The $\mathcal{L} \mathcal{D} \mathcal{L}$ System Prototype

Danette Chimenti

Ruben Gamboa

Ravi Krishnamurthy

Shamim Naqvi

Shalom Tsur

Carlo Zaniolo

The $\mathcal{LDL}$ System Prototype

DANETTE CHIMENTI, RUBEN GAMBOA, RAVI KRISHNAMURTHY, SHAMIM NAQVI,
SHALOM TSUR, AND CARLO ZANIOLO

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 query execution plans on the stored data. The paper further contains a description of the architecture and run-time 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.

Index Terms—Data dredging, declarative programming, deductive databases, logical data language, optimization, rapid prototyping, recursion, rule-based systems, transitive closure.

I. 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 [3], 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 an "impedance mismatch" [18]. Yet, the approach favored by previous researchers has been that of interfacing relational DBMS's to traditional languages [45], [50]. More recently, major efforts have been made to integrate databases and programming languages under the object-oriented paradigm [28]. These approaches tend to abandon relational databases in favor of object-oriented ones—often supporting a limited query capability and the navigational query style of prerelational 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 these extensions as an integral part of the database

management system. It is also believed that the resulting system represents the first efficient realization of the concept of deductive databases [22] and 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 the experience and various lessons learned in this effort.

A. 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 [56], and the overall techniques used for implementing $\mathcal{LDL}$ can be easily applied to suitable SQL extensions.

The basic research challenge we faced was to provide a system that combined the expressive power of Prolog with the functionality and facilities of database management systems (DBMS's), 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 [11], [13], [35], [36], and [26] would not support the level of functionality, performance, and ease of use that we were seeking. We realized that a fully integrated system was required, where there was 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 1970's and early 1980's, the database field had witnessed a dramatic evolution from navigational systems into relational ones. In navigational systems, such as Codasyl-compliant DBMS's, the programmer must explicitly navigate through the maze of database records, paying careful at-

tention to the sequential order in which these records are visited—the key to efficiency. In relational DBMS's, 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 [56]). 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 DBMS's suffer, also challenged $\mathcal{LDL}$'s quest for a harmonious integration, leaving two alternative paths open [59]. 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 become 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 (one of our goals at that time), and it will lead to more exciting research problems and technology breakthroughs. 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 [57], [37]), 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 [35]. 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 $\mathcal{LDL}$, the 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 expert system shells based on backward-chaining, which, in turn, is based on efficient implementations of SLD resolution and unification [37], [58], was too dependent on the assumption that data reside in main memory, and a

different approach was needed to obtain maximum performance on secondary-storage resident data. Thus, an execution model with a simpler target language was selected that was 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 [60], [61] 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 are not efficient or safe to implement by fixpoint computations into equivalent programs that are.

Current implementations of relational languages use either a relational algebra interface or a get-next-tuple interface to a data manager. The $\mathcal{LDL}$ project experimented and proved feasibility on both these frameworks. Thus, the first $\mathcal{LDL}$ implementation, completed in 1987, used FAD [19]; this is a relational algebra based language supported by a massively parallel database machine deveoped at MCC [12], [7]. After a fully functional compiler for $\mathcal{LDL}$ into FAD was implemented and demonstrated in 1987, our focus shifted onto the problem of providing a portable prototype for $\mathcal{LDL}$. Thus, FAD was abandoned, since it is only available on a large and expensive parallel machine. Furthermore, we recognized that, for sequential architectures, the get-next-tuple interface offered opportunities for refinements and optimization that were not at hand in the framework of relational algebra. This led to the decision of designing and developing SALAD[1]—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 (record manager). The implementation included a fact manager for a database residing in virtual memory that supported efficient access to the complex and variable record structures provided 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 has 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 $\mathcal{LDL}$

---

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

has also emerged from this experience, along with various ideas for desirable improvements.

B. Structure of the Paper

Section II 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 [38]. Thus, Section II-A gives a brief survey of the novel features of the language, while Section II-B summarizes the rule compilation techniques for constant pushing and efficient implementation of recursion. Section II-C describes the various execution schemes supported by the system, while Section II-D describes the optimizer that, at compile time, selects a safe and efficient execution for the given query.   •

Section III describes the architecture and implementation of SALAD, including a discussion of the main modules (Section III-A), various techniques for peephole optimization (Section III-B), and the fact manager (Section III-C).

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

II. Enabling Technology

A. Language Design

The $\mathcal{LDL}$ 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 have 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 nondeterminism 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 [57], [37]. 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 [42].

In deductive database theory, much attention has been paid to languages such as Datalog [56] that support rules and recursion on flat relations. However, Horn clauses also support complex terms through the use of function symbols. $\mathcal{LDL}$ fully supports this capability whereby,

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, yielding a record of the following format:

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

Each subargument 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 [9], [53]. 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 [4], [41], [43]. Informally speaking, this result disallows the definition of a predicate in terms of the negation of the same. Similar constraints must also be observed when defining the nesting of sets [9], [54].

Updates were defined so as to allow the full use of these constructs in rules and to support the notion of database transactions [40], [49]. The difficult problem of formalizing their semantics was solved through the use of dynamic logic [44]. The semantics so defined reduces to first-order logic in the absence of updates.

Finally, $\mathcal{LDL}$ provides a declarative construct called choice to support DON'T CARE nondeterminism (a very useful notion that is supported in Prolog by an expedient operational construct called the cut). The declarative semantics of choice is based on the database concept of functional dependencies [56]. Thus, the rule

$$p(X, Y) \leftarrow q(X, Y), \text{choice}((X), (Y))$$

specifies that the relation $p(X, Y)$ is a maximal subset of $q(X, Y)$, such that the functional dependency $X \rightarrow Y$ holds. Thus, if there are several Y-values for a given X-value, then one is selected nondeterministically [31].

B. The Compilation Problem

The £𝔇£ compiler performs several functions, beginning with the parsing of the rules into a predicate connection graph (PCG) [47] and ending with the code generation phase. Some details of this complex process are discussed in Section III, 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 phase is to 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; a *query form* is a query template with mode declarations specifying the arguments that will be given (ground) at actual query time (thus, it defines a compiled query with parameters). Then, the constant migration step for nonrecursive predicates is performed. For instance, consider the query form

?grandma(X, Y).

(where $X denotes that a parameter 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:

?grandma1(Y).
grandma1(Z) ← parent1(Y), mother (Y, Z).
parent1(Y) ← father(X, Y).
parent1(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 [56]. This "selection pushing" operation, which is the linchpin

of the query processing strategy of relational systems [34], [56], 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

anc1(Z) ← anc1(Y), parent(Y, Z).
anc1(marc) ← person(marc).

A single fixpoint iteration computes this transitive closure in such a way that the original query condition is now applied directly to the datum parent relation and not the derived anc relation—selection has been pushed inside recursion. The particular algorithm used to compute the fixpoint is known as *semi-naive fixpoint* [4], [5], [56], [49]. This removes the redundancy connected with the naive fixpoint computation which, while computing the jth level ancestors, also recomputes all ancestors at level $i \leq j$.

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:

anc4(X, Z) ← parent(X, Y), anc4(Y, Z).

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

Because of the frequency with which simple transitive-closure type of rules are encountered, the £𝔇£ compiler performs some sophisticated analysis to recognize the cases where the recursion can be supported efficiently by a single fixpoint computation (these are special cases [1], and detecting them all is undecidable [8]). The £𝔇£ compiler handles the general case by rewriting techniques that use two fixpoint computations. Common examples of situations requiring such techniques include recursive rules for appending lists, traversing trees, and manipulating complex structures. As we plan to draw on the intuitive semantics of ancestry to make some general points, however, we will use in our discussion 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 [6] for an overview of these techniques. The $\mathcal{LDL}$ compiler uses the *magic set method* [10], [47] and the *generalized counting method* [48], which are expressible by rule rewriting scripts and lead to efficient implementations using fixpoint computations. These methods take a recursive clique (i.e., a set of mutually recursive predicates) 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($J-1$, Y) ← 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 by analyzing the arguments that are known (bound) and those that are to be computed when the recursive procedure is called. The first call, generated by the goal sg(marc, X), binds the first argument X. But, given X, we can easily compute XP, through parent(X, XP) and call again sg(XP, YP) with a bound first argument. This binding passing analysis allows us to classify the first argument of sg as bound and the other as unbound. Now the transformed rule for sg.down is basically

constructed by dropping the bound arguments and retaining the others—with a new argument 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 (in order 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 [48].

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 [10], [47].

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. Our reader should also observe that if the language restricted the user to a bottom-up operational semantics, as, e.g., production-rule based languages do, then the user would have to write a different program for each of the different queries previously discussed—with a considerable loss in convenience and reusability. .

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 [53].

C. Modes of Execution

Even though $\mathcal{LDL}$'s semantics is defined in a bottom-up fashion (e.g., via stratification), the implementation can use any execution that is faithful to this declarative semantics. In particular, the execution can be bottom-up or top-down or it may be a hybrid execution that incorporates memoing [39]. These choices enable the compiler/optimizer to be selective in customizing the most appropriate modes of execution 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 $p(\cdots)$ be the query with the following rule, where $p1$ and $p2$ are either database or derived predicates:

$$p(X, Y) \leftarrow p1(X, Z), p2(Z, Y).$$

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 $\mathcal{LDL}$ op-

timizer 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). Furthermore, 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 executions 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.

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.

Note that the above discussion can be generalized to any predicate occurrence with a (possible empty) set of bound arguments. Even though we have limited our discussion here to a single nonrecursive rule, this can be generalized to include arbitrary rules with recursion. This is presented in detail in [16].

D. The Optimization Problem

The query optimizer is delegated the responsibility of choosing an optimal execution plan—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 predefined 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). Second, 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 predicates), and function symbols in conjunction with recursion, results in 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.

1) Search Space and Strategies: The search space for optimal executions is defined by the set of all allowable executions. This in turn is defined by a set of 1) *execution graphs* and 2) 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 [27], or rule graph [56], 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 chosen using a simple greedy algorithm. For example, given the ordering (i.e., the structure) of the joins for a conjunctive query, the choice of access methods, creation of indexes, 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 described in Section II-C are to be used. Each predicate occurrence (i.e., OR node) is annotated with an execution method. In addition, annotations de-

scribe which indexes should be used and whether duplicate elimination should be performed at the particular node.

Much effort has been devoted to 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 [51]. It is well known that even this approach is rendered useless if there is a join of 15 or more relations. In [34], 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 (e.g., ten or less).

Stochastic approaches provide effective means to find a near-optimal solution. Intuitively, near-optimal executions can be found by picking, randomly, a "large" subset of executions from the execution space and choosing the minimum cost execution. Simulated annealing [25], and variations thereof [54], 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 [30], 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. Furthermore, this algorithm was extended to include cyclic queries and other cost models.

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 their 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, and 3) safety of recursive queries.

In the presence of several levels of derived predicates, 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 nonhomogeneous 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 a value 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 [17]. 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.

$$\text{int}(K, 0) \leftarrow K \geq 0.$$
$$\text{int}(K, J) \leftarrow \text{int}(K, I), I < K, J = I + 1.$$

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 [33] 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 formulas and checks each one of them until one is found to be satisfied.

III. System Architecture

Fig. 1 shows the conceptual architecture for the current $\mathcal{LDL}$ prototype.[2] 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 III-A provides a brief overview of the functionality of the different modules and Section III-B discusses a few details pertaining to the system architecture and relevant to the compilation process.

A. 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

---

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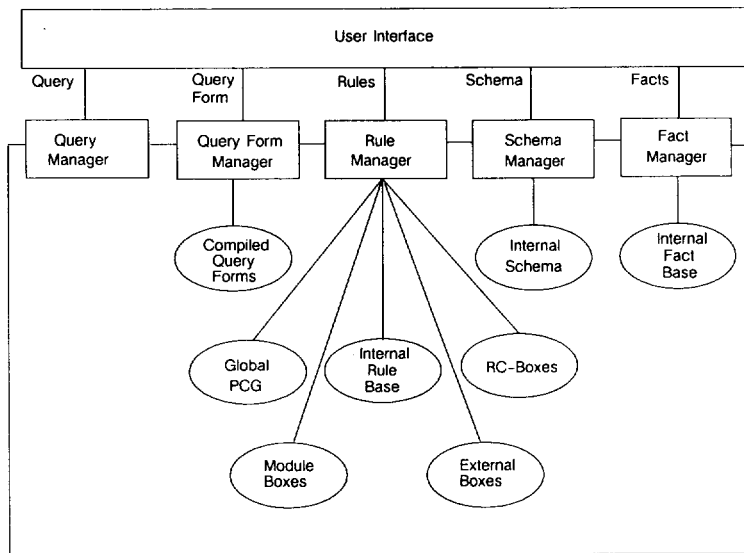
Fig. 1. Conceptual architecture.

the user interface are described in [14]. 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 £D£ 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 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 parsed, the recursive clique analyzer (RCA) is invoked to identify the strongly connected components (predicates) of the PCG, i.e., the recursive cliques, and to create the necessary internal structures to represent the cliques (RC boxes). In additon, the RCA detects those cliques for which no termination condition, i.e., exit rule, has been provided. Additional data structures for representing £D£ modules and externals [15] are also produced by the rule manager.

The query form manager embodies the bulk of the £D£ compilation technology. It receives a query form from the user interface and is responsible for producing
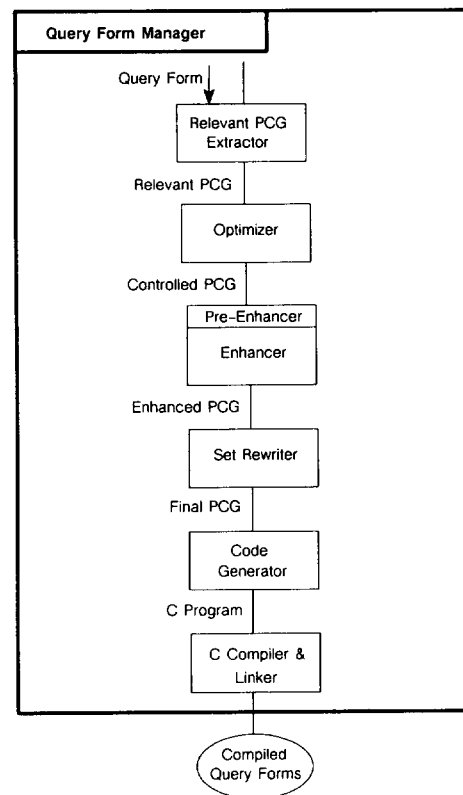

Fig. 2. Query form manager architecture.

the compiled version of the query form. Fig. 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 subexpression 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 is also performed wherever possible. 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, preselection, post-selection, and execution modes to be employed. The transformed RPCG is termed the *controlled PCG* (CPCG).

The enhancer is responsible for rewriting recursive rules such that the recursive 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 terms 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 [53].

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, and supports various execution modes, all on the fly as code is generated.

B. Compilation Techniques

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

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:

$$r(X, Y) \leftarrow b1(X), p(f(X, Y)), b2(X, Z).$$

$$p(V) \leftarrow \cdots, V = g(U), \cdots$$

$$p(V) \leftarrow \cdots, V = f(U, U), \cdots.$$

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. Since $f(X, Y)$ does not unify with $g(U)$, 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: set objects are mapped into first-order terms at compile-time so that only ordinary matching is required at run-time.

2) Static Variables: One of the goals of 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 a dynamical method, e.g., 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, nonrecursive rules that are invoked from more than one predicate are duplicated, thus resulting in larger object code.

3) Adornment: For each query form, the compiler constructs an *adorned program* using the notion of side-ways information passing (SIP) as defined in [56], 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 (this is referred to as *stability transformation* [47]). 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.

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 known as the *intelligent backtracking* problem. Two types of intelligent backtracking have been addressed in the compiler: *get-next* and *get-first*. Consider, again, the $\mathcal{LDL}$ rules given in Section III-B1. 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.[3] To illustrate get-first intelligent backtracking, consider the predicate $b2$. If the attempt to get the first tuple in $b2$ fails, it is not necessary 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 [44]. 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. The compile-time analysis has avoided unnecessary overhead at run-time.

5) Implementation of Recursion and Choice: Above, we have discussed backtracking assuming a pipelined execution ala Prolog. For efficiency and, in some cases, correctness, additional execution modes (discussed in Section II-C) must be employed for the compilation of some of $\mathcal{LDL}$ advanced constructs. For example, materialization is necessary when semi-naive fixpoint computations are used to process recursive goals. 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 mode is applied such that the bindings can be utilized. In addition, the counting set method of recursion rewriting has been implemented using a stack-based execution such that a pipelined or lazy pipelined execution mode is applicable. Hence, an appropriate execution mode for the recursive goal can be chosen in context at compile-time to ensure an efficient and correct run-time execution. Similar cases occur with the set grouping and update constructs; however, the relevant discussions have been omitted for brevity.

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.

$$r(X, Y) \leftarrow \cdots, p(X, Y), X < Y, \cdots$$

$$p(X, Y) \leftarrow b(X, Y), \text{choice}\left((X), (Y)\right).$$

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. 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.

C. 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, base relations, and derived relations (used to store temporary results). 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 ith 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 innermost 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 explicitly.

In order to speed up equality comparisons, notably in the presence of arbitrarily deep complex objects, each object in the database is assigned a unique representation that can be compared using the hardware (integer) compare instruction. In the case of numeric constants, the unique representation is the value of the object. 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 ob-

---

[3]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 $b1$.

jects are in memory. This unique representation is used to significantly speed up post-selection, which, as mentioned earlier, can be performed in the innermost loop of an execution. Moreover, it can also be used by the fact manager to perform internal database operations more efficiently. For example, when an index is used, the hash function operates directly on the unique representation rather than the LDL object itself—this can be a substantial savings, since 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 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 for grouping by converting a relation into a set, given a pattern describing the specific "group-by" operation desired.

IV. EXPERIENCE

A. Experience in Using the Language

Since 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 LDL as a database query language suggested that all but the simplest queries are easier to express in LDL than SQL [21]. This hardly represents an endorsement of 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 LDL programs. This is consistent with our experience that in 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" [42] and programs for parts explosion, inventory control, and job shop scheduling.

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

tom-up). Indeed, the latter is conducive to more efficient implementations in the database context, while the former is conducive to more expressive and powerful languages—a conclusion supported by our experience with several programs [42] and by the observation in the penultimate paragraphs of Section II-B.

Thus, programming in LDL is more similar to programming in Prolog than in OPS5. Yet, the differences between LDL and Prolog are significant, and often baffling to experienced Prolog programmers. Prolog has many powerful constructs (e.g., built-in and meta predicates) which are not in LDL (but, declarative relatives for some of these constructs are under consideration [32]). Moreover, in Prolog, variables can be instantiated in a dynamic fashion—e.g., different goals instantiate variables in a complex term. 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 produces one answer at the time provides the programmer with more opportunities for fine control than in LDL.

On the other hand, LDL leads to better structured programs and provides 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 nondeterminism and recursion [62]. (A procedure such that of Section II-D2 which provides a natural inductive definition of integers, cannot be used in Prolog.) When writing a recursive predicate to traverse a graph with possible cycles, the Prolog programmer must make provisions for termination (e.g., using an all-answer bag), while LDL handles cycles automatically. Finally, the ability of storing efficiently partial results that can be retrieved by later computations is a major plus for 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 LDL over Prolog. Furthermore, we are currently working toward levels of usability and ease-of-use that could entice nonprofessional programmers to develop complex applications, in analogy to what many 4GL users now do with simple applications. Toward this goal, we are currently working on a symbolic debugger that also supports answer justification. 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. We are currently completing an answer justification facility 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 are also working on an LDL interface that supports visual programming, along with form-oriented data entry and display.

We now briefly describe some aspects that affect the performance of the current implementation of LDL. Having the database memory resident, the performance of

any program is affected by the join in the innermost loop. Therefore, a simple join was executed, and compared to Prolog. The cardinality of the inner relation was varied from 20 to 100 and the cardinality of the outer relation was varied from 2000 to 10 000 tuples. As we were interested in measuring the work done in the innermost loop, we chose to vary the outer relation over a larger range. The other way would be favorable to £𝔇£ and quite unfavorable to Prolog. The timing ratio was in the range 1 : 1.7 to 1 : 1.95, in £𝔇£'s favor. This was further improved using existential query optimization to a ratio as high as 1 : 2.5. The conclusion was that the efficient comparison using uniqueness and in-line implementation helped £𝔇£ to outperform the Prolog system that has been fine-tuned using low-level Assembly language routines.

Another feature of £𝔇£ that affects the performance 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 the same generation query in roughly 4 s, whereas Quintus Prolog needed over 2 min, 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 £𝔇£ implementation does not compare favorably to, 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.

B. £𝔇£ Applications

In the end, the utility of the £𝔇£ 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, maintain, and extend using £𝔇£. As a result, we moved to more advanced applications, in areas beyond those of traditional DBMS's. Next, we discuss two new areas of particular interest, *data dredging* and *harnessing software*.

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* these 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 £𝔇£ 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 do not support this concept and that it should be abandoned or tried out with different data. The use of £𝔇£ over procedural languages offered the advantage of supporting the formulation at a more abstract level where the "iterative time" through reformulations is significantly shortened. With respect to existing database query languages, £𝔇£ 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 dual language 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 [42] 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 are low-level and consist of arrival/departure records of individual entities at certain nodes. The convoy concept was defined in £𝔇£ 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 [23]. 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 additional knowledge to the system as the experience with its use increases.

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; 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.

V. CONCLUSION

Perhaps the most significant result of the $\mathcal{LDL}$ experience was proving the technical feasibility of building a logic-based application language as an extension of relational database 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 through 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 very theoretical ones to very applied ones—was represented in the group, the effort remained focused and generated a remarkable degree of synergism. The result is a system that supports the theoretical declarative semantics of the language completely and efficiently.

The most obvious limitation of the current $\mathcal{LDL}$ implementation is the lack of a secondary store data manager

supporting the $\mathcal{LDL}$ data model and concurrent access to the database. While there is little question on the technical feasibility and desirability of such a system, building it is not our top priority, inasmuch as we are more concerned with providing an evolution path from current database systems than an alternative to them. Therefore, we are currently working on a tight coupling of $\mathcal{LDL}$ with an SQL system, where the two systems cooperate in the optimization of programs. This will provide an interesting testbed for evaluating possible SQL extensions.

A second area of current research aims at enhancing the functionality of current 4GL's by building upon $\mathcal{LDL}$ rather than SQL. The main research issue in this context is on how to enhance the ease-of-use of the system for less sophisticated programmers. The approach chosen consists in making powerful but demanding concepts such as rules and recursion more tractable through visual programming and answer explanation. This system will provide a powerful tool for the rapid development of database applications and the rapid prototyping of applications in the C environment [20].

REFERENCES

[1] A. V. Aho and J. Ullman, "Universality of data retrieval languages," in *Proc. POPL Conf.*, San Antonio TX, 1979.
[2] K. Apt, H. Blair, and A. Walker, "Towards a theory of declarative knowledge," in *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Los Altos, CA: Morgan-Kaufman, 1987.
[3] F. Bancilhon and P. Buneman, Eds., *Workshop on Database Programming Languages*, Roscoff, Finistere, France, Sept. 1987.
[4] F. Bancilhon, "Naive evaluation of recursively defined relations," in *On Knowledge Base Management Systems*, M. Brodie and J. Mylopoulos, Eds. New York: Springer-Verlag, 1985.
[5] I. Balbin and K. Ramamohanarao, "A differential approach to query optimization in recursive deductive databases," *J. Logic Programming*, vol. 4, no. 2, pp. 259–262, Sept. 1987.
[6] F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," in *Proc. ACM SIGMOD Int. Conf. Management Data*, Washington, DC, May 1986.
[7] H. Boral et al., "Phototyping Bubba," MCC Tech. Rep., Oct. 1989.
[8] C. Beeri et al., "Bound on the propagation of selection in logic programs," in *Proc. 6th ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, 1987.
[9] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur, "Set constructors in a logic database language," *Logic Programming*, to be published.
[10] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman, "Magic sets and other strange ways to implement logic programs," in *Proc. 5th ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, 1986.
[11] J. Bocca, "On the evaluation strategy of Educe," in *Proc. 1986 ACM-SIGMOD Conf. Management Data*, 1986, pp. 368–378.
[12] H. Boral, "Parallelism in Bubba," in *Proc. Int. Symp. Databases Parallel Distributed Syst.*, Austin, TX, Dec. 1988.
[13] S. Ceri, G. Gottlob, and G. Wiederhold, "Interfacing relational databases and Prolog efficiently," in *Expert Database Systems*, L. Kerschberg, Ed. Benjamin/Cummings, 1987.
[14] D. Chimenti and R. Gamboa, "The SALAD cookbook: A user's guide," MCC Tech. Rep. ACA-ST-064-89.

[15] D. Chimenti, R. Gamboa, and R. Krishnamurthy, "Using modules and externals in LDL," MCC Tech. Rep. ACA-ST-036-89.

[16] ——, "Abstract machine for LDL," MCC Tech. Rep. ACA-ST-268-89.

[17] ——, "Towards an open architecture for LDL," in *Proc. 15th VLDB*, 1989, pp. 195–203.

[18] G. Copeland and D. Maier, "Making SMALLTALK a database system," in *Proc. ACM SIGMOD Int. Conf. Management Data*, 1985, pp. 316–325.

[19] S. Danforth, S. Khoshafian, and P. Valduriez, "FAD—A database programming language, rev 2," MCC Tech. Rep. ACA-DB-151-85.

[20] J. M. Carey and J. D. Currey "The prototyping conundrum," *Datamation*, June 1, 1989.

[21] R. K. Foster, "Feature comparison of LDL and SQL," Control Data Corp. Interoffice Memo., Mar. 23, 1987.

[22] H. Gallaire, J. Minker, and J. M. Nicolas, "Logic and databases: A deductive approach," *Comput. Surveys*, vol. 16, no. 2, 1984.

[23] R. Herdman *et al.*, "MAPPING OUR GENES Genome Projects: How big, How fast?" Congress of the United States, Office of Technology Assessment. Baltimore, MD: John Hopkins University Press, 1988.

[24] D. Harel, "First-order dynamic logic," in *Lecture Notes in Computer Science*, G. Goos and J. Hartmanis, Eds. New York: Springer-Verlag, 1979.

[25] Y. E. Ioannidis and E. Wong, "Query optimization by simulated annealing," in *Proc. ACM SIGMOD Int. Conf. Management Data*, 1987.

[26] M. Jarke, J. Clifford, and Y. Vassiliou, "An optimizing prolog front/end to a relational query system," in *Proc. 1984 ACM–SIGMOD Conf. Management Data*, 1986, pp. 296–306.

[27] C. Kellogg, A. O'Hare, and L. Travis, "Optimizing the rule data interface in a KMS," in *Proc. 12th VLDB Conf.*, Tokyo, Japan, 1986.

[28] W. Kim and F. H. Lochosky, Eds., *Object-Oriented Concepts, Databases, and Applications*. New York: ACM Press and Addison-Wesley, 1989.

[29] R. Krishnamurthy, S. Naqvi, and C. Zaniolo, "Database transactions in £Đ£," in *Proc. Logic Programming North Amer. Conf.* Cambridge, MA: MIT Press, 1989, pp. 795–830.

[30] R. Krishnamurthy, H. Boral, and C. Zaniolo, "Optimization of non-recursive queries," in *Proc. 12th VLDB*, Kyoto, Japan, 1986.

[31] R. Krishnamurthy and S. Naqvi, "Non-deterministic choice in datalog," in *Proc. 3rd Int. Conf. Data Knowledge Bases*, June 27–30, Jerusalem, Israel.

[32] ——, "Towards a real Horn clause language," in *Proc. 1988 VLDB Conf.*, Los Angeles, CA, Aug. 1988.

[33] R. Krishnamurthy, R. R. Ramakrishnan, and O. Shmueli, "A framework for testing safety and effective computability," in *Proc. ACM SIGMOD Int. Conf. Management Data*, 1988, pp. 154–163.

[34] R. Krishnamurthy 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. New York: Springer-Verlag, 1988, pp. 16–33.

[35] S. Kunifji and H. Yokota, "Prolog and relational databases for 5th generation computer systems," in *Advances in Logic and Databases, Vol. 2*, Gallaire, Minker, and Nicolas, Eds. New York: Plenum, 1984.

[36] D. Li, "A Prolog database system," Lechtworth, Hertfordshire, U.K.: Research Institute Press, 1984.

[37] J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed. New York: Springer-Verlag, 1987.

[38] K. Morris *et al.*, "YAWN! (Yet another window on Nail!)," *Data Eng.*, vol. 10, no. 4, pp. 28–44, Dec. 1987.

[39] D. Michie, "'Memo' functions and machine learning," *Nature*, Apr. 1968.

[40] S. Naqvi and R. Krishnamurthy, "Semantics of updates in logic programming," in *Proc. 7th ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, 1988, pp. 251–261.

[41] S. Naqvi, "A logic for negation in database systems," in *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Los Altos, CA: Morgan-Kaufman, 1987.

[42] S. Naqvi and S. Tsur, *A Logical Language for Data and Knowledge Bases*. San Francisco, CA: Freeman, 1989.

[43] T. Przymusinski, "On the semantics of stratified deductive databases and logic programs," in *Foundations of Deductive Databases and Logic Programming*, J. Minker, Ed. Los Altos, CA: Morgan-Kaufman, 1987.

[44] R. Ramakrishnan, C. Beeri, and R. Krishnamurthy, "Optimizing existential datalog queries," in *Proc. 7th ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, 1988, pp. 89–102.

[45] L. Rowe and K. A. Shones, "Data abstraction, views and updates in RIGEL," in *Proc. ACM SIGMOD Int. Conf. Management Data*, 1979, pp. 71–81.

[46] D. Saccá and C. Zaniolo, "On the implementation of a simple class of logic queries for databases," in *Proc. 5th ACM SIGMOD-SIGACT Symp. Principles Database Syst.*, 1986.

[47] ——, "Implementation of recursive queries for a data language based on pure Horn logic," in *Proc. Fourth Int. Conf. Logic Programming*, Melbourne, Australia, 1987.

[48] ——, "The generalized counting method for recursive logic queries," *J. Theoretical Comput. Sci.*, vol. 61, 1988.

[49] ——, "Differential fixpoint methods and stratification of logic programs," in *Proc. 3rd Int. Conf. Data Knowledge Bases*, June 27–30, 1988, Jerusalem, Israel.

[50] J. Schmidt, "Some high level language constructs for data of type relations," *ACM Trans. Database Syst.*, vol. 2, no. 3, pp. 140–173, 1977.

[51] P. G. Selinger *et al.*, "Access path selection in a relational database management system," in *Proc. ACM SIGMOD Int. Conf. Management Data*, 1979.

[52] O. Shmueli and S. Naqvi, "Set grouping and layering in Horn clause programs," in *Proc. 4th Int. Conf. Logic Programming*, 1987, pp. 152–177.

[53] O. Shmueli, S. Tsur, and C. Zaniolo, "Rewriting of rules containing set terms in a logic data language (LDL)," in *Proc. 7th ACM SIG-MOD-SIGACT Symp. Principles Database Syst.*, 1988, pp. 15–28.

[54] A. Swami and G. Gupta, "Optimization of large join queries," in *Proc. ACM-SIGMOD Int. Conf. Management Data*, 1988, pp. 8–17.

[55] S. Tsur and C. Zaniolo, "LDL: A logic-based data language," in *Proc. 12th VLDB*, Tokyo, Japan, 1986.

[56] J. D. Ullman, *Database and Knowledge-Based Systems*. Rockville, MD: Computer Science Press, 1988.

[57] M. H. van Emden and R. Kowalski, "The semantics of predicate logic as a programming language," *J. ACM*, vol. 23, no. 4, pp. 733–742, 1976.

[58] D. H. D. Warren, "An abstract Prolog instruction set," Tech. Note 309, AI Center, Comput. Sci. Technol. Div., SRI, 1983.

[59] C. Zaniolo, "Prolog: A database query language for all seasons," in *Expert Database Systems, Proc. First Int. Workshop*, L. Kerschberg, Ed. Benjamin/Cummings, 1986.

[60] ——, "The representation and deductive retrieval of complex objects," in *Proc. 11th VLDB*, 1985, pp. 459–469.

[61] ——, "Safety and compilation of non-recursive Horn clauses," in *Proc. First Int. Conf. Expert Database Syst.*, L. Kerschberg, Ed. Benjamin/Cummings, 1986.

[62] ——, "Design and implementation of a logic based language for data intensive applications," in *Proc. Int. Conf. Logic Programming*, Seattle, WA, 1988.

Danette Chimenti received the B.S. degree in computer science from Texas A&M University, College Station, in 1980, and the M.S. degree in computer science from the University of Arizona in 1981.

Before joining MCC in 1986, she was a member of the Technical Staff at Sandia National Laboratories. Her interests include logic programming deductive databases, and fast cars.

Ruben Gamboa received the Bachelor's of Computer Science from Angelo State University in 1984 and the Master's of Computer Science from Texas A&M University, College Station, in 1986.

His current interests include deductive and graphical databases.
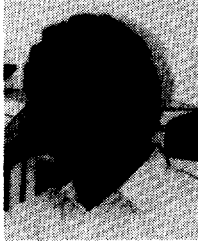
Ravi Krishnamurthy received the B.E. degree in electrical engineering from Birla Institute of Technology, Pilani, Rajasthan, India, the M.S. degree from the University of South Carolina in 1978, and the Ph.D. degree in computer science from the University of Texas at Austin in 1982.

Before returning to Austin to join MCC in 1985, he was with the IBM T. J. Watson Research Center, Yorktown, NY, for four years working on the Office-by-Example project.

Shalom Tsur received the Ph.D. degree in computer science from the Hebrew University of Jerusalem and the M.Sc. and B.Sc. degrees in electrical engineering from the Technion-Israel Institute of Technology, Haifa, Israel.

He is a senior member of the Technical Staff in MCC's Deductive Computing Laboratory. Prior to joining MCC, he was a member of the Research Staff at the IBM Israel Scientific Center, where he was Project Manager for the APE Expert System—a planning environment for advising students in their academic careers (1982-1984). In addition, he has been a member of the Technical Staff at AT&T Bell Laboratories (1980-1982), an Assistant Professor of Computer Science at the Pennsylvania State University (1977-1979), and a Senior Research Associate at the Computing Laboratory, the University of Newcastle, UK (1975-1977). His research interests include the areas of data and knowledge-base systems and theory, as well as logic programming. He is the author or coauthor of 25 technical papers and a book in these fields.
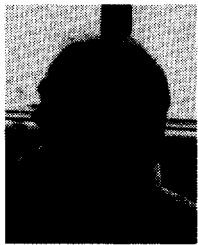
Shamim Naqvi received the Ph.D. degree in computer science from Northwestern University in 1980.

He is a member of the Technical Staff at Bell Communications Research. He was at MCC from 1986 to 1989, and at AT&T Bell Labs from 1980 to 1986. His areas of interest are database theory, programming languages, and he has published extensively in these areas.
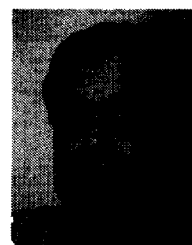
Carlo Zaniolo holds an Engineering Degree from Padua University, Italy, and the Ph.D. degree in computer science from the University of California, Los Angeles.

He has been with MCC since 1984, where he heads the Deductive Computing Laboratory in the Advanced Computer Architecture Program. MCC is a Research Consortium located in Austin, TX.