Computational Logic, Vol 3., No.1, Dec. 1996

$\mathcal{LDL}$++: A Second-Generation Deductive Database System
(Extended Abstract)

Carlo Zaniolo
University of California,
Los Angeles, CA., 90024,
zaniolo@cs.ucla.edu

Abstract

The expanding role of intelligent information systems and a new wave of database applications are creating a strong demand for database-centered programming environments. In response to this demand, deductive database systems support complex queries and reasoning, rule-based programming and the integration of databases with knowledge bases. The $\mathcal{LDL}$ and $\mathcal{LDL}$++ systems are the two prototypes produced by a leading research project in this area. This paper focuses on the second system that was shaped by the experience gained with its predecessor and recent research advances in the field. The $\mathcal{LDL}$++ system features a better integration with external databases, a new execution model, and various language extensions to support non-deterministic and non-monotonic programming.

1 Introduction

The design of the $\mathcal{LDL}$++ prototype was inspired by the experience with the $\mathcal{LDL}$ prototype [3], which was completed in 1989 and tested and deployed in many applications [17, 18, 19]. This paper focuses on progress past the $\mathcal{LDL}$ system that has already been described in previous papers. In particular, the $\mathcal{LDL}$ language and its semantics, along with the theory that underlies its design, are described in [9]; a detailed description of the technology and architecture for the first $\mathcal{LDL}$ prototype is presented in [3].

The experience with the first-generation prototype resulted in substantial feedback by users and opportunities for critical re-evaluation of design choices. In the meantime, much technical progress occurred in the deductive database area. The result of these two combined forces was a significant evolution in the design of the second-generation system, $\mathcal{LDL}$++, with respect to the first-generation system, $\mathcal{LDL}$. The most significant improvements can be summarized as follows:

- Open Architecture and integration with SQL systems
- New compilation and execution model

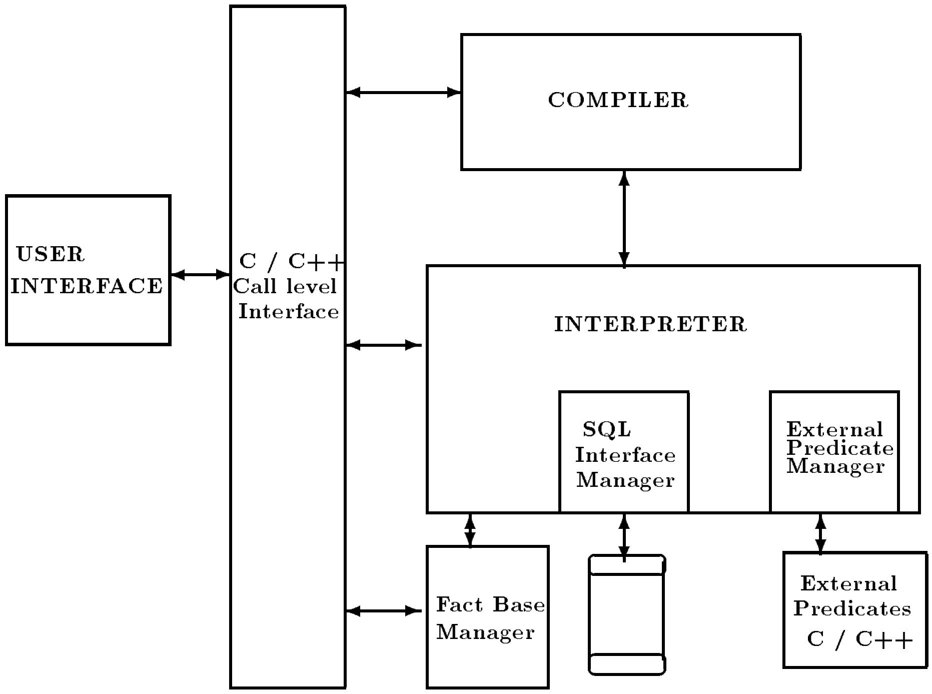

Figure 1: $\mathcal{LDL}$++ architecture.

- Language extensions to support non-monotonic and non-deterministic programming
- New applications.

2 Database Architecture

The most critical decision in building a deductive database system pertains to the usage made of (i) main memory versus secondary store, and (ii) of external database systems. For instance, some deductive database prototypes include a secondary-store based record manager and internal DBMS [14], others rely on external DBMSs as their record managers. The $\mathcal{LDL}$ application experience, particularly in a middle-ware role, confirmed the desirability of an architecture based on (i) an internal (fast-path) database and (ii) a transparent connection with multiple external DBMSs. The internal (light-weight) database optimizes the management and retrieval of $\mathcal{LDL}$++ complex objects, including sets and lists, and of temporary relations derived in the course of the computation. In addition to supporting users' data defined by the schema as internal relations, the interpreter uses the local database to store and manage temporary data sets. The internal database is designed as a virtual-memory record manager: thus internal organization and indexing is optimized for the situation where the working set resides 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 overall architecture of the system is described in Figure 1.

A significant effort was spent in building the External Predicate Manager to provide transparent access from $\mathcal{LDL}$++ to commercial RDBMSs, and the large databases managed by such systems. Thus, rules in a program make no distinction between internal and external relations. Relations from external SQL databases are declared in the $\mathcal{LDL}$++ schema as they were internal relations—with the only difference that the name and type of the SQL server holding the data must also be given for each external relation. As a result, these external resources are transparent to the inference engine and applications can access different databases in a totally uniform way.

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

Example 1 *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
          } ).
```

Rules having as goals predicates based on external SQL relations are translated into equivalent SQL queries that are off-loaded to the SQL server. These queries can express any combination of the following operators: joins, selections, projections, set-differences and set aggregates as well. Also updates on the external database relations are translated and sent to the SQL servers for execution.

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

Example 2 *SQL Generation*

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

The $\mathcal{LDL}$++ compiler will collapse all the body literals of Example 2, and transform them into one SQL node as shown below:

$$\texttt{expensive\_employee(Name)} \leftarrow \quad \texttt{sql\_node(Name)}.$$

The SQL query corresponding to `sql_node` sent to the server is:

```
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 any query written directly in SQL. Rules with negated goals are also supported and implemented via the `NOT EXIST` construct of SQL. The $\mathcal{LDL}$++ interface to SQL 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.

$\mathcal{LDL}$++ performs various optimization steps to maximize the subtasks that are off-loaded to the SQL server. One technique, called *literal collapsing*, re-orders the goals to move SQL-translatable goals together, so that they can be replaced by a single `sql_node`. The *rule compression* technique achieves similar optimization through a common-subexpression rewriting where rules at multiple levels are compressed vertically, and folded into an SQL query.

SQL Aggregation The translation and off-loading of set-aggregates operators in $\mathcal{LDL}$++ was undertook in response to a strong pull of field applications. In the same vein as the select and join operations, aggregate operations on external databases such as *max, min, count, sum* and *avg* are now executed by pushing them off to the server as SQL queries.

To support this, eight new built-in aggregate operators were defined for $\mathcal{LDL}$++. They include *max*, *min*, *count*, *sum*, *avg*, *count_distinct*, *sum_distinct* and *avg_distinct*. These operators can be specified on sets, defined by the <> grouping operator, and hence can be optimized.

Example 3 *Built-in Aggregate Operators*

$$\texttt{exportaverage\_salary(\$Y, AverageSalary)}.$$
$$\texttt{average\_salary(Y, avg} < \texttt{S} >) \leftarrow \quad \texttt{employee(\_, S, Y)},$$
$$\texttt{S} > \texttt{45000}.$$

The generated SQL would be:

```
SELECT avg(employee_0.SALARY)
FROM employee employee_0
WHERE employee_0.SALARY > 45000
      employee_0.MANAGER = 'charlie'
```

Translation of aggregate operators required the realization of rather complex and sophisticated mechanisms. For instance, conditions on the results of aggregate operators have to be expressed using `HAVING` clauses. Then, corresponding variables for grouping must be recognized and replace by a `GROUP BY` clause. The translation process is further complicated by the fact that SQL's `HAVING` and `GROUP BY` constructs have been recalcitrant to standardization, and different database vendors support slight variation of these constructs.

As illustrated by the previous example, literal collapsing will normally be performed when dealing with aggregate operators. However, rule compression requires special treatment that demands complex management of literals and variables related to the aggregated operators. The compiler has to keep track of which variable is bound to which aggregated argument of which relation. We will not further elaborate on these implementation details as it is not within the scope of this paper. However, we will demonstrate the complexity through the following example:

Example 4 *Rule Compression and Literal Collapsing for Aggregate Operators*

```
exportintegrity_rule(Manager,NumberOfEmployee,TotalSalary,Budget).
integrity_rule(Manager,NumberOfEmployee,TotalSalary,Budget) ←
            dept_total(Manager,NumberOfEmployee,TotalSalary),
            department(Manager,NumberOfEmployee,Budget),
            Budget ≥ TotalSalary.
dept_total(Manager,count < X >,sum < S >) ←   employee(X,S,Manager).
```

The program above checks the data integrity between two tables. Here we assume there is a table `department` that has the attribute `Name of Manager`, `Number of Employee` and `Budget`. The rule will fail if the number of employees does not match the number of employees specified in the `department` table or if the total salary of the employees in the department is larger than the budget of the department. This program will require compression of the rule `dept_total` as well as the collapsing of the table `department`. Furthermore, variables such as `NumberOfEmployee` and `TotalSalary` must be substituted appropriately since they are bound to values generated by the aggregate operators. In this example, we have only one level of rule compression. In general, there could be many levels of compression and many stages of literal collapsing. Thus, the compiler has to maintain complex information about how variables, aggregated arguments and database relations are related to each other even though they have been deleted in the rewriting process. Thus, the following SQL statement is generated:

```
SELECT   employee.MANAGER, count(employee.NAME),
         sum(employee.SALARY)
FROM      employee, department
HAVING   department.NUM_EMPLOYEE = count(employee.NAME) AND
         department.BUDGET < sum(employee.SALARY)
GROUP BY employee.MANAGER, department.NUM_EMPLOYEE,
          department.BUDGET
```

Note that `department.NUM_EMPLOYEE` and `department.BUDGET` are specified in the `GROUP BY` clause even though this is unnecessary based on the logical specifications. The reason is that some of the current database systems require that every variable `HAVING` clause must be specified in the `GROUP BY`. The $\mathcal{LDL}$++ system is in fact equipped with a generic SQL interface as well as an object-oriented design that allows easy access to databases from different vendors. To link the system with a specific external database, it is only necessary to write a minimal amount of code to use vendor-specific drivers to handle data conversion and the local SQL dialects. The current release of the $\mathcal{LDL}$++ system can link with Sybase, DB2, Oracle, and ESS [1]. Using ESS, $\mathcal{LDL}$++ can link with additional relational databases and some SQL-like interfaces to O-O databases.

3 Compilation and Execution

The approach followed by the $\mathcal{LDL}$++ compiler is similar of that followed in [3]. Thus, the compiler reads-in programs and constructs the *Global Predicate Connection Graph* (PCG). Then, for each query form, the compiler performs a bound-free analysis on the arguments of the predicates, annotating these predicates with adornments and checking the safety of the queries in the process. The compiler also performs the rewriting of recursive rules using various techniques, including the Magic Sets method [2] and specialized methods applicable to linear programs [20, 24]. These rewriting techniques result in an efficient execution plan for queries. Unification is also performed as part of this compilation process, resulting in the sharing of variables between predicates, to minimize the assignments required for execution.

The execution approach taken by $\mathcal{LDL}$++ is however, different from that of $\mathcal{LDL}$ which transformed the annotated PCG into a C program that was then compiled for later execution. Since the compilation of the large C programs so generated was too slow for the rapid prototyping of applications, the $\mathcal{LDL}$++ compiler takes an approach oriented toward interpretation, as follows. The annotated PCG is translated into a data structure called a LAM (for $\mathcal{LDL}$++ Abstract Machine), whose nodes are instances of C++ classes. The LAM structure is generated dynamically and executed as part of the $\mathcal{LDL}$++ running interpreter. Since the PCG consisted of OR-nodes, representing goals in the rules, and AND-nodes representing rule heads, the basic C++ classes in the LAM are OR-nodes and AND-nodes; then there are several more specialized subclasses. These subclasses include a special OR-node to serve as the distinguished *root* node for the query form, internal relations AND-nodes, external relations AND-nodes, etc.. 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.

[1]ESS, a "Extensible Services Switch" developed in the CARNOT project at MCC, supports RDA interface to several databases on a network

[2]Similar to a C++ virtual function

Some interesting techniques used in the implementation of $\mathcal{LDL}$++ include novel implementation techniques for sets [1] and a stream-oriented execution. Because of space limitations, we will only discuss the second item.

Stream-Oriented Processing $\mathcal{LDL}$++ adopts a lazy evaluation approach (*pipelining*) as its primary execution model. In such a model, a goal $p(X, Y)$ invoked with, say $X = a$ computes only the first value of Y associated with a; and then passes this value to the remaining goals in the rule. Only when these have completed their computation, the next value for Y associated with a is computed. With *eager materialization*, all the pairs $p(X, Y)$ are computed first and then filtered with the post-condition $X = a$.

$\mathcal{LDL}$++'s preference for a pipelined execution is due, at least in part, to usability considerations, since it makes the system more responsive and more interactive. This conclusion was reinforced by the our experience of watching new users executing applications on the $\mathcal{LDL}$ systems. As seconds ticked away and no result was returned, increasingly anxious users wandered whether the system was busy materializing a large answer, or their program had strayed into an infinite loop!

Furthermore, a pipelined computation lends itself better to tracing and debugging and letting the user "who has seen enough" kill the program. Also, it is supportive of various optimizations connected with choice, existential queries and intelligent backtracking, as demonstrated by the next two examples.

Example 5 *Existential Query*

$$\mathtt{query1(A)} \leftarrow \quad \mathtt{A = 0, p(A, \_)}.$$

In an eager computation, `query1(A)` will cause all tuples with `0` in the first argument of `p` to be generated in order to satisfy the goal `p(0, _)`. In the pipelined approach, only the first of these will be generated.

Example 6 *Intelligent Backtracking.*

$$\mathtt{query2(A,B)} \leftarrow \quad \mathtt{b1(A), p(A,B), b2(A)}.$$

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`. An eager approach will instead generate all `B`-values for this `A`, even they cannot satisfy `b2`.

Some of these optimizations were already supported in $\mathcal{LDL}$ [3] which used pipelining in some restricted situations. For recursive predicates however, $\mathcal{LDL}$ followed the usual seminaive fixpoint procedure, that uses the immediate consequence operator for eagerly computing all the immediate consequences [9]. $\mathcal{LDL}$++ instead supports lazy computation of fixpoints, through the *consumer/producer* architecture where the goals calling recursive predicates act as consumers and the recursive clique acts as a producer.

A complication 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), and multiple invocations of a shared node can interfere with each other due to non-reentrant code. Several solutions are possible to this problem, including Prolog's solution of using a stack, and $\mathcal{LDL}$'s solution of source code duplication (to ensure that the PCG is a tree, rather than a DAG). If the consuming predicates have no bound argument, then $\mathcal{LDL}$++ uses the lazy producer approach previously described for the case of the lazy fixpoint. If there is some bound arguments, then the node is duplicated. However, since the nodes are implemented as a C++ classes, this result in the generation of multiple instances of the same classes. Thus only data is duplicated—rather than both code and data as in $\mathcal{LDL}$.

3.1 Language Extensions

A very useful new construct supported in $\mathcal{LDL}$++ is *meta-level predicates*, which were given a first order semantics along the lines described in [10]. Furthermore, $\mathcal{LDL}$++ supports non-stratified negation (and set-aggregates) and supports powerful non-deterministic constructs, providing novel solutions to the semantics problems connected with these concepts. The problem of going beyond stratification represents in fact a difficult research challenge, with contributions and ideas coming from the areas of AI, non-monotonic logic and deductive databases. Among the most important notions, we find the concepts of well-founded models [4] and stable models [5], which provide a declarative semantics for most programs of practical interest. Formal declarative semantics, however, only represents one of the requirements that must be satisfied before non-monotonic programs can be allowed in the recursive rules of the language. A second key requirement is amenability to efficient implementation; the third one is that the resulting programs must be easy to understand. While stratification satisfies all three conditions, general well-founded model semantics, or stable-model semantics, fail the second and third condition. As a result of this situation, the user might have to fall back to procedural programming whenever dealing with problems, such as the Bill-of-Materials requiring non-stratified negation or aggregates (thus the *Nail!* user will probably shift to Glue, while a *CORAL* programmer might use annotations to induce the proper execution strategy in the system.)

In contraposition with the before-mentioned approaches, $\mathcal{LDL}$++ supports very powerful extensions without surrendering the formal logic-based framework. In fact, $\mathcal{LDL}$++ support the `choice` construct for non-deterministic programming and the concept of XY-stratification that allows for negation and set aggregates in recursive rules. The formal semantics of both constructs is based on stable models.

Because of limited space, we will not discuss XY-stratification [21], and concentrate instead on the non-deterministic `choice` construct. The idea of `choice` was introduced in [8] to express non-determinism in a declarative fashion. Thus, a construct such as $\texttt{choice}((\texttt{X}),(\texttt{Y}))$ is used to denote that the functional dependency $X \rightarrow Y$ must hold in the model defining the meaning of this program. For instance the following program, assign to each student an advisor from the same area:

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

Thus, while the result of joining `major` and `faculty` normally results in a many-to-many relation between students and professors (sharing the same area), the `st_ad` relation is restricted to be many-to-one, as a result of the presence of `choice(St, Ad)`. In $\mathcal{LDL}$, this very powerful construct was disallowed in recursion, inasmuch as the functional-dependency based semantics proposed in [8] suffers from technical problems such as a lack of justifiability property and its unsuitability to efficient implementation due to its static nature [15]. These problems were avoided by using instead a semantics based on the use of negation and stable models [6]. For instance, the meaning of the rule above is defined its rewriting into the following non-stratified program:

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

$$\begin{aligned}\texttt{chosen(St, Ad)} \leftarrow\ & \texttt{major(St, Area), faculty(Ad, Area)}, \\ & \neg\texttt{diffChoice(St, Ad)}.\end{aligned}$$

$$\begin{aligned}\texttt{diffChoice(St, Ad)} \leftarrow\ & \texttt{chosen(St,}\overline{\texttt{Ad}}\texttt{), Ad} \neq \overline{\texttt{Ad}}. \\ \texttt{diffChoice(St, Ad)} \leftarrow\ & \texttt{chosen(}\overline{\texttt{St}}\texttt{, Ad), Ad} \neq \overline{\texttt{Ad}}.\end{aligned}$$

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

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

The use of choice in recursive predicates extends the expressive power of the language. Indeed, the following program generates a total order for the elements of a set `d(X)` (`root` is a distinguished new symbol):

$$\begin{aligned}&\texttt{ordered\_d(root, root)}. \\ &\texttt{ordered\_d(X, Y)} \leftarrow\ \ \texttt{ordered\_d(\_, X),}\ \ \texttt{d(Y)}, \\ &\qquad\qquad\qquad\qquad \texttt{choice((X), (Y)),}\ \ \texttt{choice((Y), (X))}.\end{aligned}$$

Therefore using choice and stratified negation it is possible to express all algorithms that can be computed in time which is polynomial in the size of the database [7].

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

Likewise, the notion of XY-stratification [21] a combination of formal non-monotonic semantics, intuitive appeal and amenability to efficient implementations are the ingredients that allow the relaxation of the stratification requirement in $\mathcal{LDL}$++.

4 Conclusions and Future Directions

The realization of the $\mathcal{LDL}$++ prototype, a second-generation system, demonstrates the significant progress recently made by deductive databases. As a result of this progress, the field that was still in its infancy in the 80's can now claim (i) an implementation technology as advanced as that of relational databases, logic programming and AI expert systems, and (ii) theoretical foundations that combine and extend those of the before-mentioned fields is several domains, particularly non-monotonic reasoning. Furthermore, the deployment of the $\mathcal{LDL}$++ prototype in various applications has much contributed to proving the unique advantages offered by deductive databases in several domains [18, 19, 12].

Future evolution of the field could be shaped by various advances that have taken place after the completion of the $\mathcal{LDL}$++ prototype. Indeed, recent progress makes it possible to have a system that integrates active database rules with deductive rules, and supports in a unified framework temporal reasoning, spatial reasoning and reasoning in the presence of uncertainty. Furthermore, important application areas are now emerging for this technology, including middleware systems [11] and knowledge discovery [16].

Ideally, this combination of application pull and technology push will soon produce a new-generation of deductive database systems that take us beyond the current $\mathcal{LDL}$++ and its second-generation technology.

References

[1] Natraj A., S. Greco and D. Saccá "Set-Term Matching in Logic Programming" in *Proc. 4th International Conference on Database Theory, ICDT 92, Berlin, Germany, pp. 436-449*

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

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

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

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

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

[7] Giannotti, F., S. Greco, D. Saccà and C. Zaniolo, "Programming with Non-determinism in Deductive Databases" *Annals of Artificial Intelligence and Mathematics*, Vol. 19, No, 3/4, April 1997.

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

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

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

[11] Ong, K., N. Arni, C. Tomlinson, Unnikrishnan, and D. Woelk, "A Deductive Database Solution to Intelligent Information Retrieval from Legacy Databases", *Proc. of 4th International Conference on Database Systems for Advanced Applications*, April, 1995.

[12] R. Ramakrishnan. *Applications of Logic Databases.* Kluwer Academic Publisher, 1995.

[13] R. Ramakrisnhan, D. Srivastava, and S. Sudanshan, "CORAL-Control, Relations and Logic," *Proceedings of the 18th VLDB Conference*, 1992.

[14] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, James Harland, "The Aditi Deductive Database System," VLDB Journal Vol. 3, No.2, pp. 245-288 (1994)

[15] Saccà, D., and Zaniolo, C., "Stable models and non determinism in logic programs with negation", *Proc. 9th, ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 205-218, 1990.

[16] Shen, W., K.Ong, B. Mitbander and C. Zaniolo, "Metaqueries for Data Mining," Chapter 15 of *Advances in Knowledge Discovery and Data Mining*, U. M. Fayyad et al (eds.), MIT Press, 1996.

[17] Tryon, D. "Deductive Computing: Living in the Future," Proc. of the Monterey Software Conference, May 1991.

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

[19] Tsur, S., "Data Dredging," *Data Engineering*, Vol. 13, No. 4, IEEE Computer Society, Dec. 90.

[20] Ullman, J.D., *"Database and Knowledge-Based Systems*, Vols. I and II, Computer Science Press, Rockville, Md., 1989.

[21] Zaniolo, C., N. Arni, and K. Ong, "Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}$++ Approach," *Procs. Third Int. Conference on Deductive and Object-Oriented Databases*, Dec. 6-8, 1993, Scottsdale, Arizona.

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

[23] Zaniolo, C., "A Unified Semantics for Active and Deductive Databases,", Procs. Workshop on Rules In Database Systems (RIDS93), Springer Ferlag 94.

[24] Zaniolo, C., S. Ceri, C. Faloutsos, R. Snodgrass, V. Subrahamanian and R. Zicari, *"Introduction to Advanced Database Systems,"* Morgan Kaufmann, 1997.