20.1 INTRODUCTION

The Logic Data Language, LDL, combines the expressive power of a high-level, logic-based language (such as Prolog) with the nonnavigational style of relational query languages, where the user need only supply a correct query, and the system (i.e., the compiler/optimizer) is expected to devise an efficient execution strategy for it. Consequently, the optimizer is given the responsibility of choosing an optimal execution—a function similar to that of the query optimizer in a relational database system. A relational system uses knowledge of storage structures, information about database statistics, and various estimates to predict the cost of execution schemes chosen from a predefined search space and to select a minimum cost execution in such a space.

An LDL system offers to a user all the benefits of a database language—including the elimination of the impedance mismatch between the language and the query language. In addition, its rule-based deductive capability and its unification-based pattern matching capability make it very suitable for knowledge-based and symbolic applications. The power of LDL is not without
CONTROL AND OPTIMIZATION STRATEGIES IN THE IMPLEMENTATION OF LDL

a cost, since its implementation poses nontrivial compilation and optimization problems. The various compilation techniques used for LDL are described in Bancilhon, et al. [1986], Saccá and Zaniolo [1986], Zaniolo [1985], and Zaniolo and Saccá [1987]. This paper will concentrate on the control and optimization problem, which is the problem of deriving a safe (terminating) and efficient execution plan for a given program.

The optimization of LDL programs poses a new set of challenging problems beyond those faced by relational systems. First of all, Horn clauses support, in addition to flat relational data, complex objects (e.g., hierarchies), lists, and heterogeneous structures. Secondly, new operators are needed to handle complex data, and constructs such as recursion, negation, sets, etcetera. Thirdly, the complexities of data and operations emphasize the need for new database statistics and new estimations of cost. Finally, the presence of evaluable functions and of recursive predicates with function symbols give the user 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 must guarantee that the resulting execution is safe.

This paper describes a fully integrated compile-time approach that ensures both safety and optimization to guarantee the amalgamation of the database functionality with the programming language functionality of LDL. Therefore, the LDL optimizer subsumes the basic control strategies used in relational systems as well as those used in Morris, et al. [1986]. In particular, for LDL programs that are equivalent to the usual join-project-select queries of relational systems, the LDL optimizer behaves as the optimizer of a relational system [Sellinger, et al., 1979].

We limit the discussion, in this paper, to the problem of optimizing the pure fixpoint semantics of Horn clause queries [Lloyd, 1984]. After setting up the definitions in Section 20.2, the optimization is characterized as a minimization problem based on a cost function over an execution space in Section 20.3. The execution model is discussed in Section 20.4, and used to define the execution space in Section 20.5. We outline our cost function assumptions in Section 20.6. The search space is detailed in Section 20.7 by extending the traditional approach to the nonrecursive case first, and then to recursion. The problem of safety is addressed in Section 20.8, where we extend the optimization algorithm to ensure safety.

20.2 DEFINITIONS

The knowledge base consists of a rule base and a database (also known as fact base). An example of rule base is given in Fig. 20.1.

Throughout this paper, we follow the notational convention that Pi's, Bi's, and f's are predicates, base predicates (i.e., predicate on a base relation), and function symbols, respectively. The Bi's are relations from the database
and the $P_i$'s are the derived predicates whose tuples (i.e., in the relation corresponding to that predicate) can be computed using the rules. Note that each rule contains the head of the rule and the body that defines the tuples that are contributed by this rule to the set of tuples associated with the head predicate. A rule may be recursive, in the sense that the definition in the body may depend on the head predicate, either directly by reference or transitively through a predicate referenced in the body. An example of a recursive rule is $R_{21}$.

In a given rule base, we say that a predicate $P$ implies a predicate $Q$, written $P \rightarrow Q$, if there is a rule with $Q$ as the head and $P$ in the body, or there exists a $P'$ where $P \rightarrow P'$ and $P' \rightarrow Q$ (transitivity). Then a predicate $P$, such that $P \rightarrow P$, will be called recursive. Two predicates $P$ and $Q$ are called mutually recursive if $P \rightarrow Q$ and $Q \rightarrow P$. Since this implication relationship is an equivalence relation, it can be used to partition the recursive predicates into disjoint subsets, which we will call recursive cliques. A clique $C_1$ is said to follow another clique $C_2$ if there exists a recursive predicate in $C_2$ that is used to define the clique $C_1$. Obviously the ‘follow’ relation is a partial order.

In a departure from previous approaches to compilation of logic, we make our optimization query specific. A query with indicated bound/unbound arguments (called binding) will be called a query form. Thus, $P_1(c,y)$? is a query form in which $c$ and $y$ denote a bound and unbound argument, respectively. Throughout this paper we use $x,y$ to denote variables and $c$ to denote a constant. We say that the optimization is query specific because the algorithm is repeated for each such query form. For instance, the query $P_1(x,y)$?, will be compiled and optimized separately from $P_1(c,y)$?. Indeed the execution strategy chosen for a query $P_1(x,y)$? may be inefficient for a query $P_1(c,y)$?, or an execution designed for $P_1(c,y)$? may be unsafe for $P_1(x,y)$?.

In general, we can define the notion of a binding for a predicate in a rule body based on a given permutation of the literals in the body. This process of using information from the prior literals was called sideways information passing (SIP) in Ullman [1985]. We note here that a given permutation is associated with a unique SIP.
20.3 THE OPTIMIZATION PROBLEM

We define as follows the optimization problem, as the minimization of the cost over a given execution space (i.e., the set of all allowed executions for a given query).

Logic Query Optimization Problem. Given a query \( Q \), an execution space \( E \), and a cost function defined over \( E \), find an execution \( pg \) in \( E \) that is of minimum cost; that is,

\[
\min_{pg \in E} \text{cost of } pg(Q)
\]

Any solution to this optimization problem can then be described along four main coordinates, as follows:

1. The model of an execution, \( pg \).
2. The definition of the execution space, \( E \), consisting of all allowable executions.
3. The cost functions that associate a cost estimate with each execution in \( E \).
4. The search strategy to determine the minimum cost execution in the given space.

The model of an execution represents the relevant aspects of the processing so that the execution space can be defined based on the properties of the execution. The designer must select the set of allowable executions over which the least cost execution is chosen. Obviously, the main trade-off here is that a very small execution space will eliminate many efficient executions, whereas a very large execution space will render the problem of optimization intractable, for a given search algorithm. In the next sections we describe the design of the execution model, the definition of the execution space, and the search algorithm. The cost formulas are in most cases system dependent. Thus we will consider the cost formulas as a black box, where the actual formulas are not discussed except for those assumptions that impact the global architecture of the system.

20.4 EXECUTION MODEL

LDL's target language is a relational algebra extended with additional constructs to handle complex terms and fixpoint computations. An execution over this target language can be modeled as a rooted directed graph, called
processing graph, as shown in Fig. 20.2 for the example of Fig. 20.1. Intuitively, leaf nodes (i.e., the nodes with nonzero indegree) of this graph correspond to operators and the results of their predecessors are the input operands. The representation in this form is similar to the predicate connection graph [Kellog & Travis, 1981], or rule graph [Ullman, 1985], except that we give specific semantics to the internal nodes, and use a notion of contraction for recursion as described in what follows.

In keeping with our relational algebra-based execution model, we map each AND node into a join and each OR node into a union. Recursion is implied by an edge to an ancestor or a node in the sibling subtree. We restrict our attention to fixpoint methods for recursion, that is, methods that implement recursive predicates by means of a least fixpoint operator. We assume that the fixed point operation of the recursive predicates in a clique is not computed in a piecemeal fashion (i.e., the fixpoint operation is atomic with respect to other operations in the processing tree). In order to model this property, we define the notion of a contraction. A contraction of a clique is the extrapolation of the traditional notion of an edge contraction in a graph. An edge is said to be contracted if it is deleted and its ends (i.e., nodes) are identified (i.e., merged). A clique is said to be contracted if all the edges of the clique are contracted. Intuitively, the contraction of a clique consists of replacing the set of nodes in the clique by a single node and associating all the edges in/out of any node in the clique with this new node, (as in Fig. 20.3), generically called Contracted Clique node (or CC node). As the structure of the rules in the clique will be needed for optimization, we associate the set of rules in the clique to this CC node. Intuitively, a CC node corresponds to the fixpoint operation for the clique, whose operands are the results of the predecessors.
It is easy to see that a contracted processing graph is acyclic (a DAG). Moreover, for ease of exposition we also assume that this graph is converted into a tree by replicating the children with multiple successors. In the rest of the paper we assume that the processing graph has been contracted and due to the preceding stipulation, we interchangeably use the terms processing graph and processing tree.

Associated with each node is a relation that is computed from the relations of its predecessors, by doing the operation (e.g., join, union) specified in the label. We use a square node to denote materialization of relations and a triangle node to denote the pipelining of the tuples. A pipelined execution, as the name implies, computes only those tuples for the subtree that are relevant to the operation for which this node is an operand. In the case of join, this computation is evaluated in a lazy fashion as follows: using the binding from the result of the subquery to the left of a subtree, a tuple for that subtree is generated. This binding is referred to as binding implied by the pipeline. Note that we impose a left to right order of execution. Subtrees that are rooted under a materialized node are computed bottom-up, without any sideways information passing; that is, the result of the subtree is computed completely before the ancestor operation is started.

Each interior node in the graph is also labeled by the method used (e.g., join method, recursion method, etc.). The set of labels for these nodes are restricted only by the availability of the techniques in the system. Further, we also allow the result of computing a subtree to be filtered/projected through a selection/restriction/projection predicate. We extend the labeling scheme to encode all such variations due to filtering and projecting. The label for a CC node is to specify the choices for the fixpoint operation, which are the choices for SIPs and recursive method to be used.

The execution corresponding to a processing tree proceeds bottom-up left to right as follows: The leftmost subtree whose children are all leaves is computed and the resulting relation replaces the subtree in the processing tree. The computation of this subtree is dependent on the type of the root node of the subtree—pipelined or materialized—as described before. If the subtree is rooted at a contracted clique node, then the fixed point result of the
recursive clique is computed, either in a pipelined fashion or in a materialized fashion; the former (i.e., pipelining) requires the use of techniques such as Magic Sets or Counting [Bancilhon, et al., 1985; Saccá & Zaniolo, 1986].

20.5 EXECUTION SPACE

Note that many processing trees can be generated for any given query and a given set of rules. These processing trees are logically equivalent to each other, since they return the same result; however very different costs may be associated with each tree, since each embodies critical decisions regarding the methods to be used for the operations, their ordering, and the intermediate relations to be materialized. The set of logically equivalent processing trees thus defines the execution space over which the optimization is performed using a cost model, which associates a cost to each execution. We define this space by the following equivalence preserving transformations:

1. **MP**: Materialize/Pipeline: A pipelined node can be changed to a materialized node and vice versa.
2. **FU**: Flatten/Unflatten: Flattening distributes a join over an union. The inverse transformation will be called unflatten. An example of this is shown in Fig. 20.4.
3. **PS**: PushSelect/PullSelect: A select can be piggy backed to a materialized or pipelined node and applied to the tuples as they are generated. Selects can be pushed into a nonrecursive operator (i.e., join or union that is not a part of a recursive cycle) in the obvious way.
4. **PP**: PushProject/PullProject: This transformation can be defined similar to the case of select.
5. **PR**: Permute: This transforms a given subtree by permuting the order of the subtrees. Note that the inverse of a permutation is defined by another permutation.
6. **PA**: Permute & Adorn: The recursive methods such as Magic Sets and Counting require a SIP for each rule in the clique to be specified and

**FIGURE 20.4**
EXAMPLE OF FLATTEN/UNFLATTEN

![Diagram of execution space transformations](image-url)
an adornment (i.e., the binding implied by the SIP) to be chosen for each recursive predicate.

7. **EL**: *Exchange Label*: Change the label of a join/union operation to another available method.

Each of the preceding transformational rules maps a processing tree into another equivalent processing tree and is also capable of mapping vice versa. We define an equivalence relation under a set of transformational rules $T$ as follows: a processing tree $p_1$ is equivalent to $p_2$ under $T$, if $p_2$ can be obtained by zero or more application of rules in $T$. The equivalence class (induced by said equivalence relation) defines our execution space. As an equivalence class (and therefore an execution space) is uniquely determined by a set of transformational rules, an execution space is referred to by a set notation: $\{T_i \mid T_i$ is a transformational rule defined above\}. For example, $\{MP, PR\}$, $\{MP, PR, PS, PP\}$ are execution spaces.

As mentioned before, the choice of proper execution space is a critical design decision. By limiting ourselves to the preceding transformations, we have excluded many other types of optimizations like peephole optimizations (as used in traditional optimization phase of a programming language compiler), semantic optimizations, et cetera. This is a reflection of the restrictions posed in the context of relational systems from which we have generalized and is not meant to imply that they are considered less important. As in the case of relational systems, these supplementable optimizations can also be used. Even in the realm of preceding transformations, we were unable to find an efficient strategy for the entire space. Consequently, we limit our discussion in this paper to the space defined by $\{MP, PS, PP, PR, PA, EL\}$ (i.e., flattening and unflattening are not allowed). As discussed in Section 20.8, programs can be constructed for which no safe (and therefore, no efficient) executions exist without flattening. Our experience with rule-based systems, however, has been that these are artificial situations the user can be expected to avoid without any additional inconvenience.

## 20.6 COST MODEL

The cost model assigns a cost of each processing tree, thereby ordering the executions. Typically, the cost spectrum of the executions in an execution space spans many orders of magnitude, even in the relational domain. We expect this to be magnified in the Horn clause domain. Thus "it is more important to avoid the worst executions than to obtain the best execution" is a maxim widely assumed by the query optimizer designers. The experience with relational systems has shown that the main purpose of a cost model is to differentiate between good and bad executions. In fact, it is known from the relational experience that even an inexact cost model can achieve this goal reasonably well.
The cost includes CPU, disk I/O, communication, and so on, which are combined into a single cost that is dependent on the particular system. We assume that a list of methods is available for each operation (join, union, and recursion), and for each method, we also assume the ability to compute the associated cost and the resulting cardinality. For the sake of this discussion, the cost can be viewed as some monotonically increasing function on the size of the operands. As the cost of an unsafe execution is to be modeled by an infinite cost, the cost function should guarantee an infinite cost if the size approaches infinity. This is used to encode the unsafe property of the execution.

Intuitively, the cost of an execution is the sum of the cost of individual operations. This amounts to summing up the cost for each node in the processing tree.

20.7 SEARCH SPACE

In this section, we discuss the problem of choosing the proper search space. The main trade-off here is that a very small search space will eliminate many efficient executions, whereas a large search space will render the problem of optimization intractable. We present the discussion by considering the search spaces for queries of increasing complexity: conjunctive queries, nonrecursive queries, and then recursive queries.

20.7.1 Conjunctive Queries

An important lesson learned from the implementation of relational database systems is that the execution space of a conjunctive query can be viewed as the orderings of joins (and therefore relations) [Sellinger, et al., 1979]. The gist of the relational optimization algorithm is as follows: For each permutation of the set of relations, choose a join method for each join and compute the cost. The result is the minimum cost permutation. Note that for a given permutation, the choice of join methods becomes a local decision; that is, the EL label is unique. Further, a selection or a projection can be pushed to the first operation on the relation without any loss of optimality, for a given ordering of joins. Thus the choices of preselect, project, et cetera are incorporated in the choice of the join method. Consequently, the actual search space used by the optimizer reduces to {MP, PR}, yet the chosen minimum cost processing tree is optimal in the execution space defined by {MP, PR, PS, PP, EL}. (Note that PA is inapplicable as there are no recursions). Further, the binding implied by the pipelining is also treated as selections and handled in a similar manner.

This exhaustive enumeration approach, taken in the relational context, essentially enumerates a search space that is combinatoric on $n$, the number of relations in the conjunct. The dynamic programming method presented
in Sellinger, et al. [1979] only improves this to $O(n^2 2^n)$ time by using $O(2^n)$ space. Naturally, this method becomes prohibitive when the join involves many relations. Consequently, database systems (e.g., SQL/DS, commercial INGRES) must limit the queries to no more than 10 or 15 joins.

In Krishnamurthy, et al. [1986], we presented a quadratic time algorithm that computes the optimal ordering of conjunctive queries. Another approach to searching the large search space is to use a stochastic algorithm, using a technique called Simulated Annealing [Ioannidis & Wong, 1987]. The application of these techniques is discussed in Krishnamurthy and Zaniolo [1988].

**20.7.2 Nonrecursive Queries**

We extend the exhaustive approach that was used in the case of conjunctive queries to the nonrecursive case. Recall that the processing graph for any execution of a nonrecursive query is a tree; that is, an AND/OR tree. The optimization algorithm presented here finds an execution in the execution space corresponding to \{MP, PS, PP, PR\}. Here, we disallow any flattening of the tree, thus limiting the search space to the canonical structure specified in the rule base. As in the case of conjunctive query optimization, we push select/project down to the first operation on the relation and thus limit the search space to \{MP, PR\}. We shall propose an algorithm to enumerate this search space.

Let us first consider the case when we materialize all the temporary results for each predicate in the rule base. As we do not allow flatten/unflatten transformation, we can proceed as follows: optimize a lowest subtree in the AND/OR tree. This subtree is a conjunctive query, as all children in this subtree are leaves (i.e., base relations) and we may use the exhaustive case algorithm of the previous section. After optimizing the subtree we replace the subtree by a “base relation” and repeat this process until the tree is reduced to a single node. It is easy to show that this algorithm exhausts the search space \{PR\}. Further, such an algorithm is reasonably efficient if the number of predicates in the body does not exceed 10–15.

In order to allow the execution to use the sideways information by choosing pipelined executions, we make the following observation. In the algorithm of the previous paragraph, all the subtrees were materialized; as a result, the binding pattern of the head of any rule was uniquely determined. Consequently, we could outline the preceding bottom-up algorithm using this unique binding for each subtree. If we do allow pipelined execution, then the subtree may be bound in different ways (depending on the ordering of the siblings of the root of the subtree), and the subtree may be optimized differently for different binding. Observe that the number of binding patterns for a predicate is purely dependent on the number of arguments of that predicate. So, the extension to the preceding bottom-up algorithm is to optimize each subtree for all possible binding and to use the cost of the appropriate binding when computing the cost of joining this subtree with its siblings. Obviously,
the maximum number of bindings is equal to the cardinality of the power set of the arguments. In order to avoid optimizing a subtree with a binding pattern that may never be used, a top-down algorithm has been devised and is reported in Krishnamurthy and Zaniolo [1988]. In any case, the algorithm is expected to be reasonably efficient for a small number of argument positions, \( k \), and a few number of predicates in the body, \( n \). The use of strategies other than exhaustive search is also discussed in Krishnamurthy and Zaniolo [1988].

### 20.7.3 Recursive Queries

In the last two sections we have seen that pushing selection and projections is a linchpin of nonrecursive optimization methods. This was used to reduce the search space from \{MP, PR, PS, PP\} to \{MP, PR\}. Unfortunately, this simple technique is frequently inapplicable to recursive predicates [Aho & Ullman, 1979]. Therefore a number of specialized implementation methods have been proposed to allow recursive predicates to take advantage of constants or bindings present in the goal. (The interested reader is referred to Bancilhon and Ramakrishnan [1985] for an overview.) Further, the same techniques are used to incorporate the notion of pipelining (i.e., sideways information passing).

In keeping with our algebra-based approach however, we will restrict our attention to fixpoint methods, i.e., methods that implement recursive predicates by means of a least fixpoint operator. In particular, we use the magic set method [Bancilhon, et al., 1985] and generalized counting method [Sacca & Zaniolo, 1986] that have been shown to produce some of the most efficient and general algorithms to support recursion [Bancilhon & Ramakrishnan, 1986]. Moreover, they are compatible with the optimization framework used in this paper, since we can now map a recursive Horn clause query into an equivalent expression of extended relational algebra operators and least fixpoint operators.

We extend the algorithm presented in the previous section to include the capability to optimize a recursive query, using a divide and conquer approach. Note that all the predicates in the same recursive clique must be solved together—they cannot be solved one at a time. In the processing graph, we propose to replace a CC node by a single node (materialized or pipelined). We thus obtain a nonrecursive processing graph that can be optimized with the techniques described in the previous section.

The bottom-up optimization algorithm is extended as follows: choose a clique that does not follow any other clique. For this clique, use the nonrecursive optimization algorithm to optimize and estimate the cost and size of the result. Replace the clique by a single node with the estimated cost and size and repeat the algorithm. In Fig. 20.5 we have elucidated this algorithm for a single clique example. Note that the result of the cost and size estimates of the nonrecursive subquery are needed for the computation of the cost and size estimate for a given CC node; similarly, the nonrecursive query above the CC node uses the cost and size estimates of the recursive clique by view-
FIGURE 20.5
R-OPT EXAMPLE

(a) NOTE that the recursion method used for P2 is seminative because there are no constants to push.

(b) NOTE that the recursion method used takes advantage of the sideways information passing and uses a method such as magic set or counting.

(c) Using NO binding

To get the optimal processing graph for the (P3, P2) ordering.
ing the entire clique as a single node. In this manner, the executions in
the search space can be enumerated. The details of the algorithm and the
characterization of the search space are discussed in Krishnamurthy and
Zaniolo [1988].

20.8 SAFETY PROBLEM

An important concern in implementing Horn clause queries is the issue of
safety. Comparison predicates represent a first example of predicates that
are potentially unsafe; recursive predicates with function symbols represent
the second one. Comparison predicates include predicates such as \( x > y \)
and equality predicates such as \( x = y + y^*z \). While comparison predicates
will be executed by calls to built-in routines, they can be formally viewed
as infinite relations defining, for example, all the pairs of integers satisfying
the relationship \( x > y \), or all the triplets satisfying the relationship \( x = y + y^*z \)
[Tsur & Zaniolo, 1986]. Thus, they can be used only when a sufficient number of
variables is known. For instance, for \( x > y \), both \( x \) and \( y \) need to be
known, while bound \( y \) and \( z \) values will suffice for \( x = y + y^*z \).

A similar situation occurs with recursive predicates with function symbols.
Here, the Herbrand universe is infinite, and new complex terms can be
generated at each step in the fixpoint computation, such that the resulting
execution may not terminate. This is, for instance, the case of a list-reverse
predicate, that cannot be computed unless at least one of the two arguments
is instantiated. Both for comparison predicates and recursive predicates with
function symbols, therefore, certain safe binding patterns will be given that
make the predicate safe. In order to deal with safety we propose to restrict
the search space of the optimization algorithm to those processing graphs
where at least one safe binding pattern is satisfied for each potentially unsafe
goal.

20.8.1 Derivation of Safe Binding Patterns

Patterns of argument bindings that ensure safety are simple to derive for com-
parison predicates. For instance, we can assume that for comparison predi-
cates other than equality, all variables must be bound before the predicate is
safe. When equality is involved in a form \( x = expression \), then we are ensured of
safety as soon as all the variables in \( expression \) are instantiated. Clearly, these
are only sufficient conditions and more general ones, for example, based
on combinations of comparison predicates, could be given—see for instance
Maier [1984]; but for each extension, a rapidly increasing price would have
to be paid in the algorithms used to detect safety and in the system routines
used to support these predicates at run-time. Indeed, the problem of deciding
safety for Horn clauses with comparison predicates is undecidable [Zaniolo,
1985]; even when no recursion is involved. On the other hand, safety based on safe binding patterns is easy to detect and adequate in most real-life situations.

A main advantage of using safe binding patterns for comparison predicates is that the approach also works for recursive predicates with function symbols. Here again, while the general problem of deciding safety is undecidable, sufficient safety criteria that can be checked efficiently have been given to guarantee safety of the various fixpoint methods [Bancilhon, et al., 1985; Saccá & Zaniolo, 1986]. A somewhat more complex approach is available when nested recursive predicates are involved [Ullman & Van Gelder, 1985]. In either case, however, the final outcome consists of a set of binding patterns that ensure safety, independent of the specific values of these bindings.

20.8.2 Searching for Safe Executions

As mentioned before, the optimizer enumerates all the possible permutations of the goals in the rules. For each permutation, the cost is evaluated and the minimum cost solution is maintained. All that is needed to ensure safety is that for each conditionally safe goal its bound arguments (i.e., the arguments that contain variables appearing in goals before this) satisfy a safe binding pattern. If this test succeeds, then the cost of each conditionally safe predicate is evaluated and the optimization algorithm proceeds as usual. If the test fails, the permutation is discarded; in practice this can be done by simply assigning an extremely high cost of unsafe goals and then letting the standard optimization algorithm do the pruning—if the cost of the end solution produced by the optimizer is not less than this extreme value, a proper message must inform the user that the query is unsafe. It is easy to show that this algorithm is correct.

20.8.3 Comparison with Previous Work

The approaches to safety proposed in Naish [1985] and Ait-kaci and Nasr [1986] are also based on reordering the goals in a given rule; but that is done at run-time by delaying goals when the number of instantiated arguments is insufficient to guarantee safety. This approach suffers from run-time overhead and cannot guarantee termination at compile time, or otherwise pinpoint the source of safety problems to the user—a very desirable feature since unsafe programs are typically incorrect ones. Our compile-time approach overcomes these problems and is more amenable to optimization.

The reader should however be aware of some of the limitations implicit in all approaches based on reordering of goals in rules. For instance, a query

\[
\begin{align*}
p(x, y, z), \ y = 2^* \ x \ ? \\
p(x, y, z) & \leftarrow \ x = 3, \ z = x^*y
\end{align*}
\]
is obviously safe \((x = 3, y = 6, z = 18)\), but cannot be computed under any permutation of goals in the rule. Thus both the approaches given in Naish [1985] and Ait-kaci and Nasr [1986] and our and/or optimization cum safety algorithm will fail to produce a safe execution for this query. Two other approaches, however, will succeed. One, described in Zaniolo [1986], determines whether there is a finite domain underlying the variables in the rules using an algorithm based on a functional dependency model. Safe queries are then processed in a bottom-up fashion with the help of "magic sets" which make the process safe. The second solution consists in flattening, whereby the three equalities are combined in a conjunct and properly processed in the obvious order. In practice, however, queries such as the preceding are seldom encountered and Prolog's experience suggests that they can be omitted from the language without compromising its expressive power.

### 20.9 CONCLUSION

This paper has explored the new and challenging problem of optimizing a Logic-based language for data intensive applications. Thus the first contribution of the paper consists in providing a formal statement of the problem and in clarifying the main design issues involved. The second contribution is the solution approach proposed, which (i) cleanly integrates the search for a minimum cost execution with the safety analysis and (ii) is solidly rooted in the experience and know-how acquired in optimizing relational systems. Therefore the LDL optimizer includes both the conjunctive query optimization technique of relational systems [Sellinger, 1979] and the safety-oriented techniques described in Morris, et al. [1986].

Common subexpression elimination [Grant & Minkel, 1982], which appears particularly useful when flattening occurs, is one of the optimization aspects not covered in this paper. A simple technique using a hill-climbing method is easy to superimpose on the proposed strategy, but more ambitious techniques provide a topic for future research. Further, an extrapolation of common subexpression in logic queries can be seen in the following example: let both goals \(P(a, b, X)\) and \(P(a, Y, c)\) occur in a query. Then it is conceivable that computing \(P(a, Y, X)\) once and restricting the result for each of the cases may be more efficient.

### REFERENCES


