreter supporting a set of predefined user commands, command completion and on-line help. The command interpreter is supplied with the system, although it is not part of the core system. Basically, the interface is an application built in C++ that can be replaced with other front-ends, including graphical ones based on a GUI, without requiring any changes to the internals of the system. In particular, a Java-based interface for remote demoing was added recently (Zaniolo *et al.*, 1998).

3.2 Execution Model and Interpreter

The abstract machine for the $\mathcal{LDL}$++ interpreter is based upon the architecture described in (Chimenti *et al.*, 1989). An $\mathcal{LDL}$++ program is transformed into a network of active objects and the graph-based interpreter then processes these objects.

Code generation and execution Given a query form, an $\mathcal{LDL}$++ program is transformed into a Predicate Connection Graph (PCG), which can be viewed as an AND/OR graph with annotations. An OR-node represents a predicate occurrence and each AND node represents the head of a rule. The PCG is subsequently compiled into an evaluable data structure called a LAM (for $\mathcal{LDL}$++ Abstract Machine), whose nodes are implemented as instances of C++ classes. Arguments are passed from one node to the other by means of variables. Unification is done at compile time and the sharing of variables avoids useless assignments.

Each node of the generated LAM structure has a virtual[2] “GetTuple” interface, which evaluates the corresponding predicate in the program. Each node also stores a state variable that determines whether this node is being “entered” or is being “backtracked” into. The implementation of this “GetTuple” interface depends on the type of node. The most basic C++ classes are OR-nodes and AND-nodes; then there are several more specialized subclasses of these two basic types. Such subclasses include the special OR-node that serves as the distinguished *root* node for the query form, internal relations AND-nodes, external relations AND-nodes, etc.

And/OR Graph For a generic OR node corresponding to a derived relation, the “GetTuple” interface merely issues “GetTuple” calls to its children (AND nodes). Each successful invocation automatically instantiates the variables of both the child (AND node) and the parent (OR node). Upon backtracking, the last AND node which was successfully executed is executed again. The “GetTuple” on an OR node fails when its last AND node child fails.

The *Dataflow points* represent different entries into the AND/OR nodes, each entry corresponding to a different state of the computation. The dataflow points associated with each node are shown in the following table (observe their similarity to ports in Byrd’s Prolog execution model (Byrd, 1980)):

DATAFLOW	POINT	STATE OF COMPUTATION
ENTRY	*e_dest*	getting first tuple of node
BACKTRACK	*b_dest*	getting next tuple of node
SUCCESS	*s_dest*	a tuple has been generated
FAIL	*f_dest*	no more tuples can be generated

A dataflow point of a node can be directed to a dataflow point of a different node by a *dataflow destination.* The *entry destination (e_dest)* of a given node is the dataflow point to which its entry point is directed. Similarly, *backtrack (b_dest), success (s_dest)*, and *fail destinations (f_dest)* can be defined. The dataflow destinations represent logical operations between the nodes involved; for example a join or union of the two nodes. The dataflow points and destinations of a node describe how the tuples of that node are combined with tuples from other nodes (but not how those tuples are generated).

To obtain the first tuple of an OR node we get the first tuple of its first child AND node. To obtain the next tuple from an OR node we request it from the AND node that generated the previous tuple. Observe that the currently “active” AND node must be determined at run-time. When no more tuples can be generated for a given AND node, then we go to the next AND node, till the last child AND node

[2] Similar to a C++ virtual function

is reached (At this point no more tuples can be generated for the OR node). Thus, we have:

OR nodes: *e_dest*: the e_dest of the first child AND-node
b_dest: the b_dest of the "active" child AND node
f_dest: **if** node is first OR node in rule
then the f_dest point of parent AND node
else the b_dest of previous OR node
s_dest: **if** node is last OR node in a rule
then the s_dest of parent AND node
else the e_dest of next OR node.

The execution of an AND node is conceptually less complicated. Intuitively, the execution corresponds to a nested loop, where, for each tuple of the first OR node, we generate all matching tuples from the next OR node. This continues until we reach the last OR node. Thus, when generating the next tuple of an AND node, we generate the next matching tuple from the last OR node. If there are no more matching tuples, we generate the next tuple from the previous OR node. When there are no more tuples to be generated by the first OR node, we can generate no more tuples for the AND node. Thus we have:

AND nodes: *e_dest*: the e_dest of first OR child
b_dest: the b_dest of last OR child
f_dest: **if** node is last AND child
then f_dest of parent OR node
else e_ dest of next AND node
s_dest: s_dest of parent OR node.

Given a query, the $\mathcal{LDL}$++ system first finds the appropriate LAM graph for the matching query form, then stores any constant being passed to the query form by initializing the variables attached to the *root* node of the LAM graph. Finally, the system begins the execution by repeatedly calling the "GetTuple" method on the root of this graph. When the call fails the execution is complete.

Lazy Evaluation of Fixpoints $\mathcal{LDL}$++ adopts a lazy evaluation approach (*pipelining*) as its primary execution model, which is naturally supported by the AND/OR graph described above. This model is also supported through the lazy evaluation of fixpoints. The traditional implementation of fixpoints (Ullman, 1989; Zaniolo *et al.*, 1997) assumes an eager computation where new tuples are generated till the fixpoint is reached. $\mathcal{LDL}$++ instead supports lazy computation where the recursive rules produce new tuples only in response to the goal that, as a consumer, calls the recursive predicate. Multiple consumers can be served by one producer, since each consumer j uses a separate cursor C_j to access the relation R written by the producer. Whenever j needs a new tuple, it proceeds as shown in Figure 2.

A limitation of pipelining is that the internal state of each node must be kept for computation to resume where the last call left off. This creates a problem when several goals call the same predicate (i.e. the same subtree in the PCG is shared).

Multiple invocations of a shared node can interfere with each other (non-reentrant code). Solutions to this problem include (i) using a stack as in Prolog, and (ii) duplicating the source code as in the $\mathcal{LDL}$ system—thus ensuring that the PCG is a tree, rather than a DAG (Chimenti *et al.*, 1990). In the $\mathcal{LDL}$++ system, we instead use the lazy producer approach described above for situations where the calling goals have no bound argument. If there are bound arguments in consuming predicates we duplicate the node. However, since each node is implemented as a C++ class, we simply generate multiple instances of this class—i.e., we duplicate the data but still share the code.

Intelligent Backtracking Pipelining makes it easy to implement optimizations such as existential optimization and intelligent backtracking (Chimenti *et al.*, 1990). Take for instance the following example:

Example 3.1
Intelligent Backtracking.

$$\texttt{query3(A,B)} \leftarrow \quad \texttt{b1(A),p(A,B),b2(A).}$$

Take the situation where the first `A`-value generated by `b1` is passed to `p(A,B)`, which succeeds and passes the value of `A` to `b2`. If the first call to this third goal fails, there is no point in going back to `p`, since this can only return a new value for `B`. Instead, we have to jump back to `b1` for a new value of `A`. In an eager approach, all the `B`-values corresponding to each `A` are computed, even when they cannot satisfy `b2`.

Similar optimizations were also supported in $\mathcal{LDL}$ (Chimenti *et al.*, 1990), but with various limitations: e.g., existential optimization was not applied to recursive predicates, since these were not pipelined. In $\mathcal{LDL}$++, the techniques are applied uniformly, since pipelining is now used in the computation of all predicates, including recursive ones.

3.3 External Databases

A most useful feature of the $\mathcal{LDL}$++ system is that it supports convenient and efficient access to external databases. As shown in Figure 1, the External Database Interface (EDI) provides the capability to interact with external databases. The system is equipped with a generic SQL interface as well as an object-oriented design

Fig. 2. **Lazy Fixpoint Producer**

Step 1. Move the cursor C_j to the next tuple of R, and consume the tuple.
Step 2. If Step 1 fails (thus, C_j is the last tuple of R), check the fixpoint flag F.
Step 3. If the fixpoint is reached, return failure.
Step 4. If the fixpoint is not reached, call the current rule to generate a new tuple.
Step 5. If a new tuple is generated, add it to the relation R, advance C_j and return the tuple.
Step 6. Otherwise, repeat Step 2.

that allows easy access to external database systems from different vendors. To link the system with a specific external database, it is only necessary to write a small amount of code to implement vendor-specific drivers to handle data conversion and local SQL dialects. The current $\mathcal{LDL}$++ system can link directly with Sybase, Oracle, DB2, and indirectly with other databases via JDBC [3].

The rules in a program make no distinction between internal and external relations. Relations from external SQL databases are declared in the $\mathcal{LDL}$++ schema just like internal relations, with the additional specification of the type and the name of the SQL server holding the data. As a result, these external resources are transparent to the inference engine, and applications can access different databases without changes. The EDI can also access data stored in files.

The following example shows the $\mathcal{LDL}$++ schema declarations used to access an external relation `employee` in the database `payroll` running on the server `sybase_tarski`.

Example 3.2
Schema Declaration to external Sybase server.

```
database({
        sybase::employee(NAME:char(30),SALARY:int, MANAGER:char(30))
                from sybase_tarski
                use payroll
                user_name 'john'
                application_name 'downsizing'
                interface_filename '/tmp/ldl++/demo/interfaces'
                password nhoj
          } ).
```

The $\mathcal{LDL}$++ system generates SQL queries that off-loads to the external database server the computation of (i) the join, select, project queries corresponding to positive rule goals, (ii) the set differences corresponding to the negated goals, and (iii) the aggregate operations specified in the heads of the rules.

In the following example the rule defines expensive employees as those who make over 75,000 and more than their managers:

Example 3.3
SQL Generation

```
   expensive_employee(Name) <-
                employee(Name, Salary1, Manager),
                Salary1 > 75000,
                employee(Manager, Salary2, _),
                Salary1 > Salary2.
```

[3] Sybase is a trademark of Sybase Inc., Oracle is a trademark of Oracle Inc., DB2 is a trademark of IBM Inc.

The $\mathcal{LDL}$++ compiler collapses all the goals of this rule and transforms it into the following SQL node:

```
expensive_employee(Name) <- sql_node(Name).
```

where sql_node denotes the following SQL query sent to external database server:

```
SELECT  employee_0.NAME
FROM    employee employee_0, employee employee_1
WHERE   employee_0.SALARY > 75000 AND
      employee_1.NAME = employee_0.MANAGER AND
      employee_0.SALARY > employee_1.SALARY
```

Consequently, access to the external database via $\mathcal{LDL}$++ is as efficient as for queries written directly in SQL. Rules with negated goals are also supported and implemented via the NOT EXIST construct of SQL. The $\mathcal{LDL}$++ SQL interface also supports updates to external databases, including set-oriented updates with qualification conditions. Updates to external relations follow the same syntax and semantics as the updates to local relations. The execution of each query form is viewed as a new transaction: either it reaches its commit point or the transaction is aborted.

To better support middleware applications, the coupling of $\mathcal{LDL}$++ with external databases was further enhanced as follows:

- *Literal Collapsing*: The goals in the body of a rule are reordered to ensure that several goals using database relations can now be supported as a single SQL subquery to be offloaded to the DBMS.
- *Rule compression*: To offload more complex and powerful queries the remote database, literals from multiple levels of rules are combined and the rules are compressed vertically.
- *Aggregates*: Rules that contain standard SQL aggregates in their heads can also be offloaded to the remote SQL system.

3.4 Procedural Language Interface

As shown in Figure 1, the $\mathcal{LDL}$++ system is designed to achieve an open architecture where links with procedural languages, such C/C++, can be established in two ways:

- Via the Application Programming Interface (API) which allows applications to drive the system, and
- Via the External Predicate Manager which allows C/C++ functions to be imported into the inference engine as external predicates.

Via the API, any C/C++ routine can call the $\mathcal{LDL}$++ inference engine. The API provides a set of functions that enable applications to instruct the $\mathcal{LDL}$++ engine to load a schema, load rules, compile query forms, send queries, and retrieve results.

Via the external predicate manager, function defined in C/C++ can be imported into $\mathcal{LDL}$++ and treated as logical predicates callable as rule goals. A library of C/C++ functions is also provided to facilitate the manipulation of internal $\mathcal{LDL}$++ objects, and the return of multiple answers by the external functions. Therefore, external functions can have the same behavior as internal predicates in all aspects, including flow of control and backtracking. Details on these interfaces can be found in (Zaniolo *et al.*, 1998).

4 Applications

The deployment of the $\mathcal{LDL}$ and $\mathcal{LDL}$++ prototypes in various real-life applications have much contributed to understanding the advantages and limitations of deductive databases in key application domains (Tsur, 1990a; Tsur, 1990b). Moreover, this experience with application problems, has greatly influenced the design of the $\mathcal{LDL}$++ system and its successive improvements.

Recursive Queries. Our first focus was to compute transitive closures and to solve various graph problems requiring recursive queries, such as Bill-of-Materials (Zaniolo *et al.*, 1997). Unfortunately, many of these applications also require that set-aggregates, such as counts and minima, be computed during the recursive traversal of the graph. Therefore, these applications could not be expressed in $\mathcal{LDL}$ which only supported stratified semantics, and thus disallowed the use of negation and aggregation within recursive cliques. Going beyond stratification thus became a major design objective for $\mathcal{LDL}$++.

Rapid Prototyping of Information Systems. Rapid prototyping from E-R specifications has frequently been suggested as the solution for the productivity bottleneck in information system design. Deductive databases provide a rule-based language for encoding executable specifications, that is preferable to Prolog and 4GL systems used in the past, because their completely declarative semantics provides a better basis for specifications and formal methods. Indeed, $\mathcal{LDL}$ proved to be the tool of choice in the rapid *Prototyping of Information Systems* in conjunction with a structured-design methodology called *POS* (Process, Object and State) (Ackley *et al.*, 1990; Tryon, 1991). Our proof-of-concept experiment confirmed the great potential of deductive databases for the rapid prototyping of information systems; but this also showed the need for a richer environment that also supports prototyping of graphical interfaces, and the use of E-R based CASE tools. A large investment in producing such tools is probably needed before this application area can produce a commercial success for deductive databases.

Middleware At MCC, $\mathcal{LDL}$++ was used in the CARNOT/INFOSLEUTH project to support semantic agents that carry out distributed, coordinated queries over a network of databases (Ong *et al.*, 1995). In particular, $\mathcal{LDL}$++ was used to implement the ontology-driven mapping between different schemas; the main functions performed by $\mathcal{LDL}$++ include (i) transforming conceptual requests by users into a collection of cooperating queries, (ii) performing the needed data conversion, and (iii) offloading to SQL statements executable on local schemas (for both relational and O-O databases).

Scientific Databases The $\mathcal{LDL}$++ system provided a sound environment on which to experiment with next-generation database applications, e.g., to support domain science research, where complex data objects and novel query and inferencing capabilities are required.

A first area of interest was molecular biology, where several pilot applications relating to the Human Genome initiative (Erickson, 1992) were developed (Overbeek *et al.*, 1990; Tsur *et al.*, 1990). $\mathcal{LDL}$++ rules were also used to model and support taxonomies and concepts from the biological domain, and to bridge the gap between high-level scientific models and low-level experimental data when searching and retrieving domain information (Tsur, 1990b).

A second research area involves geophysical databases for atmospheric and climatic studies (Muntz *et al.*, 1995). For instance, there is a need for detecting and tracking over time and space the evolution of synoptic weather patterns, such as cyclones. The use of $\mathcal{LDL}$++ afforded the rapid development of queries requiring sophisticated spatio-temporal reasoning on the geographical database. This first prototype was then modified to cope with the large volume of data required, by off-loading much of the search work to the underlying database. Special constructs and operators were also added to express cyclone queries (Muntz *et al.*, 1995).

Knowledge Discovery and Decision Support Applications The potential of the $\mathcal{LDL}$++ technology in this important application area was clear from the start (Naqvi & Tsur, 1989), when our efforts concentrated on providing the analyst with powerful tools for the verification and refinement of scientific hypotheses (Tsur, 1990a). In our early experiments, the expert would write complex verification rules that were then applied to the data. $\mathcal{LDL}$++ proved well-suited for the rapid prototyping of these rules, yielding what became known as the 'data dredging' paradigm (Tsur, 1990a).

A more flexible methodology was later developed combining the deductive rules with inductive tools, such as classifiers or Bayesian estimation techniques. A prototype of a system combining both the deductive and inductive methods is the "Knowledge Miner" (Shen *et al.*, 1994), which was used in the discovery of rules from a database of chemical process data; $\mathcal{LDL}$++ meta predicates proved very useful in this experiment (Shen *et al.*, 1996).

Other experiments demonstrated the effectiveness of the system in performing important auxiliary tasks, such as data cleaning (Tsou *et al.*, 1993; Sheth *et al.*, 1995). In these applications, the declarative power of $\mathcal{LDL}$++ is used to specify the rules that define correct data. These allow record-by-record verification of data for correctness but also the identification of *sets* of records, whose combination violates the integrity of the data. Finally, the rules are used to clean (i.e., correct) inconsistent data. This capability can either be used prior to the loading of data into the database, or during the updating of the data after loading. This early investigations paved the way for a major research project discussed next focusing on using $\mathcal{LDL}$++ in datamining applications .

Developing Data Mining Applications The results of extensive experiences with an $\mathcal{LDL}$++ based environment for knowledge discovery were reported in (Giannotti *et al.*, 1999; Bonchi *et al.*, 1999). The first study (Giannotti *et al.*, 1999) describes the experience with a fraud detection application, while the second one reports on a marketing application using market basket analysis techniques (Bonchi *et al.*, 1999). In both studies, $\mathcal{LDL}$++ proved effective at supporting the many diverse steps involved in the KDD process. In (Bonchi *et al.*, 1999), the authors explain the rationale for their approach and the reasons for their success, by observing that the process of making decisions requires the integration of two kinds of activities: (i) knowledge acquisition from data (inductive reasoning), and (ii) deductive reasoning about the knowledge thus induced, using expert rules that characterize the specific business domain. Activity (i) relies mostly on datamining functions and algorithms that extract implicit knowledge from raw data by performing aggregation and statistical analysis on the database. A database-oriented rule-based system, such as $\mathcal{LDL}$++, is effective at driving and integrating the different tasks involved in (i) and very effective in activity (ii) where the results of task (i) are refined, interpreted and integrated with domain knowledge and business rules characterizing the specific application.

For instance, association rules derived from market basket analysis are often too low-level to be directly used for marketing decisions. Indeed, market analysts seek answers to higher-level questions, such as "Is the supermarket assortment adequate for the company's target customer class?" or "Is a promotional campaign effective in establishing a desired purchasing habit in the target class of customers?". $\mathcal{LDL}$++ deductive rules were used in (Bonchi *et al.*, 1999) to drive and control the overall discovery process and to refine the raw association rules produced by datamining algorithms into knowledge of interest to the business. For instance, $\mathcal{LDL}$++ would be used to express queries such as "Which rules survive/decay as one moves up or down the product hierarchy?" or "What rules have been effected by the recent promotions" (Bonchi *et al.*, 1999).

The most useful properties of $\mathcal{LDL}$++ mentioned in these studies (Giannotti *et al.*, 1999; Bonchi *et al.*, 1999; Giannotti *et al.*, 2001a) were flexibility, capability to adapt to the analyst's needs, and modularity, i.e., the ability to clearly separate the different functional components, and provide simple interfaces for their integration. In particular, the user defined aggregates described in Section 2.2 played a pivotal roles in these datamining applications since datamining functions (performing the inductive tasks) were modelled as user-defined aggregates which could then be conveniently invoked by the $\mathcal{LDL}$++ rules performing the deductive tasks (Giannotti *et al.*, 2001a). The performance and scalability challenge was then addressed by encoding these user-defined aggregates by means of $\mathcal{LDL}$++ procedural extensions, and, for database resident data, offloading critical tasks to the database system containing the data (Giannotti *et al.*, 2001a).

Lessons Learned The original motivations for the development of the original $\mathcal{LDL}$ system was the desire to extend relational query languages to support the development of complete applications, thus eliminating the impedance mismatch from which applications using embedded SQL are now suffering. In particular, data intensive expert systems were the intended ‘killer’ applications for $\mathcal{LDL}$. It was believed that such applications call for combining databases and logic programming into a rule-based language capable of expressing reasoning, knowledge representation, and database queries. While the original application area failed to generate much commercial demand, other very promising areas emerged since then. Indeed the success of $\mathcal{LDL}$++ in several areas is remarkable, considering that $\mathcal{LDL}$++ is suffering from the combined drawbacks of (i) being a research prototype (rather than a supported product), and yet (ii) being subject to severe licensing limitations. Unless the situation changes and these two handicaps are removed, the only opportunities for commercial deployments will come from influencing other systems; i.e., from system that borrow the $\mathcal{LDL}$++ technology to gain an edge in advanced application areas, such as datamining and decision support systems.

5 Conclusion

Among the many remarkable projects and prototypes (Ramakrishnan & Ullman, 1995) developed in the field of logic and databases (Minker, 1996), the $\mathcal{LDL}/\mathcal{LDL}$++ project occupies a prominent position because the level and duration of its research endeavor, which brought together theory, systems, and applications. By all objective measures, the $\mathcal{LDL}$++ project succeeded in its research objectives. In particular, the nondeterministic and nonmonotonic constructs now supported in $\mathcal{LDL}$++ take declarative logic-based semantics well beyond stratification in terms of power and expressivity (and stratified negation is already more powerful than SLD-NF). The $\mathcal{LDL}$++ system supports well the language and its applications. In particular, the pipelined execution model dovetails with constructs such as choice and aggregates (and incremental answer generation), while the system's open architecture supports tight coupling with external databases, JDBC, and other procedural languages. The merits of the $\mathcal{LDL}$++ technology, and therefore of deductive databases in the large, have been demonstrated in several pilot applications—particularly datamining applications.

Although there is no current plan to develop $\mathcal{LDL}$++ commercially, there remain several exciting opportunities to transfer its logic-oriented technology to related fields. For instance, the new query and data manipulation languages for web documents, particularly XML documents, bear affinity to logic-based rule languages. Another is the extension to SQL databases of the new constructs and non-stratified semantics developed for $\mathcal{LDL}$++: in fact, the use of monotonic aggregates in SQL has already been explored in (Wang & Zaniolo, 2000).

Acknowledgements

The authors are grateful to the referees for many suggested improvements. This work was partially supported by NSF Grant IIS-0070135.

References

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

Ackley, D., et al. (1990) System Analysis for Deductive Database Environments: an Enhanced role for Aggregate Entities, *Procs. 9th Int. Conference on Entity-Relationship Approach*, Lausanne, CH, Oct. 8-10, 1990.

Arni, N., Greco, S., Saccà, D. (1996) Matching of Bounded Set Terms in the Logic Language LDL++, JLP 27(1): pp. 73-87, 1996.

Bancilhon, F., Maier D., Sagiv, Y. and Ullman, J. (1986) Magic Sets and Other Strange Ways to Implement Logic Programs, in *Proc. SIGACT-SIGMOD Principles of Database Systems Conference (PODS)*, pp. 1-16, 1986.

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.

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.

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).

Byrd, L. (1980) Understanding the Control Flow of Prolog Programs, in *Proceedings of the Logic Programming Workshop,* Debrecen, Hungary, pp. 127-138, 1980.

Chimenti, D., Gamboa, R., and Krishnamurthy, R. (1989) Abstract Machine for $\mathcal{LDL}$, *Proc. EDBT Conference*, pp. 271-293, 1989.

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

Erickson, D. (1992) Hacking the Genome, *Scientific American*, April 1992.

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

Gelfond, M. and Lifschitz, V. (1988) The stable model semantics of logic programming, In *Proc. Fifth Int. Conference on Logic Programming*, pp. 1070–1080, 1988.

Giannotti, F., Pedreschi, D., Saccà, D., and Zaniolo, C. (1991) Nondeterminism in deductive databases, In *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, pp.129-141, 1991.

Giannotti, F., Manco, G., Nanni, M., Pedreschi, D. (1998) On the Effective Semantics of Nondeterministic, Nonmonotonic, Temporal Logic Databases, *Computer Science Logic, 12th International Workshop, CSL 1998,* pp. 58-72, Lecture Notes in Computer Science, Vol. 1584, Springer, 1999.

Giannotti, F., Manco, G., Pedreschi, D., Turini, F. (1999) Experiences with a Logic-based knowledge discovery Support Environment, ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, DMKD'99, Philadelphia, USA, May 30, 1999

Giannotti, F., Manco, G., Turini, F. (2001) Specifying Mining Algorithms with Iterative User-Defined Aggregates: A Case Study, *Principles of Data Mining and Knowledge Discovery, 5th European Conference, PKDD 2001*, pp. 128-139, 2001.

Giannotti, F., Pedreschi, D. and Zaniolo, C. (2001) Semantics and Expressive Power of Non-Deterministic Constructs in Deductive Databases, *Journal of Computer and System Science*, Vol. 62, No. 1, pp. 15-42, 2001.

Greco, S., Saccà, D. (1997) NP Optimization Problems in Datalog, *ILPS 1997*, pp. 181-195, 1997.

Han, J., and Kamber, M. (2001) *Data Mining, Concepts and Techniques* Morgan Kaufman, 2001

Hellerstein, J. M., Haas, P. J., Wang., H. J. (1997) Online Aggregation. *Proc. ACM–SIGMOD Conference on Management of Data*, pp. 1711-182, 1997.

Kemp, D., Ramamohanarao, K., and Stuckey, P. (1995) ELS Programs and the Efficient Evaluation of Non-Stratified Programs by Transformation to ELS, In *Proc. Fourth 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), pp. 727-745, 1998.

Krishnamurthy, R. and Naqvi, S. (1988) Non-deterministic Choice in Datalog, In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, 1988.

Lausen, G. Ludäscher, B., May, W. (1998) On Active Deductive Databases: The Statelog Approach, In *Transactions and Change in Logic Databases*, B. Freitag, H. Decker, M. Kifer, A. Voronkov (Eds.), LCNS 1472, Springer, pp. 69–106, 1998.

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

Marek, W., Truszczynski, M. (1991) Autoepistemic Logic, *Journal of ACM*, 38(3), pp. 588–619, 1991.

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.

Muntz, R.R., Shek, E.C. and Zaniolo, C. (1995) Using $\mathcal{LDL}$++ for Spatio-temporal Reasoning in Atmospheric Science, in *Applications of Logic Databases*, R. Ramakrishan (ed.), pp. 101–118, Kluwer, 1995.

Naqvi, S. and Tsur, S. (1989) *A Logical Language for Data and Knowledge Bases*, W. H. Freeman, 1989.

Overbeek, R., Price, M., and Tsur, S., Automated Interpretation of Genetic Sequencing Gels, MCC Technical Report, 1990.

Ong, K., Arni, N., Tomlinson, C., Unnikrishnan, C., Woelk, D. (1995) A Deductive Database Solution to Intelligent Information Retrieval from Legacy Databases, *Proc. Fourth Int. Conference on Database Systems for Advanced Applications*, DASFAA 1995, pp. 172-179, 1995.

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

Przymusinski, T. (1988) 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, Los Altos, CA, 1988.

Ramakrishnan, R., Srivastava, D. and Sudarshan, S. (1992) CORAL-Control, Relations and Logic, *Proceedings of the 18th VLDB Conference*, 1992.

Ramakrishnan, R., Srivastava, D., Sudarshan, S., Seshadri, P. (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), pp. 79–97 (1997).

Saccà, D., and Zaniolo, C. (1990) Stable models and Nondeterminism in Logic Programs with Negation, *Proc. 9th, ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 205–218, 1990.

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

Shen W., Mitbander W., Ong K. and Zaniolo, C, (1994) Using Metaqueries to Integrate Inductive Learning and Deductive Database Technology,, In *AAAI Workshop on Knowledge Discovery from Databases*, Seattle 1994.

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.

Sheth, A.P., Wood, C., Kashyap, V. (1995) Q-Data: Using Deductive Database Technology to Improve Data Quality, in *Applications of Logic Databases*, Raghu Ramakrishnan (ed.), pp. 23-56, Kluwer, 1995.

Shmueli, O., Tsur, S. and Zaniolo, C. (1988) Rewriting of Rules Containing Set Terms in a Logic Database Language (LDL), *Proc. of the Seventh ACM Symposium on Principles of Database Systems*, pp. 15-28, 1988.

Van Gelder, A. (1993) 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.

Tryon, D. (1991) Deductive Computing: Living in the Future, Proc. of the Monterey Software Conference, May 1991.

E. Tsou et al. (1993) Improving Data Quality Via $\mathcal{LDL}$++, ILP'93 Workshop on Programming with Logic Databases Vancouver, Canada, October 30, 1993.

Tsur, S., Olken, F. and Naor, D. (1990) Deductive Databases for Genomic Mapping, Proc. NACLP90 Workshop on Deductive Databases, Austin, Nov., 1990.

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

Tsur S. (1990) Data Dredging, *Data Engineering*, Vol. 13, No. 4, IEEE Computer Society, 1990.

Ullman, J.D. (1989) *Database and Knowledge-Based Systems*, Vols. I and II, Computer Science Press, Rockville, MD, 1989.

Wang, H. and Zaniolo, C. (2000) User Defined Aggregates in Object-Relational Systems, *Proceedings of 16th International Conference on Data Engineering, ICDE'2000,* pp. 135-144, IEEE Computer Society, 2000.

Zaniolo C., Arni N. and Ong, K. (1993) Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}$++ Approach, *Proceedings Third Int. Conference on Deductive and Object-Oriented Databases: DOOD 1993*, pp. 204-221, Springer, 1993.

Zaniolo, C. (1994) A Unified Semantics for Active and Deductive Databases, In *Proceed-*

ings Workshop on Rules in Database Systems, RIDS93, Norman W. Paton, M. Howard Williams (Eds.), pp. 271-287 Springer Verlag, 1994.

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

Zaniolo, C. (1997) The Nonmonotonic Semantics of Active Rules in Deductive Databases, *Proceedings Fifth Int. Conference on Deductive and Object-Oriented Databases: DOOD 1997*, pp. 265-282, Springer, 1997.

Zaniolo, C., Tsur, S. and Wang, H. (1998) $\mathcal{LDL}$++ Documentation and Web Demo, `http://www.cs.ucla.edu/ldl` .

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.* Apt, K.R., Marek, V., Truszczynski,M., Warren, D.S. (eds.), Springer Verlag, pp. 401-424, 1999.

Appendix I: Aggregates in Logic

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

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

A logic-based equivalent for this rule is

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

where `results(avg, Y)` is derived from `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 `d(Y)` into an order-inducing chain.

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

Now, we can define the `cagr` rules to perform the inductive computation by calling the `single` and `multi` rules as follows:

```
cagr(AgName, Y, New) ←    chain(nil, Y), Y ≠ nil, single(avg, Y, New).
cagr(AgName, Y2, New) ←   chain(Y1, Y2), cagr(AgName, Y1, Old),
                          multi(AgName, Y2, Old, New).
```

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

```
results(AgName, Yield) ←   chain(Y1, Y2), cagr(AgName, Y1, Old),
                           ereturn(AgName, Y2, Old, Yield).
results(AgName, Yield) ←   chain(X, Y), ¬chain(Y, _),
                           cagr(AgName, Y, Old),
                           freturn(AgName, Y, Old, Yield).
```

Therefore, we first compute `chain`, and then `cagr` that applies the `single` and `multi` rules to every element in the chain. Concurrently, the first `results` rule produces all the results that can be generated by the application of the `ereturn` rules to the element in the chain. The final returns are instead computed by the second `results` rule that calls on the `freturn` rules once the last element in the chain (i.e., the element without successors) is detected. The second `results` rule 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.