Optimization in a Logic Based Language for Knowledge and Data Intensive Applications

Ravi Krishnamurthy *Carlo Zaniolo*

MCC, 3500 Balcones Center Dr., Austin, TX, 78759

Abstract

This paper describes the optimization approach taken to ensure the safe and efficient execution of applications written in LDL, which is a declarative language based on Horn Clause Logic and intended for data intensive and knowledge based applications. In order to generalize the strategy successfully used in relational database systems we first characterize the optimization problem in terms of its execution space, cost functions and search algorithm. Then we extend this framework to deal with rules, complex terms, recursion and various problems resulting from the richer expressive power of Logic. Among these is the termination problem (safety), whereby an unsafe execution is treated as an extreme case of poor execution.

1. Introduction

The Logic Data Language LDL, combines the expressive power of a high level logic based language (such as Prolog) with the non-navigational style of relational query languages, where the user need only supply a correct query, and the system is expected to devise an efficient execution strategy for it. Consequently, the query optimizer is given the responsibility of choosing an optimal execution --a function similar to that of an 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 pre-defined search space and to select a minimum cost execution in such a space.

A 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 a cost, since its implementation poses non trivial compilation and optimization problems. The various compilation techniques used for LDL were described in [BMSU85, SZ86, Za85, ZS87]. This paper concentrates on the optimization problem; i.e., the problem of devising an efficient execution strategy for the given query. The termination problem (safety) is also tackled in this context, since the lack of termination can be viewed as an extreme case of poor termination. Therefore, our optimization problem revisits the well-known problem of control in logic programs as per Kowalski's famous equation Algorithm = Logic + Control [Kw 79]. Several approaches to this arduous problems have been proposed in the past.

Prolog visits and expands the rule goals in a strictly lexicographical order; thus, it is up to the programmer to make sure that this order leads to a safe and efficient execution. Approaches, such as intelligent backtracking, eliminate some of the inefficiencies of Prolog without changing this basic control mechanism. However, there has been proposals for a more explicit control of logic programs-- e.g., by supporting several 'and' connectives, each eliciting different sequencing behavior from the interpreter [Per 82].

Of more direct interest to this paper are those approaches where the control of execution is exercised, at least in part, by the system. For instance, reasoning at the metalevel is proposed in [SG 85] as the best way for the system to determine which goal is to be expanded next. A second approach consists in freezing unsafe goals until a sufficient number of arguments become instantiated [Col 82, Nai 85, AN 86]. For instance, the evaluation of an arithmetic expression is delayed until its variables become instantiated. More general situations can be treated via mode declarations added to procedures. The execution of a goal is then postponed until some modes are satisfied, for the procedure unifying with the given goal. These modes, can either be given by the user, or automatically inferred by the system [Na 85]. Observe that, in all these approaches, the control is *dynamic*, i.e., exercised by the system at run time.

The approach explored in this paper is that of *static control*; the flow of the execution is predetermined at compile time and simply enacted at run time. The NAIL! system also follows a static strategy, which matches execution strategies with the given query and rule set by means of *capture rules* [Ull 85, MUV 86]. The test to determine whether a certain capture rule is applicable in a given situation is based upon the form of the rules and the patterns of argument bindings in the goal (adornments); for each applicable capture rule there exists a corresponding substantiation algorithm for materializing the captured goals. While the testing for capture rules can be organized in such a way that more specialized rules are tried before more general ones-- under the assumption that more specialized rules produce more efficient execution strategies -- the notion of cost driven optimization is not part of the NAIL! system. Thus it appears that this system is more effective in dealing with the safety problem than with the optimization problem. This is consistent with the fact that the NAIL! system does not handle directly simple conjunctive queries; these are passed to an underlying off-the-shelf relational database system for query optimization.

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 [MUV 86]. 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[Sel 79].

The technical challenges posed by the LDL optimizer follow from its expressive power extending far beyond that of relational query languages. Indeed, in addition non recursive queries and flat relational data, Horn Clauses include recursive definitions and complex objects, such as hierarchies, lists and heterogeneous structures. Beyond that, LDL supports additional constructs including stratified negation [BN 87], set operators and predicates [TZ 86, BN 87], and updates [NK

87]. Therefore, new operators are needed to handle complex data, and constructs such as recursion, negation, sets, etc.. Moreover, 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.

In this we limit the discussion to the problem of optimizing the pure fixpoint semantics of Horn clause queries [Llo 84]. After setting up the definitions in Section 2, the optimization is characterized as a minimization problem based on a cost function over an execution space in Section 3. The execution model is discussed in Section 4, using which the execution space is defined in Section 5. We outline our cost function assumptions in Section 6. The search strategy is detailed in Section 7 by extending the traditional approach to the nonrecursive case first; and then extended to include recursion. The problem of safety is addressed in section 8, where we extend the optimization algorithm to ensure safety.

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 Figure 2-1 . Throughout this paper, we follow the notational

```
Query is P1(x,y)?
R1 : P1(x,y) <-- P2(x,x1), P3(x1,y).
R21: P2(x,y) <-- B21(x,x1), P2(x1,y1), B22(y1,y).
R22: P2(x,y) <-- P4(x,y).
R3 : P3(x,y) <-- B31(x,x1), B32(x1,y).
R4 : P4(x,y) <-- B41(x,x1), P2(x1,y).
```

Figure 2-1: Rule Base

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 Pi'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 (i.e., the predicate to the left of the arrow) 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 R21.

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 be used to partition the recursive predicates into disjoint subsets, which we will call *recursive cliques*. A clique C1 is said to *follow* another clique C2 if there exists a recursive predicate in C2 that is used to define the clique C1. Obviously the follow relation is a partial order due the definition of clique.

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, P1(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, P1(x,y)?, will be compiled and optimized separately from P1(c,y)?. Indeed the execution strategy chosen for a query P1(x,y)? may be inefficient for a query P1(c,y)? , or an execution designed for P1(c,y)? may be unsafe for P1(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 [Ull 85]. We note here that a given permutation is associated with a unique SIP.

3. The Optimization Problem

We define 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). This is formally stated below.

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; i.e.

$$\underset{pg \in E}{MIN} [\text{cost of } pg(Q)]$$

Any solution to the above optimization problem can then be described along four main coordinates, as follows:
i) the model of an execution, pg;
ii) the definition of the execution space, E, consisting of all allowable executions;
iii) the cost functions which associate a cost estimate with each point of the execution space; and
iv) 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 formulae are in most cases system dependent. Thus we will consider the cost formulae as a black box, where the actual formulae are not discussed except for those assumptions that impact the global architecture of the system.

4. Execution Model

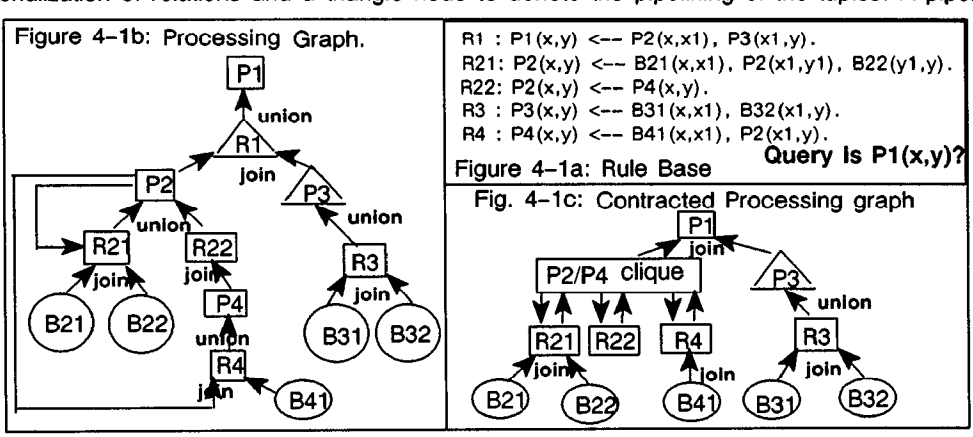
LDL's target language is a relational algebra extended with additional constructs to handle complex terms and fixpoint computations. An execution over this target language can be modelled as a rooted directed graph, called '*processing graph*', as shown in Figure 4-1b for the example of

Figure 2-1. Intuitively, leaf nodes (i.e., the nodes with non-zero in-degree) 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 [KT 81], or rule graph [Ull 85], except that we give specific semantics to the internal nodes, and use a notion of contraction for recursion as described below.

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, i.e., 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 piece-meal 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 Figure 4-1c), 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 correspond 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 the above 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: a tuple for a subtree is generated using the binding from the result of the subquery to the *left* of that subtree. 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; i.e., 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 above. 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 [BMSU 85, SZ 86].

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 Figure 4-2.
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.

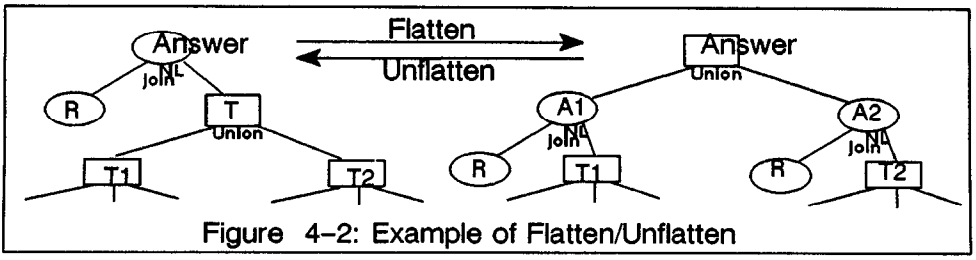


Figure 4-2: Example of Flatten/Unflatten

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 an adornment to be chosen for each recursive predicate. As we shall see in Section 7, a given permutation for each rule determines the SIPs for the clique as well as an adorned program. For each adorned program there is a set of applicable recursive methods (e.g., Semi-naive, Magic Sets, Counting).

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

Each of the above transformational rules map 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 p1 is equivalent to p2 under T, if p2 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: {Ti | Ti 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 above transformations, we have excluded many other types of optimizations like peep-hole optimizations (as used in traditional optimization phase of a programming language compiler), semantic optimizations, etc. 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 above 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 8, programs can be constructed for which no safe (and therefore, no efficient) executions exists without flattening. Our experience with rule based systems, however, has been that these are artificial situations which the user can be expected to avoid without any additional inconvenience.

6. Cost Model:

The cost model assigns a cost to 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", a maxim widely assumed by the query optimizer designers. The experience with relational system 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, etc., 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.

7. Search Strategies

We first outline the generic strategies that are known based on the experience with the relational systems. As a typical relational query is a conjunctive query, we discuss these strategies in the context of conjunctive queries. Then we generalize these strategies to the case of nonrecursive queries (i.e., AND/OR tree) and then to the complete Horn clause queries where we tackle the case for recursion.

7.1. Generic Strategies:

An important lesson learnt 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) [Sel 79]. 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 method becomes a local decision; i.e., 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 choice of preselect, preproject, etc. 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 above 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 [Sel 79] only improves this to $O(n*2^n)$ 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. As an alternative to exhaustive search, we consider two other methods.

In [KBZ 86], we presented a quadratic time algorithm that computes the optimal ordering of conjunctive queries when the query is acyclic and the cost function satisfies a linearity property called the Adjacent Sequence Interchange (ASI) property. Further, this algorithm was extended to include cyclic queries and other cost models. The resulting algorithm has proved to be heuristically effective for cyclic queries as well as other cost models [Vil 87]. The comparison was made by randomly picking queries and states of the database and then comparing the results of the quadratic time and exhaustive algorithms. The results showed that the quadratic algorithm chooses the optimal permutation in most cases and in more than 90% of the cases, it produces no worse than twice/thrice the optimal. These results have been shown to have a statistical confidence of 95% with a 3% error.

Another approach to searching the large search space is to use a stochastic algorithm. Intuitively, the minimum cost permutation can be found by picking, randomly, a "large" number of permutations from the search space and choosing the minimum cost permutation. Obviously, the number of permutations that need to be chosen approaches the size of the search space for a reasonable assurance of obtaining the minimum. This number is claimed to be much smaller by using a technique called Simulated Annealing [IW 87]. We use this technique to the optimization of conjunctive queries as follows. For any given permutation, define a neighbor to be any permutation that differs in exactly two places (i.e., two positions in one permutation is interchanged to get the other). It is easy to prove that the closure of the neighbor (equivalence) relation is indeed the set of all permutations (i.e., the execution space for conjunctive queries). The simulated annealing can then be viewed as a "random" walk of the execution space using this neighbor relation. If we ignore the annealing parameters, then the neighbor relation completely characterizes the simulated annealing process. We shall use this notion to characterize the strategy using simulated annealing.

In short, we have summarized three generic strategies: exhaustive, quadratic and stochastic. The main trade-offs amongst these strategies is between efficiency (i.e., time complexity) and flexibility. Note that the quadratic strategy is the most efficient, whereas it is least flexible in terms of the possible modifications to cost functions, query structure, etc. Our goal is to present a design for the search strategy that is capable of using multiple strategies interchangeably. The main reason for requiring the system to be flexible is that the system is initially intended as an experimental vehicle since there is no prior experience in the design of an optimizer for a logic language and the field of logic languages is still in its infancy; thus new ideas will be forthcoming that the design should be capable of incorporating into the system.

7.2. Nonrecursive Queries

Initially, we extend the exhaustive strategy that was used in the case of conjunctive queries to the nonrecursive case, which is then extended to the other two strategies. Extrapolating from the conjunctive case, selects/projects are always pushed down any number of levels for non-recursive rules by simply migrating to the lower level rules the constraints inherited from the upper rules.

Simple compile-time rule-rewriting techniques can be used to push selection/projection down into non-recursive rules.

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 (e.g., base relations) and we may use the exhaustive strategy. 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} and finds an optimal execution over {PR, PS, PP, EL}. Further, as in the relational systems, such an algorithm is *reasonably efficient* if number of predicates in the body does not exceed 10–15.

In order to allow the execution to use the side ways information by choosing pipelined executions, we make the following observation. Because all the subtrees were materialized above, the binding pattern of the head of any rule was uniquely determined. Consequently, we were able to outline the above 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. Consequently, the subtree may be optimized differently. 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 above bottom-up algorithm is to optimize each subtree for all possible binding and to use the cost for 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 can be devised. Such an algorithm has been shown in Figure 7–1.

This algorithm guarantees that each subtree is optimized exactly ONCE for each binding. The worst case time complexity can be computed as follows: Suppose that there are k variables per predicate, N total predicates in the rule base, and n be the number of predicates per conjunct, then we have the following worst case estimates:

$2^k * n!$ = worst case cost of reducing one AND-subtree.

N/n = number of AND-subtrees.

Thus, the total worst case complexity of this algorithm is $O(N * 2^k * n!)$. However, using the dynamic programming approach for the enumeration of the conjunctive search [Sel 79], we reduce the $n!$ permutations to 2^n choices. Thus the worst case complexity becomes $O(N * 2^k * 2^n)$. Normally, the number of arguments per predicate (k) is usually less than five and number of predicates per conjunct (n) is usually less than 10. For these values of k and n, we conclude the feasibility of this approach based on the experience from commercial database systems.

NR-OPT: Compute a processing tree for a nonrecursive logic query.
Input is a processing tree rooted at a node N.
Output is an optimized processing graph.

1) Node N is an AND node, say As:
   I) For each permutation of the sequence of subtrees,
      Using the binding implied by the permutation do:
      a) For each OR-subtree Os of As do: Compute NR-OPT(Os).
      b) Compute the cost for this permutation using the cost model.
      c) Maintain the minimum cost permutation.
   II) Return cost, cardinality, and the graph for the minimum cost processing graph.

2) Node N is an OR node say Os:
   1) IF this subtree, Os, has NOT already been optimized for this binding
   THEN do:
      a) For each AND-subtree As of Os : Compute NR-OPT(As).
      b) Compute the cost of the union of the children.
      c) record the cost, cardinality, graph, etc., for Os, indexed by the binding.
   2) ELSE read cost, cardinality, graph, etc., for Os, based on the binding.

Figure 7-1: NR-OPT algorithm for non-recursive query.

The algorithm of Figure 7-1 becomes impractical for large values of k and/or n. The main practical concern is n since the number of arguments in recursive predicates is either small, or reducible to a small number by the use of complex terms. We discuss below how the algorithm in Figure 7-1 can be easily modified to take advantage of the quadratic strategy [KBZ 86] or of simulated annealing.

Note that the step 1) of the algorithm NR-OPT is responsible for the exponential behavior w.r.t. n. This step is a generalization of the optimization search for conjunctive query. Consequently, replacing the exhaustive strategy with the stochastic strategy is straightforward, whereas the incorporation of quadratic strategy is little more involved requiring the generalization of the ASI property. As this involves more detail discussion of the ASI property, we omit this discussion in this paper for reasons of brevity. In either case, the resulting algorithm is capable of optimizing rule bases that have large n. Also note that the choice of strategies may be made per rule. That is, the more efficient strategies are used only if the rules actually has a large number of literals in the body of the rule. The optimizer for LDL currently has all three strategies implemented whereas only the exhaustive strategy is integrated into the system.

Even though we do not expect k to be very large, it would be comforting if we could find an approximation for this case too. This remains a topic for further research.

7.3. Recursive queries

In the last two sections we have seen that pushing selection and projection is a linchpin of non-recursive 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 [AU 85]. 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 [BR 86] for an overview.) Further, the same techniques are used to incorporate the notion of pipelining (i.e., side ways information passing). We use the *magic set method* [BMSU 85] and *generalized counting method* [SZ 86] that have been shown to produce some of the most efficient [BR 86] and general algorithms to support recursion in the LDL optimizer. 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.

Consider the recursive predicate plus the binding used in the operation corresponding to the successor node to the CC node. This recursive predicate plus the binding can be viewed as a *"subquery"* for the CC node. We replicate the recursive rules in the clique as follows: for each rule (say with head predicate P) and for each binding pattern, a, the rule is replicated by renaming the head as 'P.a'. An adornment is a binding pattern that is associated with a literal.

Given a subquery for the CC node and a SIP per (replicated) rule (i.e., the permutation for the literals in the body for each rule), then we can adorn the program, similar to the adornment of rules in [BMSU 86, Ull 85] as follows: We construct the adorned version of the program *Pgm'* for the original program *Pgm* by replacing the derived predicates in the body by the adorned versions. The process starts from the given subquery whose adornments determine an adorned version of the predicate. For each adorned predicate, P.a, and for each rule that has P.a in the head, we generate an adorned version for the rule as described below and add it to *Pgm'*. We then mark P.a. Note that the adorned version of a rule may generate additional predicates that are adorned. The process terminates when no unmarked adorned predicates are left.

The adornment for a recursive predicate in the body is assigned as follows: an argument is bound if the variable(s) in the argument occurs either in a bound argument of the head literal or in a goal that precedes it in the chosen permutation. All other arguments of this literal are adorned as free. Each literal P that is associated with a binding a is renamed as 'P.a'. We present below, the adorned programs for the query forms sg.bf and sg.bb, in which the chosen SIP for all replicated rules is self evident.

Original Rule: sg (X,Y) <- up(X,X1), sg(Y1,X1), dn(Y1,Y)

Adorned clique for the query sg.bf: ('bf' is the binding)

sg.bf (X,Y) <- up(X,X1), sg.fb(Y1,X1), dn(Y1,Y)
sg.fb (X,Y) <- dn(Y1,Y), sg.bf(Y1,X1), up(X,X1)

Adorned clique for the query sg.bb:

sg.bb (X,Y) <- up(X,X1), sg.fb(Y1,X1), dn(Y1,Y)
sg.fb (X,Y) <- dn(Y1,Y), sg.bf(Y1,X1), up(X,X1)
sg.bf (X,Y) <- up(X,X1), sg.fb(Y1,X1), dn(Y1,Y)

Note that for a given subquery and a permutation for each rule in the clique, the resulting adorned program is unique. Further, for a given adorned program, the transformed program by Magic Sets or Counting is also unique. As a result, the execution (and the associated cost) is uniquely determined, for a given cost and size estimates for all the literals (in the rules of the clique) that are not in the clique. From this we can conclude that the space of executions that are to be enumerated is defined by the different permutations of the rules in the clique. In other words, if there are *nc* rules in the clique, then each possible cross product of *nc* permutations defines a *c-permutation.* For each c-permutation, and a subquery there is an adorned program. Note that all of them are not distinct, but collectively they exhaust the possible adorned programs.

We extend the algorithm presented in the previous section to include the capability to optimize a recursive query. When a subtree rooted at a CC node is to be optimized, the choice is in adorning

OPT: Compute a processing tree for a recurrsive logic query.
Input is a processing tree rooted at a node N.
Output is an optimized processing graph.

1) Node N is an AND node, say As:
   I) For each permutation of the sequence of subtrees,
      Using the binding implied by the permutation do:
      a) For each OR-subtree Os of As do: Compute OPT(Os).
      b) Compute the cost for this permutation using the cost model.
      c) Maintain the minimum cost permutation.

   II) Return cost, cardinality, and the graph for the minimum cost processing graph.

2) Node N is an OR node say Os:
   1) IF this subtree, Os, has NOT already been optimized for this binding
   THEN do:
      a) For each AND-subtree As of Os : Compute OPT(As).
      b) Compute the cost of the union of the children.
      c) record the cost, cardinality, graph, etc., for Os, indexed by the binding.
   2) ELSE read cost, cardinality, graph, etc., for Os, based on the binding.

3) Node N is a CC node, say Cs:
   I) For each c-permutations of the rules in Cs do:
      i) Adorn relevant rules
      ii) For each adorned literal, Os, NOT in the clique Cs do: Compute OPT(Os).
      iii) For each applicable recursive method do:
         Compute cost of the clique Cs and maintain the minimum cost solution.

   II) Return the minimum cost soulution

Figure 7-2: OPT algorithm for recursive queries.

the node with the proper label. We have to enumerate all the c-permutations for the clique. For each such assignment of c-permutations, the rules are adorned and the literals not in the clique are optimized for the respective adornment. Then the cost of the fixpoint operations is computed for each applicable recursive method (e.g., Magic Sets, Naive, Counting) and the minimum cost execution is chosen. This algorithm is shown in Figure 7-2. Note that the optimization for the clique is, once again, dependent only on the adornment of the subquery. Therefore, the result of the optimization can be saved and used to avoid recomputation as it was done in the OR-subtree case. This has been omitted for brevity of the algorithm.

It is easy to show that the algorithm finds the optimal execution in the execution space {MP, PR, PS, EA, PA} while searching the space {MP, PR, PA}. Note that the recursive techniques such as Magic Sets and Counting can only handle pushing selections. In order to push projections we use the techniques proposed in [RBK 87], which is used as a preprocessing step to the optimizer. It is easy to see that enumeration of all the permutations for the rules in the clique is impractical even for small number of rules in the clique. It is conjectured by many researchers that the mutual recursions are not common and complicated ones are used even less. So if this conjecture is true then exhaustive search may not be impractical.

Nevertheless, we are interested in being able to optimize larger class of queries. For this we present the use of the stochastic strategy. Note that if the enumeration of the search space consisting of all possible c-permutations of a clique (in case 3 of the algorithm) is improved, then the algorithm can be used for a larger class of queries. Further note that we observed that by specifying the neighbor relation for a given execution, such that the closure of this relation defines the space to be searched, we can characterize the simulated annealing process. We present such a neighbor relation here. Let us define a neighbor of a c-permutation, CP1, to be another cross product of *nc* permutations, CP2, such that all but one of these *nc* permutations in CP2 are identical to the ones in CP1 and the one that differs, is obtainable by interchanging exactly two literals in the permutation. Obviously, the closure of this (equivalence) relation is the space that we set out to search. Consequently, we have characterized the simulated annealing process and the iterative loop choosing the c-permutations in the algorithm OPT can be replaced by the simulated annealing process.

An interesting open question is the incorporation of a polynomial time algorithm by superimposing some linearity property on the cost function for a recursive clique, as it was done for the conjunctive case in [KBZ 86].

8. Safety Problem:

Safety is a serious concern in implementing Horn clause queries. Any evaluable predicates (e.g., comparison predicates like x>y, x=y+y*z), and recursive predicates with function symbols are examples of potentially unsafe predicates. While an evaluable predicate 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 [TZ 86]. Consequently, these predicates may result in unsafe executions in two ways: 1) the result of the query is infinite; 2) the execution requires the computation of a rule resulting in an

infinite intermediate result. The former is termed the lack of *finite answer* and the latter the lack of *effective computability or EC.* Note that the answer may be finite even if a rule is not effectively computable. In this section we outline our approach with the emphasis on the interaction with the optimizer. For a more complete treatise on this topic see [KRS 87].

8.1. Checking for safety:

Patterns of argument bindings that ensure EC are simple to derive for comparison predicates. For instance, we can assume that for comparison predicates 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 EC as soon as all the variables in *expression* are instantiated. These are only sufficient conditions and more general ones – e.g., based on combinations of comparison predicates – could be given (see for instance [M 84]). But for each extension of a sufficient condition, a rapidly increasing price would have to be paid in the algorithms used to detect EC and in the system routines used to support these predicates at run time. Indeed, the problem of deciding EC for Horn clauses with comparison predicates is undecidable even when no recursion is involved [Za 86] . On the other hand, EC based on safe binding patterns is easy to detect. Thus, deriving more general sufficient conditions for ensuring EC that is easy to check is an important problem facing the optimizer designer.

If all rules of a nonrecursive query are effectively computable, then the answer is finite. However, for a recursive query, each bottom-up application of any rule may be effectively computable, but the answer may be infinite due to unbounded iterations required for a fixpoint operator. In order to guarantee that the number of iterations are finite for each recursive clique, a well-founded order based on some monotonicity property must be derived. For example, if a list is traversed recursively, then "the size of the list is monotonically decreasing with a bound of an empty list" is a well-founded order. This forms the well-founded condition for termination of the iteration. Some methods to derive the monotonicity property are discussed in [Nai 85, UV 85, SZ 86]. A general algorithm to ensure the existence of a well-founded condition is outlined in [KRS 87]. As these are only sufficient conditions, they do not necessarily detect all safe executions. Consequently, more general monotonicity properties must be either inferred from the program or declared by the user in some form. These are topics of future research.

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 recorded. All that is needed to ensure safety is that EC is guaranteed for each rule and well founded order(s) is associated with each recursive clique. If both these tests succeed, then the cost of this particular execution is estimated and the optimization algorithm proceeds as usual. If the tests fails, the permutation is discarded. In practice, this can be done by simply assigning an extremely high cost to unsafe goals and then let 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.

8.3 Comparison with Previous Work

The approaches to safety proposed in [Col 82, Nai 85, AN 86] is 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

p(x, y, z), y= 2*x ?

on the rule

p(x, y, z) <-- x=3, z=x*y

is obviously finite since the only answer is <x=3, y=6, z=18>. However, this answer cannot be computed under any permutation of goals in the rule. Thus both the approach given in [Col 82, Nai 85, AN 86] and the above optimization cum safety algorithm will fail to produce a safe execution for this query. Two other approaches, however, will succeed. One, described in [Za 86], 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 referred to earlier.

This example clarifies the drawbacks that follow from our expedient decision of not pursuing fllattening in the first version of the optimizer. Some flattening is being considered for later versions of the optimizer. Observe that, unlike previous approaches to control where such strategic decisions were wired-in into the system, an extension of the LDL optimizer to support flattening only requires adding another equivalence-preserving transformation.

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 [Sel 79] and the safety-oriented techniques described in [MVU 86]. Finally the paper has introduced two new algorithms, one for optimizing non-recursive Horn Clauses, the other for recursive ones, and proposed three search strategies as the vehicle for implementing these algorithms. The first one is an exhaustive search that, be-

cause of its complete nature, supplies the basis for assessing the soundness of the overall approach and the effectiveness of the two alternative algorithms (quadratic strategy and simulated annealing) which are to be used to tame the computational complexity of the problem. The results of early experiments were also reported that confirm the heuristic effectiveness of the more efficient algorithms.

Common subexpression elimination[GM 82], 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 technique 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.

Acknowledgments:
We are grateful to Shamim Naqvi for inspiring discussions. We are also grateful to an anonymous referee for a very detailed comments.

References:

[AN 86] Ait-Kaci, H. and R. Nasr, "Residuation: a Paradigm for Integrating Logic and Functional Programming," submitted for publication.

[AU 79] Aho, A. and J. Ullman, Universality of Data Retrieval Languages, *Proc. POPL Conf.*, San Antonio, TX, 1979.

[BMSU85] Bancilhon, F., D, Maier, Y. Sagiv and Ullman, Magic Sets and other Strange Ways to Implements Logic Programs, *Proc. 5-th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pp. 1-16, 1986.

[BR 86] Bancilhon, F., and R. Ramakrishan, An Amateur's Introduction to Recursive Query Processing Strategies, *Proc. 1986 ACM-SIGMOD Intl. Conf. on Mgt. of Data*, pp. 16-52, 1986.

[BN 87] Beeri, C., S. Naqvi, R. Ramakrishnan, O. Shmueli, S. Tsur, Sets and Negation in a Logic Database Language, *Proc. 6-th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1987.

[Col 82] Colmemauer, A. et al., Prolog II: Reference Manual and Theoretical Model, Groupe d'Intelligence artificielle, Faculte de Sciences de Lumin, 1982.

[GM 82] Grant, J. and Minker J., On Optimizing the Evaluation of a Set of Expressions, *Int. Journal of Computer and Information Science, 11, 3 (1982), 179-189.*

[IW 87] Ioannidis, Y. E, Wong, E, Query Optimization by Simulated Annealing, *Proc. 1987 ACM-SIGMOD Intl. Conf. on Mgt. oof Data*, San Francisco, 1987.

[Kw 79] Kowalski, R.A., "Algorithm = Logic + Control", *CACM, 22, 7*, pp. 424-436, (1979).

[KBZ 86] Krishnamurthy, R., Boral, H., Zaniolo, C. Optimization of Nonrecursive Queries, *Proc. of 12th VLDB*, Kyoto, Japan, 1986.

[KRS 87] Krishnamurthy, R., R. Ramakrishnan. O. Shmueli, "A Framework for Testing Safety and Effective Computability", MCC Report 1987 and also submitted for external publication.

[KT 81] Kellog, C., and Travis, L. Reasoning with data in a deductively augmented database system, in *Advances in Database Theory: Vol 1*, H.Gallaire, J. Minker, and J. Nicholas eds., Plenum Press, New York, 1981, pp 261–298.

[Llo 84] Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, 1984.

[M 84] Maier, D., *The Theory of Relational Databases*, (pp. 553–542), Comp. Science Press, 1984.

[MUV 86] K. Morris, J. D. Ullman and A. Van Gelder, Design Overview of the Nail! System, *Proc. Third Int. Symposium on Logic Programming*, pp. 127–139, 1986.

[Nai 85] Naish, L., Negation and Control in Prolog, Ph. D. Thesis, Dept. of CS, Univ. of Melbourne, Austr., 1985.

[NK 87] Naqvi Shamim and R. Krishnamurthy, Semantics of Updates in Logic Programming, Workshop on Database and Programming Languages, Roscoff, France 1987.

[Per 82] Pereira Luis Moniz, Logic Control with Logic, UNL Report 2/82 (1982).

[RBK 87] Ramakrishnan, R, C. Beeri, R. Krishnamurthy, Optimizing Existential Queries, MCC Technical Report, 1987, (also submitted for external publication).

[Sel 79] Sellinger, P.G. et. al. Access Path Selection in a Relational Database Management System., *Proc. 1979 ACM–SIGMOD Intl. Conf. on Mgt. of Data*, pp. 23–34, 1979.

[SG 85] Smith, D. E. and M. R. Genesereth, Ordering Conjunctive Queries, *Artificial Intelligence* **26**, pp. 171–185, 1985.

[SZ 86] Sacca', D. and C. Zaniolo, The Generalized Counting Method for Recursive Logic Queries, *Proc. ICDT '86 --Int. Conf. on Database Theory*, Rome, Italy, 1986.

[TZ 86] Tsur, S. and C. Zaniolo, LDL: A Logic–Based Data Language,*Proc. of 12th VLDB*, Kyoto, Japan, 1986.

[Ull 85] Ullman, J. D., Implementation of logical query languages for databases, *TODS*, 10, 3, (1985), 289–321.

[UV 85] Ullman, J.D. and A. Van Gelder, Testing Applicability of Top–Down Capture Rules, Stanford Univ. Report STAN–CS–85–146, 1985.

[Vil 87] Villarreal, E., "Evaluation of an O(N**2) Method for Query Optimization", MS Thesis, Dept. of Computer Science, Univ. of Texas at Austin, Austin, TX.

[Za 85] Zaniolo, C. The representation and deductive retrieval of complex objects, *Proc. of 11th VLDB*, pp. 458–469, 1985.

[Za 86] Zaniolo, C., Safety and Compilation of Non–Recursive Horn Clauses, *Proc. First Int. Conf. on Expert Database Systems*, Charleston, S.C., 1986.

[ZS 87] Zaniolo C. and D. Sacca', "Rule Rewriting Methods for Efficient Implementations of Horn Logic," MCC Technical Report 1987, submitted for publication.