Deductive Databases–Theory Meets Practice

Carlo Zaniolo

MCC
3500 West Balcones Center Drive
Austin, Texas 78759
USA

Abstract

Deductive Databases are coming of age with the emergence of efficient and easy to use systems that support queries, reasoning, and application development on databases through declarative logic-based languages. Building on solid theoretical foundations, the field has benefited in the recent years form dramatic advances in the enabling technology. This progress is demonstrated by the completion of prototype systems offering such levels of generality, performance and robustness that they support well complex application development. Valuable know-how has emerged from the experience of building and using these systems: we have learned about algorithms and architectures for building powerful deductive database systems, and we begin to understand the programming environments and paradigms they are conducive to. Thus, several application areas have been identified where these systems are particularly effective, including areas well beyond the domain of traditional database applications. Finally, the design and deployment of deductive databases has provided new stimulus and a focus to further research into several fundamental issues. As a result, the theory of the field has made significant progress on topics such as semantic extensions to Horn logic and algorithms for compilation and optimization of declarative programs. Thus, a beneficial interaction between theory and practice remains one of the strengths of Deductive Databases as the field is entering the '90s and the age of technological maturity.

1 Background

Deductive Databases are coming of age with the emergence of efficient and easy to use systems that support *queries, reasoning*, and *application development* on databases through *declarative logic-based* languages.

Interest in the area of Deductive Databases began in the '70s, with most of the early work focusing on establishing the theoretical foundations for the field. An excellent review of this work and the beneficial impact that it had on various disciplines of computing, and the database area in particular, is given in [GMN]. Throughout the '70s and the first part of the '80s, concrete

system implementations of these ideas were limited to few ground breaking experiments [Kell]. This situation contrasted quite dramatically with the significant system-oriented developments that were taking place at the same time in two fields very close to deductive databases. The first field was relational databases, where systems featuring logic-based query languages of good performance, but limited expressive power, were becoming very successful in the commercial world. The second field is Logic Programming, where successive generations of Prolog systems were demonstrating performance and effectiveness in a number of symbolic applications, ranging from compiler writing to expert systems.

A renewed interest in deductive database systems came about as a result of the flare-up of attention and publicity generated by the idea of Fifth Generation Computing. It was realized that the rule based reasoning of logic, combined with the capability of database systems of managing and efficiently storing and retrieving large amounts of information could provide the basis on which to build the next-generation of knowledge base systems. As a result, several projects were started that focused on extending Prolog systems with persistent secondary-based storage management facilities [RaSh] or on coupling Prolog with relational databases [JaCV, KuYo, Li, CeGW]. Several commercial systems are now available that support the coupling of SQL databases with Prolog or expert system shells. In particular, is the system described in [Boc, LeVi] provides close integration between Prolog and Database facilities, and smart algorithms for supporting recursive queries against the database.

Yet several other researchers were critical of the idea of using Prolog as a front-end to relational databases. In particular, it was noted that the sequential left-to right execution model of Prolog was a throw-back to navigational query languages used before relational systems. In relational systems, the user is primarily responsible for correct queries, and the system takes care of finding efficient sequencing of joins (query conjuncts), thus optimizing navigation through the database—a special module called the *query optimizer* sees to that [Seta]. In Prolog, instead, the programmer must carefully select the order of rules and of goals in the rules, since the correctness, efficiency and termination of the program depend on it. A second problem follows from the fact that efficient Prolog implementations are based on a abstract machine (WAM) and features (pointers) that rely on the assumption that data resides in main memory rather than secondary store [War]. Thus a number of research projects opted for an approach that builds more on extensions of relational database technology than on adaptations of Prolog technology. While several of these projects limited their interests to extending query languages with specific constructs such as rules and recursion, projects such as NAIL! [Meta] and $\mathcal{LDL}$ [Ceta1, NaTs] feature declarative languages of expressive power comparable to Prolog. This paper recounts and summarizes the author's experience in designing, developing and deploying the $\mathcal{LDL}$ system.

2 Overview

The motivation for designing and building the $\mathcal{LDL}$ system was twofold:

- To provide support for advanced database applications, with a focus on expert systems and knowledge based applications.
- To provide better support for traditional database applications by integrating the application development and database queries into one language—thus solving the impedance mismatch problem.

A serious problem with current database applications is due to the limited power of languages such as SQL, whereby the programmer has to write most of the application in a procedural language with embedded calls to the query language. Since the computing paradigm of a procedural language, such as COBOL, is so different from the set-oriented declarative computation model of a relational language, an *impedance mismatch* occurs that hinders application development and can also cause slower execution [CoMa]. Realization of this problem has motivated a whole line of database research into new languages, commonly called *database languages* [BaBu]. The typical approach taken by previous researchers in database languages consisted in building into procedural languages constructs for accessing and manipulating databases [Sch77, RoSh]. Persistent languages, where the database is merely seen as an extension of the programming language, represent an extreme of this emphasis on programming languages. In a sharp departure from these approaches, $\mathcal{LDL}$ focuses on the query language, and extends it into a language powerful enough to support the development of applications of arbitrary complexity. Rather than extending current database query languages such SQL, however, $\mathcal{LDL}$ builds on the formal framework of Horn clause logic—a choice that had less to do with the well-known shortcomings of SQL, than with the influence of Prolog (a language based on Horn clause logic). In fact, we were impressed with the fact that this rule-based language was effective for writing symbolic applications and expert applications as well as being a powerful and flexible database query language [Zan1].

A closer examination on why Horn clauses represent such a desirable rule-based query language reveals the following reasons:

- Horn Clauses are akin to domain relational calculus [Ull], which offer two important advantages with respect to tuple calculus on which languages such as SQL are based—but the two calculi are known to be equivalent in terms of expressive power. One advantage is that domain calculus supports the expression of joins without explicit equality statements; the other is that lends itself to the visualization of queries —both benefits vividly demonstrated by QBE [Ull].
- Horn clauses support *recursion* and *complex terms* (through function symbols) thus eliminating two important limitations of relational query languages and systems.
- Horn clauses have a declarative semantics based on the equivalent notions of minimal model and least fixpoint [Llo, vEKo].
- Horn clauses can also be used effectively as a navigational query language.

As the last two points suggest, Horn clauses can be used effectively as either a declarative query language or navigational one [Zan1]. In the declarative interpretation of Horn Clauses, the order of goals in a rule is unimportant (much in the same way in which the order of conjuncts in a relational query is immaterial). The navigational interpretation of Horn clauses follows from the operational semantics of Prolog. Under this interpretation, goals are executed respectively in a left-to-right order, and the programmer is basically entrusted with the task of using this information to write terminating and efficient programs. For instance, when the goals denote database relations, the order defines a navigation through the database records; the programmer

must carefully select the best navigation, e.g., one that takes advantage of access structures and limits the size of intermediate results.

A most critical decision in designing $\mathcal{LDL}$ was to follow the path of relational systems and build on the declarative semantics, rather than on the operational interpretation of Horn clauses. This approach was considered to be superior in terms of data independence and ease of use. Indeed this approach enables the user to concentrate on the meaning of programs, while the system is now entrusted with ordering goals and rules for efficient and safe executions. A further step toward declarative semantics was taken by freeing the user from the concern of whether forward chaining or backward chaining should be used in executing a set of rules. Current expert system shells frequently support only one of these two strategies; when they provide for both, they leave to the programmer the selection of the proper strategy for the problem at hand and its encoding as part of the program. In $\mathcal{LDL}$, the actual implementation is largely based on a forward chaining strategy which is more suitable for database applications [Ceta2]. But the compiler has also the capability of using rule rewrite methods, such as the magic set method or the counting method [BMSU, SaZ1, SaZ2], to mimic backward chaining through a bottom-up computation. Thus the $\mathcal{LDL}$ user is also provided automatically by the system with the functionality and performance benefits of backward chaining. This substantial progress toward declarative programming represents one of the most significant contributions to the technology of rule-based systems brought about by the research on deductive database systems in the recent years.

Another major area of progress for deductive databases is that of semantics. Indeed many other constructs beyond Horn clauses are needed in a language such as $\mathcal{LDL}$ to support application development. In particular, $\mathcal{LDL}$ includes constructs supporting the following notions:

- Negation [ApBW, Naq, Prz1],
- Sets, including grouping and nested relations [BNST, ShTZ],
- Updates [NaKr, KNZ]
- Don't-care non-determinism [KrN1].

Most of these constructs (excluding set terms) are also in Prolog—they were added because they were needed for writing actual applications. But, in Prolog, their semantics is largely based on Prolog's operational model. Therefore, a major challenge of the $\mathcal{LDL}$ research was to define a formal declarative semantics for these constructs, in a way that naturally extends the declarative semantics of Horn clauses. The problem of extending the power of declarative logic is in fact the second main area of recent advances promoted by research in deductive databases. Of particular interest is the fact that many open problems in knowledge representation and non-monotonic reasoning have been given a clearer definition, and in some cases brought close a solution by these new advances [MaSu, Prz2]

The combined challenge of designing a powerful and expressive language, with declarative semantics, and efficient techniques for compilation and optimization describes the whole first phase of $\mathcal{LDL}$ research. This began in mid 1984, and culminated in the implementation of the first prototype at the end of 1987. This prototype compile $\mathcal{LDL}$ into a relational algebra based language FAD for a parallel database machine [Bor]. Rule rewriting methods, such as magic set and counting,

were used to map recursive programs into equivalent ones that can be supported efficiently and safely by fixpoint iterations [BMSU, SaZ1, SaZ2]. A description of this system is given in [Ceta1].

The implementation of the first $\mathcal{LDL}$ prototype confirmed the viability of the new technology, but did little to transfer this technology from the laboratory to actual users, since FAD is only available on an expensive parallel machine. Seeking a better vehicle for technology transfer, a new prototype system was designed with the following characteristics:

- Portability,
- Efficiency,
- Open System Architecture.

This effort produced a portable and efficient $\mathcal{LDL}$ system under UNIX, called *SALAD*. [1] This implementation assumes a single-tuple, get-next interface between the compiled $\mathcal{LDL}$ program and the underlying fact manager (record manager). This provides for more flexible execution modes than those provided by relational algebra [Ceta1, Zan1]. The new design yields better performance, since the optimizer can now take advantage of different execution modes, and the compiler can cut out redundant work in situations where intelligent backtracking or existential optimization can be used [CGK2, RaBK]. *SALAD* includes a fact manager for a database residing in virtual memory that supports efficient access to the complex and variable record structures provided in $\mathcal{LDL}$. By using *C* as an intermediate target language and an open system architecture, *SALAD* ensures portability, support for modules, and for external procedures written in procedural languages—including controlled access by these routines to internal *SALAD* objects.

The *SALAD* prototype was completed in November 1988, and has undergone improvements and extensions during 1989. By the end of 1989, the system includes a fully functional optimizer, a powerful symbolic debugger with answer justification capability and an X-windows interface. The availability of *SALAD* led to the writing of significant applications and to the emergence of an $\mathcal{LDL}$ programming style. It was found that, in addition to supporting well database applications, $\mathcal{LDL}$ is effective as a rule-based system for rapid prototyping of applications in the *C* environment. Also in 1989, a complete description of the $\mathcal{LDL}$ language with sample applications appeared at bookstores [NaTs], and the first executable copies of *SALAD* were given to universities for experimentation.

I interpret these events as signs of a maturing technology. But, in the end, only the level of satisfaction experienced by users with *SALAD* or similar systems can confirm or disprove my claim that deductive databases are coming of age. To promote this goal, however, this paper will summarize the highlights of my experience with the $\mathcal{LDL}$ system, hoping that the readers will be enticed to experiment with it and then become enthusiastic users of the system. Therefore, the paper focuses on the functionality and usability aspects of the system. The reader interested in the architecture and enabling technology is referred to a recent overview [Ceta2]

3 Declarative Programming and Debugging

A declarative semantics for rules offer several advantages over the operational one, including the following ones:

[1] SALAD—System for Advanced Logical Applications on Data.

- Naturalness,
- Expressive Power,
- Reusability and Data Independence.

Frequently, the most natural definition of a programming object is inductive. For instance, the following $\mathcal{LDL}$ program defines all the integers between zero and K, using Peano's inductive definition (zero is an integer and if J is an integer, so is J+1).

```
int(K,0).
int(K,J) ← int(K,I), I< K, J = I+1.
```

The $\mathcal{LDL}$ compiler has no problem turning this definition into an efficient fixpoint iteration. This pair of rules, or any one obtained by scrambling the order of their goals, cannot be supported by any interpreter or compiler implementing the backward chaining strategy. For instance, in Prolog the user must be go through some interesting contortions to recast this program to fit the operational model.

As the next example, consider the situation where there is a binary tree of atoms. For instance a tree with leaves a and b will be represented by the complex term `tree(a, b)`. Associated with the leaf nodes of a tree there is a weight represented by facts such as

```
node(a, 1).
node(aa, 2).
node(ab, 3).
```

The weight of a tree is inductively defined as the sum of the weights of its two subtrees. We want now to define all the trees with weight less than a certain M. Immediately from these definitions we derive the following rules.

```
w(N, W, M) ← node(N, W), W < M.
w(tree(T1,T2), W, M) ← w(T1,W1), w(T2, W2), W = W1+W2, W < M.
```

This simple definition is not implementable with backward chaining and, unlike the previous example, we do not know of any set of rules that will support this predicate well in Prolog (assuming that the weights of the nodes are not known before hand). While forward chaining is the preferred strategy for these two examples, there are many situations where backward chaining is instead the only reasonable strategy. For instance, say that a tree is at hand and its weight must be computed, as per the the following query goal (where 10000 denotes a value high enough not to be a factor in the computation).

? w(tree(aa,tree(a,ab)), X, 10000).

In this situation, the $\mathcal{LDL}$ compiler simply mimics backward chaining by the use of a rewriting method—the efficient counting method in this particular case [SaZ3]. What is most important here is that the program has not changed. The same program works for different situations and the

compiler/optimizer takes care of matching the actual execution method to the problem at hand. The final result is a level of reusability of programs that is well beyond that of Prolog programs. The elimination of cuts, replaced by the `choice` and `if-then-else` constructs, is also very beneficial in terms of reusability [Zan1]. The concept of reusability for database programs is an extension of the notion of data independence, defined as the ability of queries and applications to survive changes in the database physical organization. In relational databases the key instrument in delivering data independence is the optimizer. In $\mathcal{LDL}$ the optimizer ensures data independence, reusability of code and economy of programming, since the user can call the same module or predicate with different set of bindings.

The previous example was inspired by an Alkane Molecules generation problem [Tsur] that was first proposed to illustrate the power of functional languages. The same problem was quite easily formulated in $\mathcal{LDL}$ due to the ability of expressing inductive definitions and to the ease of checking equivalent structures while avoiding cyclic loops, discussed next. Semantically the structures previously discussed are unordered trees. Thus, a given tree is equivalent to those obtained by recursively exchanging the left subtree with the right one. Equivalence can be expressed by following set of rules:

```
eq(T, T).
eq(tree(T1,T2), tree(T2,T3)) ← eq(T3, T1).
```

Thus two trees are equivalent, if they are equal or if their subtrees have been exchanged and possibly replaced with equivalent ones. The problem is that the composition of several exchanges can return the original structure, and the SLD-resolution will cycle. Thus in Prolog the programmer has to carry around a bag of previous solutions and check for cycles at the cost of inefficiency of programming and execution. In $\mathcal{LDL}$ instead, the system can deal with cycles automatically and efficiently. This feature is particularly important in situations involving negations, since it is the key to a complete realization of stratified negation which avoids the floundering problem of negation by failure [Prz1, Llo].

One of the most interesting aspects of programming with a declarative language is debugging. Any trace-based debugger would be a little use in $\mathcal{LDL}$ since the optimizer rearranges the rules to a point that they do not resemble the original program. On the other hand, the logical nature of the system makes it possible to explain and justify the answers, and thus support a truly logical debugger. The current $\mathcal{LDL}$ system provides logical debugging and answer justification capabilities as sophisticated as those of any expert shell or rule-based system available today.

The conceptual basis for the logical debugger consists in combined `why` and `whynot` explanation capabilities, whereby the system carries out a conversation with the user explaining why a certain answer was returned, and why another was not returned.

Thus to a user asking

```
why eq(tree(a, tree(b,c)), tree(a, tree(c,b)).
```

the system will return the instantiated rule that produced it:

```
eq(tree(a, tree(b,c)), tree(a, tree(c,b))) ← eq(tree(c,b),tree(b,c)).
```

If the user is still not convinced and ask

```
eq(tree(c,b),tree(b,c)).
```

then the system returns the unit clause `eq(c,c)`.

In the whynot interaction, the user asks an explanation on why some tuple was not returned. This capability is needed for supporting why answers in rules with negation, and yields a level of reasoning about programs which is not possible with traditional debuggers. For instance, if the user ask the question

```
whynot eq(tree(tree(a,b) ,c)), tree(tree(b,a), c))).
```

then, the system request the user to point out the rule that should have produced this answer. When the user does so, the system tries to instantiate this rule. In our example, the user will probably point out the second rule; then, the system reports back that no instantiation is possible since T2 cannot be unified with both `tree(a,b)` and `tree(b,a)`. That identifies the problem: the given rules do not capture the correct notion of unordered tree.

This was not a contrived example: I had actually written the program above, and, in a attempt to save a rule, combined the switching of arguments and the recursive equivalence of the subtrees into one rule. Only through the debugger I was able to recognize my mistake (which can be fixed by either adding a new eq goal or an additional rule). Exploring $\mathcal{LDL}$ programs with the logical debugger is indeed a very interesting experience—further enhanced by an X-window based visualization capability.

4 Open System Architecture

An $\mathcal{LDL}$ program calling a graphic routine or a windowing system calling an $\mathcal{LDL}$ program are two concrete examples that motivated the open system architecture of the system. This architecture also follows from the realization that the combination of a high-level declarative language with a procedural language frequently offers the greatest flexibility and effectiveness in actual applications. Typical situations where this bilingual programming paradigm is useful are as follows:

- *Building on existing software.* Existing libraries are often at hand for performing specific tasks, including computation-intensive operations such as graphics or Fast Fourier Transforms. The natural solution consists in importing these routines into $\mathcal{LDL}$ applications while preserving $\mathcal{LDL}$ amenities such as safety and optimization.
- *Rapid prototyping and hot spot refinement.* As many rule-base systems, $\mathcal{LDL}$ is a good vehicle for the rapid prototyping of applications. Large applications can be easily developed and modified until their functionality and behavior satisfy specifications and clients' requirements. Once this validation is completed, the programmer can turn his attention to the performance problem by identifying the *hot spots*. These are predicates or segments of the $\mathcal{LDL}$ program that, because of taking too long to execute and being executed very often, slow down the execution of the whole program. Then the programmer can re-code these hot spots for efficiency, using a procedural language, such as C.

- *Extensibility.* It is often convenient to use higher order predicates or metalevel predicates. These are not provided as part of $\mathcal{LDL}$; however they can be written as external procedures, say in C. Since external procedures can be given access to the internal objects of *SALAD* and can be made behave exactly as $\mathcal{LDL}$ predicates, this becomes a very effective way to add new built-ins.

Support for an open architecture required the following components to be provided [CGK1]:

1. $\mathcal{LDL}$ language extensions to allow external predicates,
2. An optimization strategy and optimizer extensions to deal with external predicates,
3. Run time interfaces between $\mathcal{LDL}$ and external procedures.

There are actually two kinds of external procedures recognized by $\mathcal{LDL}$. The first kind are *external functions*, which are traditional procedures written in languages such as *C* or FORTRAN that are simply imported by statements such as:

```
import strlen($Str:  string, Len:integer) from C.
```

Then an $\mathcal{LDL}$ rule to select strings of length greater than 80 can be written as follows:

```
long_atom(Str) <- strlen(Str , Len), Len > 80.
```

The second kind of external procedures are *external predicates* which, although written in languages such as *C* or FORTRAN, behave as $\mathcal{LDL}$ predicates: they can fail or return more than one solution. External predicates can manipulate all internal *SALAD* objects, including sets, complex terms and relations. Thus, when invoking an external predicate, the calling $\mathcal{LDL}$ program initializes a temporary relation which is passed to the external procedure. This adds the computed results to the relation, and returns control to $\mathcal{LDL}$ that reads the tuples of this relation as from any other internal relation. Likewise, by creating a temporary relation and calling an $\mathcal{LDL}$ precompiled module, conventional programs can also call $\mathcal{LDL}$. Of course, an impedance mismatch between the procedural language and the underlying system could be a problem for the last situation. This is not the case for the previous situations, where the external module becomes completely integrated into $\mathcal{LDL}$.

The main problem with an open architecture, is to have the optimizer "understand" external predicates and use them effectively in optimization. The first problem, dealing with safety, is directly handled through the notion of finiteness constraint, whereby the finiteness of an argument implies the finiteness of others. For instance in the previous `strlen` relation, the finiteness of the first argument implies that of the second. This information is directly extracted from the `import` statement and used by the optimizer in creating a safe ordering of goals. In order to predict and minimize the execution cost, the optimizer uses descriptors characterizing the selectivity, fan-out and cost of the external predicates. The user has to supply this information that can be deduced either from known properties of the external procedure, or from some experimental runs [CGK1]

An interesting example of the benefits offered by this architecture, is the problem of coupling $\mathcal{LDL}$ with an SQL database. Basically, precompiled SQL modules can simply be imported as externals into $\mathcal{LDL}$. The $\mathcal{LDL}$ system views each precompiled query as another computed relation. Once the usual information about finiteness constraints, cost, selectivity and fan-out is given, the system deal with it without any further complication (at least in those SQL systems where this

information is easy to access). The simplicity of this task contrasts with the challenges encountered by various projects coupling Prolog with SQL [JaCV, Boc]. A more interesting problem, which we are now investigating, consists in taking a pure $\mathcal{LDL}$ program using SQL schema relations and translate it into a mixure of $\mathcal{LDL}$ rules and SQL statements. This involves translation of segments of $\mathcal{LDL}$ programs into equivalent SQL queries, and extensions to the $\mathcal{LDL}$ optimizer to determine optimum load sharing between the $\mathcal{LDL}$ front-end and the SQL back-end.

4.1 $\mathcal{LDL}$ Applications

In the end, the utility of the $\mathcal{LDL}$ technology can only be assessed through application development. We first experimented with traditional applications, such as parts explosion, inventory control and job shop scheduling, which are currently implemented by a procedural application program with embedded query calls. Our experience with these applications, has been uniformly positive: we found them easy to write and maintain and extend using $\mathcal{LDL}$. As a result, we moved to more advanced applications, in areas beyond those of traditional DBMSs. Next, we discuss two new areas of particular interest, *data dredging* and *harnessing software*.

4.1.1 Data Dredging

The paradigm that we will describe in this section includes a large class of scientific and engineering problems. The source of the data is typically a large volume of low-level records, collected from the measurement or the monitoring of some empirical process or a system simulation. The objective is to ascertain whether this data lends support to certain abstract concepts where, conceptually, *the level of abstraction of the concepts may be far-removed from the level at which the data was collected.* The procedure adopted to meet the objective is as follows:

1. Formulate hypothesis or concept;
2. Translate (1) into an $\mathcal{LDL}$ rule-set and query;
3. Execute query against the given data and observe the results;
4. If the results do not verify or deny (1) then, reformulate and goto (2); otherwise exit.

Obviously, the decision to exit the process is entirely subjective and is decided by the programmer. At this stage he/she may have either decided that the concept is now properly defined or, that the data does not support this concept and that it should be abandoned or tried out with different data. The use of $\mathcal{LDL}$ over procedural languages offered the advantage of supporting the formulation at a more abstract level where the "iteration time" through reformulations is significantly shortened. With respect to existing database query languages, $\mathcal{LDL}$ supported a more natural expression of higher and higher levels of abstraction via rules, and the ease of incorporating efficient C-based routines for the filtering and preprocessing of low-level data—a demonstration of the two languages programming paradigm. These benefits were observed in experiments with data dredging in two different domains: computer system performance evaluation and scientific data analysis in the area of Molecular Biology. The first application [NaTs] involved the formulation of the "convoy" concept in a distributed computing system. Intuitively, a convoy is a subset of the system entities (processes, tasks) that move together for some time from one node to the other in the network

of processors and queues. The recorded data is low-level and consists of arrival/departure records of individual entities at certain nodes. The convoy concept was defined in $\mathcal{LDL}$ using a small set of rules, and actual instances were detected in the simulation data that were used. The second instance of data dredging—performed in collaboration with researchers from the Harvard Medical School and the Argonne National Laboratories—involves the identification of DNA sequences from (very) low-level, digitized autoradiographs, that record the results of the experiments that are performed in the sequencing of the *E.Coli* bacteria [GENE88]. Again, the task is to extract the definitions for the four DNA bases *A,C,G,T* from this low-level, noisy and often imperfect data. Thus, the interpretation proceeds in two phases:

1. An *alignment phase* during which the raw, digitized data is brought to a common base value and smoothed, using standard signal-processing techniques.

2. An *interpretation phase* during which the aligned data is interpreted, using a set of domain-specific heuristics.

This is a good example of the use of the dual language programming paradigm, since the first phase is best supported by procedural routines, mostly library ones, and the second part requires a high-level rule-based language. In fact, a large number of heuristics need to be applied in this case and the use of $\mathcal{LDL}$ has the additional advantage that it is simple to add special definitions, that need to be used within narrow contexts, to the general definitions. It is thus relatively simple to add additional knowledge to the system as the experience with its use increases. The problem just described can also be viewed as that of writing an expert application to do gel-interpretation. However, the approach is more data-driven than in a typical expert application: the focus has been on extracting knowledge from data, rather than capturing human expertise in the area.

4.1.2 Harnessing Software

We mentioned that external *C* procedures can be used in the definition of $\mathcal{LDL}$ programs. In the $\mathcal{LDL}$ context, these are regarded as evaluable predicates. While normally we expect the use of external code to be the exception rather than the rule (reserved for special purposes e.g., graphical routines), we can think of situations that lay at the other extreme: the bulk of the software is written in standard, procedural code and only a small fraction of it is rule-based and encoded in $\mathcal{LDL}$. In this situation the rule-set forms the "harness" around which the bulk of the code is implemented. The rule portion forms a knowledge base that contains:

1. The definition of each of the C-module types used in the system.

2. A rule set that defines the various ways in which modules can be combined: inheritance and export/import relationships between modules, constraints on their combinations, etc.

The advantage of this organization becomes apparent in information systems where most new service requests can be supported by building on a Lego-set of basic, reusable modules. The knowledge base and rule-based harness externally encode the logic of module interaction and subsets of instances of the existing module types can now be recombined, subject to the rule-restrictions, to support different task-specifications. An added advantage is that each of the individual module-types can be verified using any of the existing verification methods and their global behavior is

controlled by the rule-set. We are currently experimenting with such an application in the domain of banking software.

5 Looking Ahead

Coming of age also means having to accept the limitations of reality. Foremost among these is that the novelty of a technology and the soundness of the underlying theory will not ensures the acceptance a new system or, even less, or a new paradigm for application development. For acceptance, there must be commercial potential, and obvious benefits to the users— the theory must meet practice. Perhaps the most significant aspect of $\mathcal{LDL}$ research is a serious determination of bridging the gap between theory and practice. This is demonstrated by the fact that a team of six to eight people with a wide spectrum of interests and backgrounds—from very theoretical ones to very applied ones—closely collaborated during the last five years in the design and implementation effort. The result is a system that supports the declarative semantics of $\mathcal{LDL}$ and its underlying theory completely and efficiently.

An healthy interaction between theory and practice remains a distinctive mark of some of the most recent developments in $\mathcal{LDL}$. On the theory side, the focus is still on defining key semantic concepts, in areas such as negation and non-monotonic logic [SaZ3], higher order extensions [KrN2], and support for object identity and inheritance [Zan3, BNS]. In the longer term, we expect many of these concepts will become an integral par of the next generation of deductive database systems. The shorter term focus, however, is on deploying the current technology and on developing application areas particularly suitable for deductive databases. Among these we find the area of rapid prototyping and development development of applications—an important area poorly served by existing systems. In fact, the most popular use of rapid prototyping relies on SQL, often enhanced with 4GL facilities [CoSh, DM89, Gane]. Because of its power, open architecture, and support of Knowledge based programming, $\mathcal{LDL}$ can be used in this role much more effectively than SQL, which has obvious the functionality limitations. To further extend the capabilities of the $\mathcal{LDL}$ system in this domain, we are enhancing its environment, (e.g., by providing a standard interface to SQL databases) and addressing the ease-of-use problem. In terms of ease of use, the challenge is not doing better than SQL. Standard relational queries are normally easier to express in $\mathcal{LDL}$ than in SQL. The real research challenge is make it easier and more natural to express the much more complex applications which can now be written in $\mathcal{LDL}$; e.g., those which require complex terms and non-linear recursion. We are currently exploring with the use of visualization, since Horn clauses provide a good conceptual basis on which to support visual programming and debugging. Clearly the conceptual task of solving a complex problem will remain complex; but we expect that the task of coding or debugging the solutions can be greatly facilitated by these interfaces. Our first experience with a visual debugger is encouraging in this respect.

Acknowledgments

The author would like to recognize the following persons for their contribution to the $\mathcal{LDL}$ project: Brijesh Agarwal, Danette Chimenti, François Bancilhon, Ruben Gamboa, Fosca Giannotti, Charles Kellogg, Ravi Krishnamurthy, Tony O'Hare, Shamim Naqvi, Kayliang Ong, Oded Shmueli, Leona Slepetis, Carolyn West, and, last but not least, Shalom Tsur on whose work the section on Applications is based.

References

[ApBW] Apt, K., H. Blair, A. Walker, "Towards a Theory of Declarative Knowledge," in Foundations of Deductive Databases and Logic Programming, (Minker, J. ed.), Morgan Kaufman, Los Altos, 1987.

[BaBu] Bancilhon, F. and P. Buneman (eds.), "Workshop on Database Programming Languages," Roscoff, Finistere, France, Sept. 87.

[BNS] Beeri, C., R. Nasr and S.Tsur, "Embedding *psi*-terms in a Horn-clause Logic Language", *Procs. Third Int. Conf. on Data and Knowledge Bases—improving usability and responsiveness*, Jersualem, June 28-30, pp. 347-359, 1989

[Bor] Boral, H. "Parallelism in Bubba," Proc. Int. Symposium on Databases in Parallel and Distributed Systems, Austin, Tx, Dec. 1988.

[BNST] Beeri C., S. Naqvi, O. Shmueli, and S. Tsur. "Set Constructors in a Logic Database Language", to appear in the Journal of Logic Programming.

[BMSU] Bancilhon, F., D. Maier, Y. Sagiv, J. Ullman, "Magic sets and other strange ways to implement logic programs", Proc. 5th ACM SIGMOD–SIGACT Symp. on Principles of Database Systems, 1986.

[Boc] Bocca, J., "On the Evaluation Strategy of Educe," Proc. 1986 ACM–SIGMOD Conference on Management of Data, pp. 368–378, 1986.

[Ceta1] Chimenti, D. et al., "An Overview of the LDL System," Database Engineering Bulletin, Vol. 10, No. 4, pp. 52-62, 1987.

[Ceta2] Chimenti, D. et al., "The LDL System Prototype," IEEE Journal on Data and Knowledge Engineering, March 1990.

[CeGW] Ceri, S., G. Gottlob and G. Wiederhold, "Interfacing Relational Databases and Prolog Efficiently," Expert Database Systems, L. Kerschberg (ed.), Benjamin/Cummings, 1987.

[CG89] Chimenti, D. and R. Gamboa. "The *SALAD* Cookbook: A User's Guide," MCC Technical Report No. ACA-ST-064-89.

[CGK1] Chimenti, D., R. Gamboa and R. Krishnamurthy. "Towards an Open Architecture for LDL," Proc. 15th VLDB, pp. 195-203, 1989.

[CGK2] Chimenti, D., R. Gamboa and R. Krishnamurthy, "Abstract Machine for LDL," Proc. 2nd Int. Conf on Extending Database Technology, EDBT'90, Venice, Italy, 1990.

[CoMa] Copeland, G. and Maier D., "Making SMALLTALK a Database System," Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 316–325, 1985.

[CoSh] Connell, J.L. and Shafer, L.B., "Structured Rapid Prototyping", Prentice Hall, 1989.

[DM89] "The Rapid Prototyping Conundrum", DATAMATION, June 1989.

[Fost] Foster, R.K. "Feature Comparison of LDL and SQL", Control Data Corporation Interoffice Memorandum, March 23, 1987.

[Gane] Gane, C. "Rapid System Development," Prentice Hall, 1989.

[GMN] Gallaire, H.,J. Minker and J.M. Nicolas,"Logic and Databases: a Deductive Approach," Computer Surveys, Vol. 16, No. 2, 1984.

[GENE88] R. Herdman, et al. "MAPPING OUR GENES Genome Projects: How Big, How Fast?" Congress of the United States, Office of Technology Assessment. The John Hopkins University Press, 1988.

[JaCV] Jarke, M., J. Clifford and Y. Vassiliou, "An Optimizing Prolog Front End to a Relational Query System," Proc. 1984 ACM-SIGMOD Conference on Management of Data, pp. 296–306, 1986.

[Kell] Kellogg, C., "A Practical Amalgam of Knowledge and Data Base Technology" Proc. of AAAI Conference, Pittsburg, Pa., 1982.

[KNZ] Krishnamurthy, S. Naqvi and Zaniolo, "Database Transactions in $\mathcal{LDL}$", Proc. Logic Programming North American Conference 1989, pp. 795-830, MIT Press, 1989.

[KrN1] Krishnamurthy and S. Naqvi, "Non-Deterministic Choice in Datalog," Proc. 3rd Int. Conf. on Data and Knowledge Bases, June 27–30, Jerusalem, Israel.

[KrN2] Krishnamurthy and S. Naqvi, "Towards a Real Horn Clause Language," Proc. 1988 VLDB Conference, Los Angeles, California, August 1988.

[KrZa] Krishnamurthy, R. and C. Zaniolo, "Optimization in a Logic Based language for Knowledge and Data Intensive Applications," in Advances in Database Technology, EDBT'88, (Schmidt, Ceri and Missikoff, Eds), pp. 16–33, Springer-Verlag 1988.

[KuYo] Kunifji S., H. Yokota, "Prolog and Relational Databases for 5th Generation Computer Systems," in Advances in Logic and Databases, Vol. 2 (Gallaire, Minker and Nicolas eds.), Plenum, New York, 1984.

[Li] Li, D. "A Prolog Database System," Research Institute Press, Letchworth, Hertfordshire, U.K., 1984

[LeVi] Lefebvre, A. and Vieille, L. "On Deductive Query Evaluation in the DedGin System," Proc. 1st Int. Conf. on Deductive and O-O Databases, Dec. 4-6, 1989, Kyoto, Japan.

[Llo] Lloyd, J. W., Foundations of Logic Programming, Springer Verlag, (2nd Edition), 1987.

[MaSu] Marek, V. and V.S. Subramanian, "The Relationship between Logic Program Semantics and Non-Monotonic Reasoning," Proc. 6th Int. Conference on Logic Programming, pp. 598-616, MIT Press, 1989.

[Meta] Morris, K. et al. "YAWN! (Yet Another Window on NAIL!), Data Engineering, Vol.10, No. 4, pp. 28–44, Dec. 1987.

[Mi68] Michie, D. "'Memo' Functions and Machine Learning" in Nature, April 1968.

[NaKr] Naqvi, S. and R. Krishnamurthy, "Semantics of Updates in logic Programming", Proc. 7th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems, pp. 251–261, 1988.

[Naq] Naqvi, S. "A Logic for Negation in Database Systems," in Foundations of Deductive Databases and Logic Programming, (Minker, J. ed.), Morgan Kaufman, Los Altos, 1987.

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

[Prz1] Przymusinski, T., "On the Semantics of Stratified Deductive Databases and Logic Programs", in Foundations of Deductive Databases and Logic Programming, (Minker, J. ed.), Morgan Kaufman, Los Altos, 1987.

[Prz2] Przymusinski, T., "Non-Monotonic Formalism and Logic Programming," Proc. 6th Int. Conference on Logic Programming, pp. 656-674, MIT Press, 1989.

[RaBK] Ramakrishnan, R., C. Beeri and Krishnamurthy, "Optimizing Existential Datalog Queries," Proc. 7th ACM SIGMOD–SIGACT Symp. on Principles of Database Systems, pp. 89–102, 1988.

[RaSh] Ramamohanarao, K. and J. Sheperd, "Answering Queries in Deductive Databases", Proc. 4th Int. Conference on Logic Programming, pp. 1014-1033, MIT Press, 1987.

[RoSh] Rowe, L. and K.A. Shones, "Data Abstraction, Views and Updates in RIGEL", Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 71-81, 1979.

[SaZ1] Saccá D., Zaniolo, C., "Implementation of Recursive Queries for a Data Language based on Pure Horn Logic," Proc. Fourth Int. Conference on Logic Programming, Melbourne, Australia, 1987.

[SaZ2] Saccá D., Zaniolo, C., "The Generalized Counting Method for Recursive Logic Queries," Journal of Theoretical Computer Science, 61, 1988.

[SaZ3] Saccá D., Zaniolo, C., "Stable Models and Non-Determinism in Logic Programs with Negation," MCC Tech. Rep., ACT-ST-202, 1989.

[Sch77] Schmidt, J., "Some High Level Language Constructs for Data of Type Relations", ACM Transactions on Database Systems, 2(3), pp. 140-173, 1977.

[Seta] Selinger, P.G. et al. "Access Path Selection in a Relational Database Management System," Proc. ACM SIGMOD Int. Conf. on Management of Data, 1979.

[ShNa] Shmueli, O. and S. Naqvi, "Set Grouping and Layering in Horn Clause Programs," Proc. of 4th Int. Conf. on Logic Programming, pp. 152–177, 1987.

[ShTZ] Shmueli, O., S. Tsur and C. Zaniolo, "Rewriting of Rules Containing Set Terms in a Logic Data Language (LDL)," Proc. 7th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems, pp. 15–28, 1988.

[Tsur] Tsur S., "Applications of Deductive Database Systems," Proc. IEEE COMCON Spring '90 Conf., San Francisco, Feb 26-March 2.

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

[vEKo] van Emden, M.H., Kowalski, R., "The semantics of Predicate Logic as a Programming Language", JACM 23, 4, 1976, pp. 733–742.

[War] Warren, D.H.D., "An Abstract Prolog Instruction Set," Tech. Note 309, AI Center, Computer Science and Technology Div., SRI, 1983.

[Zan1] Zaniolo, C. "Prolog: a database query language for all seasons," in Expert Database Systems, Proc. of the First Int. Workshop L. Kerschberg (ed.), Benjamin/Cummings, 1986.

[Zan2] Zaniolo, C. "Design and implementation of a logic based language for data intensive applications. *Proceedings of the International Conference on Logic Programming*, Seattle, 1988.

[Zan3] Zaniolo, C. "Object Identity and Inheritance in Deductive Databases: an Evolutionary Approach," Proc. 1st Int. Conf. on Deductive and O-O Databases, Dec. 4-6, 1989, Kyoto, Japan.