The $\mathcal{LDL}$ System Prototype

D. Chimenti, R. Gamboa, R. Krishnamurthy,
S. Naqvi*, S. Tsur and C. Zaniolo

Microelectronics and Computer Technology Corporation
Austin, Texas, USA

November 9, 1999

Abstract

The $\mathcal{LDL}$ system provides a declarative logic-based language and integrates relational database and logic programming technologies so as to support advanced data and knowledge-based applications. This paper contains a comprehensive overview of the system and contains a description of the $\mathcal{LDL}$ language and the compilation techniques employed to translate $\mathcal{LDL}$ queries into target queries on the stored data. The paper further contains a description of the architecture and runtime environment of the system and the optimization techniques employed in order to improve the performance and assure the safety of the compiled queries. The paper concludes with an account of the experience gained so far with the system, and discusses application areas where the $\mathcal{LDL}$ approach appears to be particularly effective.

1 Introduction

The objective of the Logic Data Language ($\mathcal{LDL}$) System is to develop the technology for a new generation of database systems that support the rapid development of sophisticated applications—such as expert systems and advanced scientific and engineering applications. This objective is not new, since there has been considerable interest in database languages [BaBu], which have been proposed as the vehicle for facilitating the development of complex data intensive applications, and bridging the gap between the database and the programming language—this gap is often described as

*Now at Bell Communication Research, Morristown, N.J, 07960

an 'impedance mismatch' [CoMa]. Yet, the approach favored by previous researchers has been that of interfacing relational DBMSs to traditional languages [RIGEL, Sch77] More recently, major efforts have been made to integrate databases and programming languages under the Object-Oriented paradigm [KiLo]. These approaches tend to abandon relational databases in favor of an object-oriented one—often supporting a limited query capability and the navigational query style of pre-relational systems. In contradistinction with these approaches, the $\mathcal{LDL}$ research has taken the viewpoint that full programming capabilities can and should be achieved through extensions of relational query languages, and through technology advances that provide efficient support for this as an integral part of the database management system. It is also believed that such a system represents an important way-station toward future Knowledge Management Systems, which will have to combine efficient inference mechanisms from Logic with efficient and secure management of large information banks from Database Systems. Toward this goal, the $\mathcal{LDL}$ project, which began in 1984, has produced a new language, new techniques for compilation and query optimization and an efficient and portable prototype. This paper recounts this experience and various lessons learned in this effort.

1.1 Overview

From the beginning, $\mathcal{LDL}$ was designed as a rule-based extension to relational domain calculus based languages. (In a domain calculus, variables stand for values, rather than tuples as in tuple-oriented calculus.) This was largely due to the influence of Prolog, and also to QBE (in-line version). It was felt that the expressive power of the former and the ease of use of the latter provided more desirable beacons for our endeavor than a straight extension of SQL. Yet, domain calculus and tuple calculus are known to be equivalent [Ull], and the overall techniques used for implementing $\mathcal{LDL}$ can be easily applied to suitable SQL extensions.

The basic research challenge faced was to provide a system that combined the expressive power of Prolog with the functionality and facilities of Data Base Management Systems (DBMSs), such as, support for transactions, recovery, schema-based integrity, and efficient management of secondary storage. It soon became clear that an approach based on coupling Prolog with relational databases [Boc, CeGW, KuYo, Li, JaCV] would not support the level of functionality, performance and ease of use that we were seeking. We realized that a fully integrated system is required, where there is no distinction between query language and application language, and that

arduous research challenges stood in the way of realizing such a goal.

The first issue that came into focus was that of users' responsibility for execution control. In the '70s and early '80s, the database field had witnessed a dramatic evolution from navigational systems into relational ones. In navigational systems, such as Codasyl-compliant DBMSs, the programmer must explicitly navigate through the maze of database records, paying careful attention to the sequential order in which these records are visited—the key to efficiency. In relational DBMSs, instead, the user is only responsible for the formulation of a correct query (using logic-based languages of limited expressive power, such as SQL or QUEL [Ull]). A special system module, called the query optimizer, then compiles each query into an efficient execution plan. By contrast, in Prolog, the programmer must carefully order rules and goals to ensure efficient execution and termination. This basic mismatch, from which all systems coupling Prolog with relational DBMSs suffer, also challenged $\mathcal{LDL}$'s quest for a harmonious integration, leaving two alternative paths open [Zan1]. One consisted of adding navigational database facilities to a Prolog-like language; the other of rejecting the navigational (procedural) semantics of Prolog, in favor of a purely declarative one, whereby the order of goals and rules in a program becomes immaterial.

In the fall of 1984, the critical decision was taken to pursue the second solution, with the expectation that it would provide better usability and suitability for massive parallelism, and it will lead to more exciting research problems and technology break-throughs. As described in the following paragraphs, this early decision had profound repercussions on both the design of the language and its implementation.

A Prolog programmer must be keenly aware of its sequential execution model (SLD-resolution where the leftmost goal and the first rule is selected [vEKo, Llo]), not only because the termination and performance of the program will depend on it, but also because the very semantics of the many non-Horn constructs —primarily cuts, and updates, but also negation and 'set-of' predicates— are based on such an execution model. These non-Horn constructs were introduced in Prolog to obtain the expressive power needed for application development. Having decided to divorce execution from the order of rules and goals in the program, the first technical challenge facing $\mathcal{LDL}$ research was to provide a clean design and a formal declarative semantics for the non-Horn constructs that were needed in the language for reasons of expressive power. The result is a language that is very different from Prolog in terms of the constructs and programming style it entails.

Most design choices regarding the $\mathcal{LDL}$ implementation approach were dictated by the need for supporting database applications efficiently. Thus,

in $\mathcal{LDL}$ only rules are compiled. The fact base is described at compile time by a schema, and can then be updated freely at run time with no need for program interpretation or recompilation. This is a first difference from Prolog systems where facts and rules are treated in the same way (thus requiring interpretation when facts are changed). Furthermore, we concluded that the implementation technology of Prolog and systems based on backward-chaining, which is based on efficient implementations of SLD-resolution and unification [Llo, War], was too dependent on main memory, and a different approach was needed to obtain maximum performance on secondary-storage resident data. Thus, a simpler execution model was selected that is based upon the operations of matching and the computation of least fixpoints through iterations. A benefit of this approach is that matching operators on sets of facts can be implemented using simple extensions to the Relational Algebra [Zan2, Zan3] used by many relational databases. A second advantage is that since recursion has been replaced by iteration, we can now use a simpler and more static environment for execution.

Having chosen a simpler target language, the $\mathcal{LDL}$ designers were faced with the challenge of designing a more sophisticated compiler to support the full functionality of the source language. The approach chosen is built on two pillars:

- the use of global analysis to infer the bindings induced by a specific query in rules and goals, and
- the compilation methods which rewrite recursive programs that, as such, are not efficient or safe to implement by fixpoint computations into equivalent programs that are.

The first $\mathcal{LDL}$ implementation, completed in 1987, was based on a compiler using an early version of a language called FAD as the target language, and on an interpreter for this language [DaKV]. FAD is a language based on relational algebra that is supported by a massively parallel database machine designed at MCC. While this experiment produced a fully functional system, FAD was then dropped as the target language for the following reasons. The FAD interpreter that was available was not robust and fast enough to support serious experimentation. Furthermore, the only FAD implementation which was to be made available was for a large and expensive parallel system—hardly an affordable and portable vehicle for the release of $\mathcal{LDL}$. This led to the decision of designing and developing *SALAD*—an efficient and portable $\mathcal{LDL}$ system for UNIX. This implementation assumed a single-tuple get-next interface between the compiled $\mathcal{LDL}$ program and

the underlying fact manager. The single-tuple framework created an opportunity for refinements and optimization that was not available in the framework of relational algebra. The implementation included a fact manager for a database residing in virtual memory that supported efficient access to the complex and variable record structures available in $\mathcal{LDL}$.

The completion of the *SALAD* prototype in 1988 made it possible to start developing interesting applications in $\mathcal{LDL}$. Various extensions and improvements were added to the system as a result of this experience. As the system improved, we have expanded the domain of its applications beyond traditional database applications. Owing to its open architecture and its compiling into C, *SALAD* finds applications as a rule-based system for rapid prototyping of applications in the C environment. An incipient understanding of a paradigm for programming in $\mathcal{LDL}$ has also emerged from this experience, along with various ideas for desirable improvements.

1.2 Structure of the Paper

Section 2 summarizes key techniques and concepts implemented in the system—most of them novel and untried techniques developed by the $\mathcal{LDL}$ researchers or by parallel efforts, such as [Meta]. Thus, Section 2.1 gives a brief survey of the novel features of the language, while 2.2 summarizes the rule compilation techniques for constant pushing and efficient implementation of recursion. Section 2.3 describes the various execution schemes supported by the system, while 2.4 describes the optimizer that, at compile time, selects a safe and efficient execution for the given query.

Section 3 describes the architecture and implementation of *SALAD*, including a discussion of the main modules (Section 3.1), various techniques for peephole optimization (Section 3.2) and the fact manager (Section 3.3).

Section 4 recounts our experience with $\mathcal{LDL}$ and *SALAD* and with using them in novel application areas.

2 Enabling Technology

2.1 Language Design

The language was designed to combine the declarative style of relational languages with the expressive power of Prolog. Concretely, that meant using Horn Clauses as Prolog did, and rejecting all the remaining Prolog constructs, such as negation, set_of, updates, cuts, etc. These constructs were added to Prolog to obtain the expressive power necessary for writing

general applications. While Horn Clauses have a well-defined declarative semantics, these additional constructs only had an operational semantics which is based on Prolog's execution model. Thus, a first challenge in our work was to design into the language proper constructs for negation, sets, updates and non-determinism and give them a formal semantics that extends that of Horn Clauses. This semantics can be formally defined using the notion of minimal model; an alternative but equivalent definition based on the notion of least fixpoint is also possible [vEKo, Llo]. A detailed discussion of the $\mathcal{LDL}$ design is outside the scope of this paper which focuses on implementation issues. Thus, we will only provide a brief discussion of the main constructs to illustrate the richness of the language and the complexity of compilation and optimization issues posed by its implementation. The reader interested in a detailed discussion of $\mathcal{LDL}$ and its formal semantics is referred to [NaTs].

Languages such as DATALOG support rules and recursion. A full Horn Clause language also supports complex terms through the use of function symbols. Thus, for instance, the record of an employee could have the following form:

```
employee(name(joe, doe), admin,
         education(high_school, 1967))
```

Along with the employee name we find the department where he works (`admin`) and his education. While `admin` is a simple term, the other two are complex terms, entailing an internal structure of unrestricted complexity. For instance, in the education field, one may want to keep more detailed information (such as school name, level and major) for people with college degrees, and, for instance, have a record of the following format:

```
employee(name(joe, cool), sales,
         education(college(harvard, bs, math), 1971))
```

Each sub-argument can be further refined into a more detailed description, thus enabling the modeling of objects of arbitrarily complex structure—including recursive structures such as lists and trees. $\mathcal{LDL}$ has enhanced this complex term capability by providing for set terms and nested relations. Thus, we can now have a complete education record for a person as follows:

```
employee(name(joe, smart), mts,
         education({(high_school, 1967),
                    (college(harvard, bs, math), 1971)
                    (college(harvard, ms, engr), 1973) })).
```

Set terms in $\mathcal{LDL}$ are first class citizens, having the well-known properties of sets, such as commutativity and idempotence— but not associativity [BNST, ShTZ]. In addition to nested relations, $\mathcal{LDL}$ provides simple constructs for nesting and unnesting these relations.

The problem of negated goals in recursive rules represents one of the main research challenges in defining a declarative semantics for $\mathcal{LDL}$. This problem has been resolved with the introduction of the rather natural concept of stratification [ApBW, Naq, Prz]. Informally speaking, this result disallows the circular definition of a predicate using the negation of the same. Similar constraints must also be observed when defining the nesting of sets [BNST, ShNa].

Updates were defined so as to allow the full use of these constructs in rules and to support the notion of database transactions [NaKr]. The difficult problem of formalizing their semantics was solved through the use of dynamic logic [Har]. The semantics so defined reduces to first order logic in the absence of updates.

Finally, the notion of functional dependencies was used to support nondeterminism through a construct called choice [KrN1].

2.2 The Compilation Problem

The $\mathcal{LDL}$ compiler performs several functions, beginning with the parsing of the rules into a Predicate Connection Graph (PCG) [KeOT] and ending with the code generation phase. Some details of this complex process are discussed in Section 3, others are beyond the scope of this paper. In this section, we describe the rule rewriting phase which is the conceptual kernel of the compiler. The objective of this is to specialize and refine the original program into one that is specialized for the particular constraints resulting from the query and rules at hand. To a large extent, this process can be viewed as a generalization of the well-known principle of pushing selection and projection operations into relational expressions. This compilation phase begins when a query form is given, i.e., a query with mode declarations specifying the arguments that will be given (ground) at actual query time. Then, the constant migration step for non-recursive predicates is performed. For instance, consider the query form

```
?grandma($X,Y).
```

(where `$X` denotes that a value is to be supplied at actual query time) and the following set of rules:

```
grandma(X,Z) ← parent(X,Y), mother (Y,Z).
parent(X,Y) ← father(X,Y).
parent(X,Y) ← mother(X,Y).
```

The constant migration step will actually insert `$X` (since this value is known at run time, it is treated as a constant by the compiler) into the corresponding arguments and variables in the rules, yielding

```
?grandma($X,Y).
grandma($X,Z) ← parent($X,Y), mother (Y,Z).
parent($X,Y) ← father($X,Y).
parent($X,Y) ← mother($X,Y).
```

This set of rules can be further simplified by dropping the first argument in `grandma` and `parent`:

```
?grandma'(Y).
grandma'(Z) ← parent'(Y), mother (Y,Z).
parent'(Y) ← father($X,Y).
parent'(Y) ← mother($X,Y).
```

Thus, the original program has been specialized for the given query form. Furthermore, since `$X` has been migrated from the query form into the database predicates (`father` and `mother`), the corresponding selection operation has been pushed from the root of the relational algebra tree representing the query to the leaf nodes, where the selection is applied against the database tuples [Ull]. This 'selection pushing' operation, which is the linchpin of the query processing strategy of relational systems [KrZa, Ull], is implemented here by simple rule transformation techniques.

The treatment of recursive predicates is, in general, more complex. The program specialization approach described above works for some simple cases of recursive predicates. For instance, the following query

```
?anc(marc, Z).
anc(X, Z) ← anc(X, Y), parent(Y, Z).
anc(X, X) ← person(X).
```

can be supported by specializing the `anc` rules into

```
anc(marc, Z) ← anc(marc, Y), parent(Y, Z).
anc(marc, marc) ← person(marc).
```

and then dropping the constant argument from `anc` to yield:

```
anc'(Z) ← anc'(Y), parent(Y, Z).
anc'(marc) ← person(marc).
```

A single fixpoint iteration computes this transitive closure efficiently. The original query condition is now applied directly to the datum `parent` relation and not the derived `anc` relation, i.e., selection has been pushed inside recursion. Furthermore, a refinement of fixpoint known as *semi-naive fixpoint* is used to solve this problem [Ban, BaR, Ull, SaZ4]. The seminaive fixpoint iteration basically will begin by computing the parents of `marc` and then the parents of the parents, and so on until no new ancestor is found.

More complex rewriting is required, however, before the following query can be mapped into a single fixpoint:

```
?anc(X, brian).
```

Here, the recursive rule must be first rewritten in its right-linear form, as follows:

```
anc"(X, Z) ← parent(X,Y), anc"(X, Z).
```

Then, the specialization approach can be applied, resulting in linear transitive closure kind of rules that are easily mapped into a single seminaive fixpoint.

Because of the frequency with which simple transitive-closure type of rules are encountered, the $\mathcal{LDL}$ compiler performs some sophisticated analysis to recognize cases where the recursion can be supported efficiently through a single fixpoint computation.

However, there are many situations were constants cannot be pushed into recursion [AhUl], and, therefore, a recursive goal with bound arguments cannot be computed efficiently or safely by a single fixpoint computation. (The problem of detecting when constants can be pushed into recursion is in general undecidable [Bet2]).

Thus, more complex rewriting techniques are used to handle the general case. Take, for instance, the well-known same generation example (two individuals are of the same generation if their parents are, and everyone is of the same generation as him/herself).

```
sg(X,X).
sg(X,Y) ← parent(X, XP), sg(XP, YP), parent(Y, YP).
```

A query such as,

```
?sg(marc, X).
```

cannot be supported by the rules obtained by replacing `X` by `marc`. Moreover, a bottom-up computation is impossible since the exit rule, `sg(X,X)`, could qualify an infinite number of tuples. Similar problems occur in computational procedures, such as list-append, where taking advantage of bound arguments is essential for a safe and efficient implementation.

A considerable amount of research has been devoted to this key problem and the reader is referred to [BaRa] for an overview of these techniques. The $\mathcal{LDL}$ compiler uses the *magic set method* [BMSU, SaZ2] and the *generalized counting method* [SaZ3], which are expressible by rule rewriting scripts and lead to efficient implementations using fixpoint computations. In a nutshell, these methods take a recursive clique that, for the given query, cannot be supported well by means of a fixpoint computation and recast it into a pair of connected recursive cliques, each amenable to efficient fixpoint implementation.

This transformation can be illustrated by the example where people of the same generation as `marc` are sought. One alternative way to find these people consists of

- deriving the ancestors of `marc` and counting the levels as we go up (`marc` being a zero level ancestor of himself).
- once an ancestor of `marc`, say `X`, is found, then the descendants of `X` are computed, while levels are counted down. Descendants for which the level counter is zero are of the same generation as `marc`.

We can express the previous computations as follows (`J+1` and `J-1` denote the respective successor and predecessor of the integer `J`):

```
sg.up(0, marc).
sg.up(J+1, XP) ← parent(X, XP), sg.up(J, X).
sg.down(J, X) ← sg.up(J, X).
sg.down(J-1,X) ← sg.down(J, YP), parent(Y, YP).
?sg.down(0,X).
```

Thus, the initial recursive clique has been reformulated into a pair of recursive cliques connected via the index J. Each recursive clique can now be implemented efficiently and safely using a fixpoint computation (indeed each is basically a transitive closure operation).

The equivalence preserving transformation that we have just introduced using the intuitive semantics of ancestry, can be performed with full generality on a purely syntactic basis. Indeed, observe that in the succession

of recursive calls generated by the goal `sg(marc, X)`, `X` and `XP` are bound whereas `Y` and `YP` are not. Thus, the recursive `sg.down` rule is basically constructed by dropping the bound arguments and retaining the others, while a new argument is added to perform the count-down. The recursive rule for `sg.up` is instead built by retaining the bound arguments and then exchanging the recursive predicate in the head with that in the tail of the rule (indeed, we want to simulate a top-down computation by a bottom-up one), and then adding the count-up indexes. Also observe that the original exit rule is used to glue together the up and down computations. Finally, the bound part of the query goal becomes the new exit rule for `sg.up`, while the unbound part becomes the new query goal. The generalized and formal expression of these rule rewriting techniques, known as the generalized counting method are given in [SaZ3].

The counting method is very efficient for acyclic databases, but will loop forever, as Prolog does, for cyclic databases, e.g., for the same-generation example above, if the `parent` relation has cycles. The magic set method can be used to solve the cycle problem and also for complex recursive situations [BMSU, SaZ2].

While no function symbols were present in the previous examples, all the compilation techniques just described apply when these are present. This entails the manipulation of trees, lists and complex structures.

Another area of considerable innovation in the $\mathcal{LDL}$ compiler is the support for set terms. Set terms are treated as complex terms having the commutativity and idempotence properties. These properties are supported via compile time rule transformation techniques, that use sorting and various optimization techniques to eliminate blind run-time searches for commutative and idempotent matches [ShTZ].

2.3 Modes of Execution

Even though $\mathcal{LDL}$'s semantics is defined in a bottom-up fashion fashion (e.g., via stratification), the implementor can use any execution that is faithful to this declarative semantics. In particular, the execution can be bottom-up and top-down as well as hybrid executions that incorporate memoing [Mi68]. These choices enable the optimizer/compiler to be selective in customizing the most appropriate mode for the given program.

As a first approximation, it is easy to view the $\mathcal{LDL}$ execution as a bottom up computation using relational algebra. For instance, let `p(...)` be the query with the following rule, where `p1` and `p2` are either database or derived predicates:

```
p(X,Y) ← p1(X,Z), p2(Z,Y).
```

Then, this query can be answered by first computing the relations representing `p1` and `p2` and then computing their join followed by a projection. In actuality, the $\mathcal{LDL}$ optimizer and compiler can select and implement the rule above using four different execution modes, as follows:

- **Pipelined Execution** computes *only* those tuples in `p2` that join with tuples of `p1` in a pipelined fashion. This avoids the computation of any tuple of `p2` that does not join with `p1` (i.e., *no superfluous work*), whereas, if a tuple in `p2` joins with many tuples in `p1` then it is computed many times.
- **Lazy Pipelined Execution** is a pipelined execution in which, as the tuples are generated for `p2`, they are stored in a temporary relation, say `rp2`, for subsequent use. Therefore, any tuple in `p2` is computed exactly once even if it is used many times (i.e., *amortized work* as well as no superfluous work of pipelined execution). Further, as both these pipelined executions compute `p2`-tuples one at a time, it is possible to avoid residual computation in the case of intelligent backtracking—this will be called *backtrackable advantage*.
- **Lazy Materialized Execution** proceeds as in the lazy pipelined case except that, for a given `Z`-value, all tuples in `p2` that join with the tuple in `p1` are computed and stored in a relation before proceeding. The main advantage of this execution is that the execution is *reentrant* (a property that is important in the context of recursion), whereas the above two pipelined execution are not as they compute tuples of `p2` one at a time. On the other hand, this execution does not have the backtrackable advantage.
- **Materialized Execution** computes *all* tuples in `p2` and stores them in the relation, say `rp2`. Then, the computation proceeds using the tuples from `rp2`. Note this has the amortized work and reentrant advantages but lacks the backtrackable and superfluous work advantage.

Note that the above discussion can be generalized to any OR-node with a (possibly empty) set of bound arguments.

In conclusion, the pipelined execution is useful if the joining column is a key for `p1`, whereas the materialized execution is the best if all the `Z`-values of `p2` are joined with some `p1` tuple. Note that in both of these cases, the respective lazy evaluation incurs more overhead due to the checking that is needed for each `p1` tuple. The reentrant property is especially useful if the predicate is in the scope of a recursive query that is being computed top-down. Therefore, in such cases, lazy materialized execution is preferred over

lazy pipelined execution. Otherwise, lazy pipelined execution is preferred to exploit the backtrackable property.

Even though we have limited our discussion here to a single non-recursive rule, this can be generalized to include arbitrary rules with recursion. This is presented in detail in [CGK89b].

2.4 The Optimization Problem

The query optimizer is delegated the responsibility of choosing an optimal execution —a function similar to that of an optimizer in a relational database system. The optimizer uses the knowledge of storage structures, information about database statistics, estimation of cost, etc. to predict the cost of various execution schemes chosen from a pre-defined search space and select a minimum cost execution.

As compared to relational queries, $\mathcal{LDL}$ queries pose a new set of problems which stem from the following observations. First, the model of data is enhanced to include complex objects (e.g., hierarchies, heterogeneous data allowed for an attribute). Secondly, new operators are needed not only to operate on complex data, but also to handle new operations such as recursion, negation, etc. Thus, the complexity of data as well as the set of operations emphasize the need for new database statistics and new estimations of cost. Finally, the use of evaluable functions (i.e., external procedures), and function symbols in conjunction with recursion, provide the ability to state queries that are *unsafe* (i.e., do not terminate). As unsafe executions are a limiting case of poor executions, the optimizer guarantees the choice of a safe execution.

We formally define the optimization problem as follows: "Given a query Q, an execution space E and a cost model defined over E, find an execution in E that is of minimum cost." We discuss the advances in the context of this formulation of the problem. Any solution to this problem can be described along three main coordinates: (1) execution space, (2) search strategy, and (3) cost model.

2.4.1 Search Space and Strategies

The search space for optimal executions is defined by set of all allowable executions. This in turn is is defined, by a set of (i) *execution graphs* and (ii) for each graph, a set of allowable *annotations* associated with its nodes.

An execution graph is basically a structure of nested AND/OR graphs. This representation is similar to the predicate connection graph [KeOT], or

rule graph [Ull], except that we give specific semantics to the internal nodes as described below. The AND/OR graph corresponding to a nonrecursive program is the obvious graph with AND/OR nodes having one-to-one correspondence to the head of a rule and predicate occurrence. A recursive predicate occurrence, `p`, has subtrees whose roots correspond not only to the rules for this predicate but also to the rules in the recursive clique containing `p`. Intuitively, the fixpoint of all the rules below this OR node (i.e., predicate occurrence for `p`) need to be computed, to compute `p`.

The annotation provides all other information that is needed to model the execution. Intuitively, a parameter or property is modeled as an annotation if, for a given structure of the execution graph, the optimal choice of that information can be greedily chosen. For example, given the ordering (i.e., the structure) of the joins for a conjunctive query, the choice of access methods, creation of indices, and pushing of selection are examples of choices that can be greedily decided. On the other hand, the pushing of selection into a recursive clique is not a property that can be greedily chosen.

For instance, annotations define which of the four execution methods previously described are to be used. Each predicate occurrence (i.e., OR node) is annotated with an execution method. In addition, annotations describe which indexes should be used and whether duplicate elimination should be performed at the particular node.

Much effort has been devoted in devising efficient search strategies and enabling the optimizer to use alternative strategies, including exhaustive search, stochastic search and polynomial algorithms.

The traditional DBMS approach to using exhaustive search is to use the dynamic programming algorithm proposed in [Seta]. It is well known that even this is rendered useless if there is a join of 15 relations. In [KrZa] we propose an exhaustive search for optimizing $\mathcal{LDL}$ programs over the execution space. This approach is feasible as long as the number of arguments and the number of predicate occurrences in the body are reasonably small (i.e., 10).

Stochastic approaches provide effective means to find a near-optimal solution. Intuitively, near-optimal executions can be found by picking, randomly, a "large" sub-set of executions from the execution space and choosing the minimum cost execution. Simulated Annealing [IoWo], and variation thereof [SG88], are very effective in limiting the subset which must be searched before a reasonable approximation is found.

Polynomial search algorithms can be obtained by making some simplifying assumptions on the nature of cost functions. In [KrBZ], we presented a quadratic time algorithm that computes the optimal ordering of conjunctive

queries when the query is acyclic and the cost function satisfies a linearity property called the Adjacent Sequence Interchange (ASI) property. Further, this algorithm was extended to include cyclic queries and other cost models.

2.4.2 Cost Estimates and Safety

The cost model assigns a cost to each execution, thereby ordering them. Intuitively, the cost of an execution is the sum of the cost of its individual operations. Therefore, the cost function must be capable of computing the cost of each operation based on the descriptors of the operands. Three major problems are faced in devising such cost functions: 1) computation of the descriptors, 2) estimating the cost of external predicates, 3) safety of recursive queries.

In the presence of nested views, especially with recursion and complex objects, estimating the descriptor for a relation corresponding to a predicate is a very difficult problem. This is further complicated by the fact that logic based languages allow the union of non-homogeneous sets of objects. The net effect is that the estimation of the descriptor for any predicate is, in effect, computing the query in an algebraic fashion. That is, the program is executed in the abstract domain instead of the concrete domain. For instance, the *age* attribute may take on values such as 16 in the concrete domain whereas, in the abstract domain, it takes on values such as *integer between 16 to 65*. Obviously, computation in this domain is very difficult and approximations to such computation had to be devised that are not only efficient but are also effective.

In $\mathcal{LDL}$, external procedures (e.g., 'C' programs) are treated in an interchangeable manner with any predicate. Intuitively, the external procedure is viewed as an infinite relation satisfying some constraints. Therefore, a concise descriptor of such an infinite relation must be declared in the schema, and the cost functions for the operations on these infinite relations must be devised. The abstraction of the approach taken in $\mathcal{LDL}$ has been presented in [CGK89c]. This approach integrates it with the traditional optimization framework in a seamless fashion.

The cost model must associate an infinite cost for an execution that computes an infinite answer or that never completes. Such *unsafe* queries are to be detected so that the Optimizer can avoid choosing them. For example consider the following definition of all integers from zero to a given integer `K`.

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

As intended, the above program is unsafe when all arguments are free. So let us discuss the safety of this predicate when the first argument is bound and the second is free. Note that for each iteration of the recursive rule, the value of `J` is increasing and there is an upper bound on the value, which is the *given* value of `K`. Thus it can be concluded that the number of iterations is finite and each iteration produces only finite tuples. Consequently, the rule is safe.

In general, the problem of checking for safety is undecidable. The safety checking algorithm proposed in [KrRS] is to find a well-founded formula that can be used as a sufficient condition to guarantee safety. This algorithm is an enumerative algorithm that exhausts an exponential number of cases, to ensure the existence of a well-founded formula for each recursive cycle. The enumerative algorithm guesses well-founded formulae and checks each one of them until one is found to be satisfied.

3 System Architecture

Figure 1 shows the conceptual architecture for the current $\mathcal{LDL}$ prototype[1]. There are six basic components or modules in the current prototype: the User Interface, the Fact Manager, the Schema Manager, the Query Manager, the Rule Manager and the Query Form Manager. Section 3.1 provides a brief overview of the functionality of the different modules and section 3.2 discusses a few details pertaining to the system architecture and relevant to the compilation process

3.1 Main Modules

The User Interface receives and processes user commands, i.e., it invokes various procedures in the appropriate manager modules. The commands available from the User Interface are described in [CG89]. The Fact Manager is responsible for maintaining the various data structures associated with the extensional database as well as for providing run-time support for $\mathcal{LDL}$ queries. The Fact Manager data structures are collectively referred to as the Internal Fact Base. The Schema Manager receives the schema definition file from the User Interface and records the information in an internal form. Type, index and key constraints are subsequently used by the Fact Manager to verify the database. Base relation specifications are used by the Rule

[1]The current implementation contains approximately 70,000 lines of code, of which half is in Prolog and half is in C.

Figure 1: Conceptual architecture

Manager to verify consistency. The Query Manager receives queries from the User Interface, determines which compiled query form is appropriate for the query, and invokes the corresponding C program, passing any constants from the query as arguments.

The Rule Manager is responsible for processing the intentional database, i.e., the rule base. During the initial processing, the rules are parsed and various syntactic consistency checks are performed. Each parsed rule is stored in the Internal Rule Base and then sent to the Global PCG Generator, which is responsible for transforming the rule set into a Predicate Connection Graph (PCG). The *Global PCG* is a tabular data structure with entries specifying the rule/goal index for all predicates occurring in the rule base. It provides an efficient means of accessing the rules during subsequent query form processing. After all rules have been processed, the Recursive Clique Analyzer is invoked to identify maximal recursive cliques, detect cliques with no exit rules, and create the necessary internal structures to represent the cliques (RC-Boxes). The strongly connected components (predicates) of the PCG define its recursive cliques. Additional data structures for representing $\mathcal{LDL}$ modules and externals [CGK89a] are also produced by the Rule Manager.

The Query Form Manager embodies the bulk of the $\mathcal{LDL}$ compilation technology. It receives a query form from the User Interface and is responsible for producing the compiled version of the query form. Figure‘2 shows the organization of the Query Form Manager.

The Relevant PCG Generator generates a *Relevant PCG* (RPCG) which is an AND/OR graph containing only those rules relevant to the query form. The data structure generated is actually a tree instead of a graph since common sub-expression elimination is not currently part of the compiler design. During the RPCG extraction process, constant migration, i.e., the process of substituting deferred constants from the query form or constants from the relevant rules for variables wherever possible is also performed. Note that constants are not migrated into recursive rules.

The Optimizer transforms the RPCG and its associated recursive cliques as necessary to choose an optimal execution. It performs safety analysis and reorders goals (OR nodes) in the PCG appropriately. The nodes of the PCG are annotated by the Optimizer to reflect, among other things, adornment, pre-selection, post-selection and execution strategies to be employed. The transformed RPCG is termed the *Controlled PCG* (CPCG).

The Pre-Enhancer is responsible for providing the program adornment when the Optimizer is not used (AS-IS compilation). Miscellaneous rewriting optimizations, e.g., for choice, are also handled by the Pre-Enhancer. The Enhancer is responsible for rewriting recursive rules such that the re-

Figure 2: Query form manager architecture

cursive cliques are recast into a form that guarantees efficient execution via fixpoint operators. Various recursive query processing strategies are supported including a stack based implementation of the generalized counting method, the magic set method, and the semi-naive fixpoint method. The output of the Enhancer is the *Enhanced PCG* (EPCG).

The Set Rewriter uses rule transformation techniques to produce a revised but equivalent PCG where set objects have been mapped into first order terms in order to avoid set unification at run-time. The set properties of commutativity and idempotence are supported via this rule rewriting. In the process, the context of the rule is used to constrain the set of alternatives that must be explored [ShTZ].

Finally, the Code Generator traverses the PCG, generating C code, ultimately resulting in a complete C Program which is then compiled and linked to form the final compiled query form. The Code Generator is actually quite sophisticated in that it performs various peephole optimizations, e.g., intelligent backtracking and existential query optimization, provides default annotations in the case of AS-IS compilation, and supports various execution strategies, all on the fly as code is generated.

3.2 Compilation Techniques

In addition to the rule transformations described in Section 2.2, the $\mathcal{LDL}$ compiler applies a number of techniques to improve the efficiency of the run-time code.

3.2.1 Pruning the Execution Graph

Much of the unification required to support complex terms is performed at compile-time. Consider, for instance, the following rules:

```
r(X,Y) ← b1(X), p(f(X,Y)), b2(X,Z).

p(V) ← ..., V = g(U), ...
p(V) ← ..., V = f(U,U), ...
```

Compile-time rewriting of these rules will result in the function `f(X,Y)` being *migrated* into the rules for `p` such as to replace all occurrences of `V`. Subsequently, the first rule for `p` will be deemed false and will be thrown out of the relevant rule set. Furthermore, the second rule for `p` will result in the unification of `X` with `Y` and the substitution throughout the rule of `X` for `U`. At compile-time it will be determined whether an assignment (value of `X`

assigned to `Y`) or a check (value of `X` is the same as value of `Y`) is required, based on whether the given variables are bound or not. Note that the code generator would choose the entry to the rule for `p` as the appropriate place for the check in order to detect early failure, whereas the assignment would be placed at the success of the rule in order to avoid an unnecessary assignment should the rule fail. Thus, the run-time effort is reduced by eliminating rules and performing compile-time unification such that only simple matching and assignments are necessary at run-time. This same philosophy is employed for set unification such that set objects are mapped into first order terms at compile-time so that only ordinary matching is required at run-time.

3.2.2 Static Variables

One of the goals of the rewriting performed by the system is to rename variables, so that the scope of each variable is global with respect to the program. The purpose of this rewriting is run-time efficiency. By making each variable global, space for the variables can be allocated statically, as opposed to dynamically as offsets from a frame pointer. Moreover, assigning a variable can be done more efficiently in a global framework, as parameter passing becomes unnecessary. On the other hand, non-recursive rules that are invoked from more than one predicate are duplicated, thus resulting in larger object code.

3.2.3 Adornment

For each query form, the compiler constructs an *adorned program* using the notion of sideways information passing (SIP) as defined in [Ull], marking each argument of each predicate as either *bound* (instantiated to a particular constant value at run-time), *free* (the current predicate occurrence will instantiate it at run-time) or *existential* (it does not appear elsewhere in the rule, except possibly in the head as an existential argument). Note that the rules, in some cases, are duplicated (and renamed) for different adornments of the predicate occurrence (referred to as *stability transformation* [SaZ2]). Thus, each predicate in the adorned program is associated with an unique binding pattern and every occurrence of that predicate conforms to that binding pattern. The program segment generated for a predicate can exploit the bound/existential arguments to generate efficient code. This approach of generating code for a particular predicate with respect to a given binding pattern is an important deviation from the approach taken in Prolog and it results in an improved performance.

3.2.4 Intelligent BacKtracking

The nested-loop join operation which is implied by pipelined execution presents significant opportunities for avoiding computation that cannot generate new results. In the literature this is know as the *intelligent backtracking* problem. Two types of intelligent backtracking been addressed in the compiler: *get-next* and *get-first*. Consider, again, the $\mathcal{LDL}$ rules given above. Let us assume that the rules are compiled for the query `?r(X,Y)`. After computing a tuple for `r`, backtracking to get the next tuple for `b2` is unnecessary since it will not yield any new tuples for `r`. The compiler will choose the predicate `p` as the get-next backtrack point for the rule since the variable `Y` is bound there[2]. To illustrate get-first intelligent backtracking, consider the predicate `b2`. If the attempt to get the first tuple in `b2` fails, it is unnecessary to backtrack to `p` since it does not change the bound argument for `b2`. Therefore, if no tuples are found for `b2`, the backtrack point will be `b1` since that is where the variable `X` is bound. Hence, by doing compile-time analysis, intelligent backtracking is implemented with little (if any) overhead incurred at run-time, and results in the elimination of unnecessary run-time processing.

The rule for `r` also serves to illustrate an additional optimization utilized by the compiler with respect to existential arguments. In the predicate `b2`, the variable `Z` is a don't care or existential variable. Therefore, the assignment of a value to `Z` is unnecessary. While this might seem an inconsequential optimization, experience has shown that the avoidance of a single assignment in the innermost loop can have a great influence on execution time. Again, compile-time analysis has avoided unnecessary overhead at run-time.

3.2.5 Implementation of Recursion, Updates and Choice

Above, we have discussed backtracking assuming a pipelined execution ala Prolog. In order to efficiently compile some of the advanced constructs of $\mathcal{LDL}$, additional execution strategies, i.e. materialized, lazy materialized, lazy pipelined, snapshot and stack-based executions, must be used. These different execution methods along with their respective advantages and disadvantages are described in detail in [CGK89b]. The $\mathcal{LDL}$ code generator is capable of selectively applying any execution strategy (chosen by the op-

[2]It is interesting to note that if the query were `?r(X,_)` such that the variable `Y` was existential with respect to the rule for `r`, then the get-next backtrack point for that rule would be the predicate `b1`.

timizer) to any predicate in the rule set. Moreover, some language features dictate the appropriate execution strategy that must be applied. Set grouping and recursion are examples where materialization is essential to a correct execution. Full materialization, however, does not allow for selection pushing and is, therefore, very inefficient in the presence of bound arguments. Therefore, a lazy materialized execution is applied such that the bindings can be utilized. Additionally, for recursion, rewriting strategies are employed at compile-time which recast the recursion into a form that guarantees efficient execution via fixpoint operators. The magic set rewriting method, which uses the lazy materialized execution strategy, is applied when there is a possibility of cyclic data. The user can compile with an option which states that there no need to detect cycles, in which case the compiler can choose a stack-based implementation of the counting method for better performance With this approach, a pipelined or lazy-pipelined execution strategy can be employed. Hence, an appropriate execution strategy can be chosen in context at compile-time to ensure an efficient run-time execution.

Because the semantics of $\mathcal{LDL}$ ascribe a dynamic logic interpretation to updates[NaKr], a snapshot may be required for every update operation. The compiler does, however, recognize instances where snapshots are not necessary and, thus, sequences of updates can be collapsed.

The implementation of $\mathcal{LDL}$'s nondeterministic choice construct requires materialization to store the functional dependencies. In the following rules, a table with `X` and `Y` values will be materialized due to the choice construct.

```
r(X,Y) ← ..., p(X,Y), X < Y, ...
p(X,Y) ← b(X), q(X,Y), choice((X),(Y)).
```

The chosen `Y` value for a particular `X` will only be committed, however, at the success of the query. In the rule for `r`, it is possible that the goal `X` $<$ `Y` will fail resulting in backtracking into the rule for `p` and obtaining a new choice, i.e., value for `Y`. This may be contrasted with the Prolog cut with which a bad choice will result in failure for the query[3]. After values have been committed, the materialized table can be used to avoid unnecessary recomputation. Thus, after the binding for `X` is obtained from the predicate `b`, a check is performed to determine if a value for `Y` has already been committed and, if so, the remainder of the rule need not be executed. Again, compile-time techniques have been used to reduce the computation effort at run-time.

[3]The Prolog cut also does not provide for functional dependencies to be expressed.

3.3 The Fact Manager

The fact manager provides the run-time environment for an $\mathcal{LDL}$ program. It supports $\mathcal{LDL}$ objects, such as atoms, sets and lists, as well as database objects, such as tuples and base and derived relations. In the current implementation, all objects are kept in virtual memory.

The $\mathcal{LDL}$ data types are directly supported by the fact manager, which implements them as C abstract data types. That is, the fact manager provides C type definitions as well as a set of routines that operate on objects of these types. This is the level of abstraction maintained by the translator. The fact manager itself, on the other hand, is free to take advantage of the data representation for the sake of efficiency. For example, complex objects are stored as one-dimensional arrays, where the first (zeroth in C) component is the functor name. The function `fm_get_functor_arg(object,i)` is used by the translator to select the i^{th} component of a complex object. The fact manager implements this in-line (i.e., in the preprocessor) as the array lookup `object[i]`. Similarly, the fact manager stores sets as sorted arrays, so that set operations such as union and intersection can be implemented efficiently.

Efficient support for base and derived relations is provided at the tuple level, with calls such as `fm_get_first` and `fm_get_next`. A key consideration in the design of the fact manager was the number of operations performed at the inner-most loop of an execution (i.e., nested join); for example, getting the next tuple from a base relation and post-selecting for bound arguments. Thus, relations are stored so that the call to `fm_get_next` is reduced to following a linked list, and is hence suitable for in-line implementation. This is possible, because the database is kept in-memory, thus it is never necessary to access the next tuple from disk.

In order to speed up equality comparisons, used in post-selection, for example, each object in the database is assigned an unique representation that can be compared using the hardware (integer) compare instruction. In the case of numeric constants, the unique representation is quite natural. For strings and complex objects, the memory address of the actual object is used as the unique representation. Whenever a new complex object is created, the fact manager guarantees this address is unique by first checking whether the object already exists, an efficient operation, since all objects are kept in memory. This unique representation can also be used by the fact manager to perform other database operations more efficiently. For example, when an index is used, the hash function operates directly on the unique representation rather than the $\mathcal{LDL}$ object itself — this can

be a substantial savings, since $\mathcal{LDL}$ objects can be arbitrarily complex. Moreover, once a bucket is selected, searching to find the matching tuples involves an equality comparison, so the unique representation is exploited here as well. Intuitively, the unique representation allows the fact manager to reduce the cost of subsequent index lookups by partially hashing an object when it is created.

Derived relations are used by the translator to support some $\mathcal{LDL}$ language features, such as recursion and grouping. Recursion can be implemented using a semi-naive fixpoint operation, after rewriting for magic sets, etc, has taken place. Thus, the efficient execution of recursion depends on the efficient implementation of the semi-naive operation. Therefore, the fact manager supports this operation directly by partitioning recursive relations into delta and cumulative components, and returning tuples only from the delta component when a semi-naive scan is desired. Since tuples are inserted sequentially, the delta component is implemented easily by maintaining a "high-water" mark. Similarly, the fact manager provides efficient support of grouping by converting a relation into a set, given a pattern describing the specific "group-by" operation desired.

4 Experience

4.1 Experience in Using the Language

Since $\mathcal{LDL}$ was designed to be both a query language and a rule-based application language we need to evaluate its functionality and usability starting from these two domains.

An independent comparison of $\mathcal{LDL}$ as a database query language, suggested that all but the simplest queries are easier to express in $\mathcal{LDL}$ than SQL. This hardly represents an endorsement of $\mathcal{LDL}$, since the inordinate difficulty of expressing sophisticated queries in SQL is well-known. Yet, our experience suggests that even the most complex of queries can be readily expressed as short $\mathcal{LDL}$ programs. This is consistent with our experience that in $\mathcal{LDL}$, any distinction between complex queries and simple applications is arbitrary and blurred. We found it easy to develop rapidly complex database applications, including the "Computer Science Genealogy" [NaTs] and programs for parts explosion, inventory control, shop scheduling and

The other side of the coin involves comparing $\mathcal{LDL}$ with other rule-based systems. As our reader may have noticed, a coarse description of the $\mathcal{LDL}$ compiler is that it maps the functionality of a backward chaining system (top-down) into the mechanisms of forward chaining (bottom-up). Indeed,

we felt that the former is conducive to more expressive and powerful languages while the the second is conducive to more efficient implementations in the database context. Thus, programming in $\mathcal{LDL}$ is more similar to programming in Prolog than in OPS5. Yet, the differences between $\mathcal{LDL}$ and Prolog are significant, and often baffling to experienced Prolog programmers.

Prolog is more powerful than $\mathcal{LDL}$ in many respects, such as built-in predicates, including metapredicates. Moreover, Prolog variables can be instantiated in a dynamic fashion—e.g., different goals instantiate variables in a complex term. $\mathcal{LDL}$ is more restrictive since, although goals can be reordered at compile time, the execution of a goal is assumed to bind all its variables. Also, the fact that Prolog streams through one answer at the time provides the programmer with more opportunities for fine control than in $\mathcal{LDL}$.

On the other hand, $\mathcal{LDL}$ provides a more structured programming paradigm, a cleaner syntax and semantics, and an optimizer that excuses the user from thinking too hard about an execution sequence. The benefits become apparent in challenging areas such as non-determinism and recursion. For instance, a recursive procedure, to generate all integers between zero and a given integer `K`, can be expressed as follows in $\mathcal{LDL}$:

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

This represents a very natural rendering of Peano's inductive definition of integers, augmented with a condition on `K` in the second rule to ensure termination, and one in the first rule to ensure that no answer is returned for a negative `K`. A second formulation is also possible in $\mathcal{LDL}$, as follows:

```
int(K,J) ← K>0, K1 = K-1, int(K1, J).
int(K,K) ← K >= 0.
```

This is a less clear and intuitive definition, but it is the only one that can be handled by Prolog (the equal signs would also have to be replaced by '`is`'). Also, when writing a recursive predicate to traverse a graph with possible cycles, the Prolog programmer must make provisions for termination (e.g., carrying around in a bag all answers produced so far). Cycles can be easily handled by the $\mathcal{LDL}$ compiler through a specific option.

Finally, the ability of storing efficiently partial results that can be retrieved by later computations is a major plus for $\mathcal{LDL}$, and so is the ease of dealing with externals and modules.

Therefore, a plausible argument can be made for the ease-of-use of $\mathcal{LDL}$ over Prolog; but this is hardly a reason for jubilation. Much more work is needed to bring the system to a level of usability and ease-of-use that will entice non-professional programmers to develop complex applications, in analogy to what many 4GL users now do with simple applications. We are currently working on two major extensions directed toward enhancing the ease of use. One is a debugger that, given the nature of the system, is tantamount to an answer justification capability. A traditional debugger that retraces the execution of the program would be of little help to the unsophisticated user, since the compiler and optimizer completely transform the original program. What is instead planned, is an answer justification capability capable of carrying out a dialogue with a user asking questions such as, *"Why did you (or did you not) return this answer?"* and through this dialogue directing the user to the incorrect rule or missing fact that was the source of the problem. We also plan to add visual interfaces both for data entry and display and for program visualization. While in procedural languages, the focus of visualization is on the changes to the program state, for a declarative language such as $\mathcal{LDL}$ the focus is on displaying the static relationships defined by the rules.

We now briefly describe some aspects that affect the performance of the current implementation of $\mathcal{LDL}$. One important feature of $\mathcal{LDL}$ is the elimination of duplicate results in the processing of recursion—that is, an "all answers" as opposed to "all proofs" approach. The duplicates need to be eliminated in certain cases to guarantee termination, such as when traversing a cyclic graph. Moreover, the elimination of duplicates can speed up the execution in many cases. For example, in the computation of the same generation query, it was discovered that removing duplicates resulted in a major performance improvement, since for most siblings in the database, there are two ways to prove their relation (through the father or mother), and this becomes even more significant as more distant relations (e.g., through great-grandparents) are explored. A timing comparison using a database of 500 tuples showed that the system computed same generation in roughly 4 seconds, whereas Quintus Prolog needed over 2 minutes, resulting in a ratio of over 1:30.

On the other hand, there are also recursive queries where no duplicates are ever generated, for example, when appending two lists. In these queries, the overhead of duplicate elimination is wasted, hence the $\mathcal{LDL}$ implementation does not compare favorably with, say, Prolog. In particular, for list append, we found a ratio of between 6:1 and 10:1 in favor of Prolog. Another factor contributing to this result is the uniqueness check performed at

the creation of each new object, i.e., temporary list. When this check was removed, the ratio was reduced to 2:1.

4.2 $\mathcal{LDL}$ Applications

In this section we will report on the experience that we have gained with the $\mathcal{LDL}$ system so far. We recognized that the only way in which the utility of this new technology can be assessed is by application development. In this process it is useful to distinguish between two classes of applications:

- "Old" applications such as database ones that have traditionally been implemented by a procedural application program with embedded query calls to the underlying database.
- "New" applications. Applications, that thus far were never implemented at all or, if they were implemented, then this was accomplished without any use of database technology.

As described in the previous section, the experience with traditional database applications has been positive. Here we will concentrate on two new promising application area; these are *data dredging* and *harnessing software*.

4.2.1 Data Dredging

This is a class of applications in which the source of data is typically (but not exclusively) a very large set of empirical observations or measurements, organized into one or more base relations. Additional data may be added over time but existing data are seldom updated. In fact, they are only updated when found to be erroneous. Typical sources are measurement data of empirical processes or data, recorded during simulation experiments. The problem is to *interpret* this data, i.e., to use it for the verification of certain hypotheses or, to use it for the formulation of new concepts. In both cases the hypotheses or concepts may be conceptually far removed from the level of the recorded data and their crystalization or *definition* entails an interactive human/system process 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. While this process could be carried out using any programming language, the use of $\mathcal{LDL}$ has the advantage that the formulation can be done at an abstract level and hence, the "iteration time" through this process is significantly shortened as compared to the traditional way, in which each iteration involves the usual programming/compile/debug cycle.

We experimented with data dredging in two different application domains: computer system performance evaluation and scientific data analysis in the area of Microbiology. 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 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 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. 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 "smarts" to the system as the experience with its use increases.

4.2.2 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: export/import relationships between modules, constraints on their combinations etc.

The advantage of this organization is that the knowledge base can be used in decisions that pertain to the reuse of software. 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 this application type in the domain of Banking software.

5 Conclusion

Perhaps, the most significant result of the $\mathcal{LDL}$ experience is proving the technical feasibility of building a logic-based application language as an extension of relational databases technology. The realization of this objective has required solving technical challenges on many fronts—language design and formal definition, compilation, optimization and system implementation. In the five years since the beginning of the project, problems have been solved trough the combined efforts of a group of six to eight people. Perhaps the most encouraging aspect of the whole experience is that while a wide spectrum of interests and backgrounds—from a very theoretical one to a very applied one—was represented in the group the effort remained foucused and generated a remarkable degree of synergism. The result is a system that supports the theoretical declarative semantics of the language completely and efficiently.

Our experience suggests that it is reasonably easy to develop applications using the $\mathcal{LDL}$ programming paradigm. But this conclusion is based on a small sample of forward-looking programmers who are leaning toward to declarative languages and logic programming. Whether the language incorporating concepts such as recursion can attract large throngs of mainstream practitioners is still to be seen. But it also clear that $\mathcal{LDL}$ has much

more to offer than current SQL-based 4GLs that are widely used for rapid prototyping [DM89]. Thus, $\mathcal{LDL}$ shows some real potential as a powerful rule-based language for the rapid development of data intensive applications and applications in the C environment. A main thrust of our current efforts is to improve the usability of the system by supporting interfaces for visual programming and answer justification.

Acknowledgments

The authors would like to recognize the contribution the following persons: Brijesh Agarwal, François Bancilhon, Catriel Beeri, Charles Kellogg, Paris Kanellakis, Tony O'Hare, Kayliang Ong, Arshad Matin, Raghu Ramakrishnan, Domenico Saccá, Oded Shmueli, Leona Slepetis, Peter Song, Emilia Villarreal, Carolyn West.

References

[AhUl] Aho A. V. and J. Ullman, " Universality of Data Retrieval Languages," Proc. POPL Conference, San Antonio Tx, 1979.

[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.

[Ban] Bancilhon, F., "Naive Evaluation of Recursively defined Relations", On Knowledge Base Management Systems, (M. Brodie and J. Mylopoulos, eds.), Springer–Verlag, 1985.

[BaR] Balbin, I., K. Ramamohanarao, "A Differential Approach to Query Optimization in Recursive Deductive Databases", Journal of Logic Programming, Vol. 4, No. 2, pp. 259–262, Sept 1987.

[BaRa] Bancilhon, F., and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," Proc. ACM SIGMOD Int. Conference on Management of Data, Washington, D.C., May 1986.

[Bet1] Beeri, et al., "Sets and Negation in a Logic Data Language (LDL1)", Proc. 6th ACM SIGMOD–SIGACT Symp. on Principles of Database Systems, pp. 269–283, 1987.

[Bet2] Beeri, et al., "Bound on the Propagation of Selection in Logic Programs", Proc. 6th ACM SIGMOD– SIGACT Symp. on Principles of Database Systems, 1987.

[BKBR] Beeri, C., P. Kanellakis, F. Bancilhon, R. Ramakrishnan, "Bound on the Propagation of Selection into Logic Programs", Proc. 6th ACM SIGMOD–SIGACT Symp. on Principles of Database Systems, 1987.

[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.

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

[Ceta] Chimenti D. et al., "An Overview of the $\mathcal{LDL}$ System," Database Engineering Bulletin, Vol. 10, No. 4, pp. 52–62, 1987.

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

[CGK89a] Chimenti, D., R. Gamboa and R. Krishnamurthy. "Using Modules and Externals in LDL," MCC Technical Report No. ACA-ST-036-89.

[CGK89b] Chimenti, D., R. Gamboa and R. Krishnamurthy. "Abstract Machine for LDL," MCC Technical Report No. ACA-ST-268-89.

[CGK89c] Chimenti, D., R. Gamboa and R. Krishnamurthy. "Towards an Open Architecture for LDL," Proc. 15th VLDB, pp. 195-203, 1989.

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

[DaKV] Danforth, S.,S. Khoshafian and P. Valduriez, "FAD—A Database Programming Language. Rev 2", Submitted for publication.

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

[deSi] deMandreville C. and E. Simon, "Modelling Queries and Updates in Deductive Databases" Proc. 1988 VLDB Conference, Los Angeles, California, August 1988.

[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.

[Har] Harel, D., "First-Order Dynamic Logic," Lecture Notes in Computer Science, (G. Goos and J. Hartmanis, eds.), Springer Verlag, 1979.

[IoWo] Ioannidis, Y. E. and E. Wong, "Query Optimization by Simulated Annealing", Proc. ACM SIGMOD Int. Conf. on Management of Data, 1987.

[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.

[KeOT] Kellogg, C., A. O'Hare and L. Travis, "Optimizing the Rule Data Interface in a KMS," Proc. 12th VLDB Conference, Tokyo, Japan, 1986.

[KiLo] Kim, W. and F.H. Lochosky (eds.), Object-Oriented Concepts, Databases, and Applications, ACM Press, New York, NY, Addison-Wesley Publishing Co., 1989.

[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.

[KrBZ] Krishnamurthy, R., H. Boral and C. Zaniolo, "Optimization of Non-Recursive Queries," Proc. 12th VLDB, Kyoto, Japan, 1986.

[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.

[KrRS] Krishnamurthy, R. R. Ramakrishnan and O. Shmueli, "A Framework for Testing Safety and Effective Computability," Proc. ACM SIGMOD Int. Conf. on Management of Data, pp. 154–163, 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.

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

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

[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.

[Prz] 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.

[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.

[RIGEL] 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., "On the implementation of a simple class of logic queries for databases", Proc. 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems, 1986.

[SaZ2] 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.

[SaZ3] Saccá D., Zaniolo, C., "The Generalized Counting Method for Recursive Logic Queries," Journal of Theoretical Computer Science, 61, 1988.

[SaZ4] Saccá D., Zaniolo, C., "Differential Fixpoint Methods and Stratification of Logic Programs," Proc. 3rd Int. Conf. on Data and Knowledge Bases, June 27-30, 1988, Jerusalem, Israel.

[Sch77] Schmidt, J., "Some High Level Language Constructs for Data of Type Relations", ACM Transactions on Database Systems, 2(3), pp. 140173, 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.

[SG88] Swami, A., and G. Gupta. "Optimization of Large Join Queries" in Proceedings ACM-SIGMOD International Conf. on Management of Data, 1988, pp. 8-17.

[TsZa] Tsur, S. and C. Zaniolo, "LDL: A Logic-Based Data Language," Proc. of 12th VLDB, Tokyo, Japan, 1986.

[Ull] Ullman, J.D., Database and Knowledge-Based Systems, Computer Science Press, Rockville, Md., 1988.

[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. "The Representation and Deductive Retrieval of Complex Objects," Proc. 11-th VLDB, pp. 459–469, 1985.

[Zan3] Zaniolo, C. "Safety and Compilation of Non-Recursive Horn Clauses," Proc. First Int. Conference on Expert Database Systems, L. Kerschberg (ed.), Benjamin/Cummings, 1986.