Towards an Open Architecture for $\mathcal{LDL}$

Danette Chimenti Ruben Gamboa Ravi Krishnamurthy

MCC, 3500 West Balcones Center Drive, Austin, Texas 78759

danette@mcc.com,ruben@mcc.com,ravi@mcc.com

Abstract

We extend $\mathcal{LDL}$ to allow programs to call external procedures and vice versa. This extension allows the modularization of $\mathcal{LDL}$, since external predicates are equivalent to external procedures written in $\mathcal{LDL}$. External predicates are viewed as infinite relations so that the traditional semantics of logic programs remain applicable. To avoid computing infinite relations, well-formedness conditions for programs in extended $\mathcal{LDL}$ are given. The traditional optimization framework can still be used; it is only necessary to define a new set of cost functions capable of handling the infinite relations. The problem of interfacing $\mathcal{LDL}$ programs with external procedures—exchanging complex objects and returning multiple solutions—is discussed. Thus, we provide a general framework to allow logic programs to interact with external procedures without sacrificing amenities such as optimization, safety, etc. This approach forms the basis for the implementation of externals and modules in the $\mathcal{LDL}$ compiler and optimizer at MCC.

1 Introduction

An $\mathcal{LDL}$[1] program calling a graphics routine or a windowing system calling an $\mathcal{LDL}$ program are obvious cases that motivate the incorporation of external predicates into $\mathcal{LDL}$. External predicates in $\mathcal{LDL}$ also allow a user to develop large programs by—what we call—*hot-spot refinement*. The user writes a large $\mathcal{LDL}$ program, validates its correctness and identifies the hot-spots; i.e., predicates in the program that are highly time consuming. Then, he can rewrite those hot-spots more efficiently in a procedural language such as *C*, maintaining the rest of the program in $\mathcal{LDL}$. Naturally, the correctness, optimization, safety and other amenities provided by the $\mathcal{LDL}$ environment should not be sacrificed. Hence, the optimizer must understand the nature of such external procedures (referred to as computed predicates), and optimize programs that use them.

Yet another reason for including external predicates in $\mathcal{LDL}$ comes from the optimizer. The time complexity of an optimization algorithm is typically exponential on the number of predicate occurrences. By modularizing the program and optimizing each module separately, compilation time can be reduced. Intuitively, the modularization allows the user to provide a unique binding pattern for the global (imported) predicate, thus simplifying the optimization task.

Computed relations were incorporated into relational databases in [MW81]. In this paper we present an extension of $\mathcal{LDL}$ to allow computed predicates by addressing the following three issues: 1) Providing $\mathcal{LDL}$ language extensions to allow external predicates; 2) Interfacing $\mathcal{LDL}$ programs with external predicates; 3) Optimizing $\mathcal{LDL}$ programs using external predicates. Since Datalog is a simpler subset of $\mathcal{LDL}$, we first discuss these issues in the context of Datalog and, subsequently, in the full context of $\mathcal{LDL}$.

We extend Datalog to Datalog^E, which allows the use of externals. We also provide well-formedness conditions for Datalog^E programs. By viewing external predicates as infinite relations, we show that the semantics of Datalog programs are preserved in the transition to Datalog^E. Then, we show that Datalog^E programs can be optimized in the traditional framework [Sell79, KBZ86], simply by redefining the cost functions to handle such infinite relations.

$\mathcal{LDL}$ is viewed as an extension of Datalog, where more complex operations and data types are allowed. We show that these extensions do not affect the semantics and optimization results presented in the context of Datalog^E. However, the presence of complex objects, e.g., sets and functor objects, does pose new problems, since the external predicates can manipulate them. We define enhanced cost functions, capable of handling complex objects as arguments to external predicates. We also provide protocols to interface $\mathcal{LDL}$ programs with external predicates and discuss implementation issues.

After overviewing the background in Section 2, the discussion in the context of Datalog is presented in Section 3. The issues in the context of $\mathcal{LDL}$ are discussed in Section 4, followed by the conclusion.

[1] $\mathcal{LDL}$, Logic Data Language, is a Horn clause based language with extensions for negation, set operations, updates, etc.

Proceedings of the Fifteenth International
Conference on Very Large Data Bases

Amsterdam, 1989

2 Background

We briefly introduce Datalog, then $\mathcal{LDL}$ as an extension of Datalog, and, finally, the framework for optimization.

2.1 Datalog

We assume that the reader is familiar with Datalog and the associated definitions for rule, base/derived predicates, predicate occurrence and the mutually recursive property. A *rule base*, RB is a finite set of rules. The *extensional database*, EDB, is a finite set of finite *base* relations. A *query* is a predicate occurrence written as $?q(\ldots)$. For a given query, Q, we denote a *program* P by a triple (EDB, RB, Q). Given a program $P = (EDB, RB, Q)$, the answer to Q can be defined in the usual way.

Datalog is semantically defined using the bottom-up model, which coincides with the declarative model when the program is viewed as a first-order formula[Ll84]. However, the program execution does not have to be bottom-up. It is only required that the execution preserve the bottom-up semantics; i.e., the answers returned must be exactly the same as those returned under the bottom-up semantics. The use of sideways information passing [Ull85] and memo-ing (as is done in Prolog) results in many different executions (including top-down) that preserve the bottom-up semantics.

2.2 $\mathcal{LDL}$

Conceptually, we can view the execution of an $\mathcal{LDL}$ program as the manipulation of objects. Therefore, the $\mathcal{LDL}$ language can be viewed as an extension of Datalog in two categories: first, the addition of constructs such as negation, nondeterministic choice (i.e., declarative cut,!)[KN88a], updates[NK88], etc.; and second, the addition of functors, sets with their associated grouping constructor, etc. These extensions enhance the computational capability of $\mathcal{LDL}$ programs and the data model of the objects in the $\mathcal{LDL}$ programs, respectively. As we shall be interested mainly in the additions to the data model, only these extensions are reviewed below. We model $\mathcal{LDL}$ objects abstractly as follows:

1. An *atomic object* can be an integer, a string, etc.
2. A *functor object* is recursively defined as an object of the form $f(object_1, \ldots, object_n)$, where f is an n-ary functor symbol. A *tuple object* is a functor object with no functor name.
3. A *set object* is a (not necessarily homogeneous) collection of objects. For example, $\{(john, 10K), \ldots\}$ is a set object whose elements are tuple objects.

2.3 Optimization

The optimization of $\mathcal{LDL}$ (and therefore, Datalog) programs can be described abstractly as follows:

> Given a query Q, an execution space E, and a cost function defined over E, find an execution e in E that is of minimum cost; i.e.,
>
> $$\min_{e \in E}[Cost\ of\ e\ for\ Q]$$

Any solution to the above problem can be characterized by choosing: 1) an execution model and, therefore, an execution space; 2) a cost model; and 3) a search strategy.

The execution model encodes the decisions regarding the ordering of the joins, join methods, materialization strategy, etc. The cost model computes the execution cost. The search strategy is used to enumerate the search space, while the minimum cost execution is being discovered. These three choices are not independent; the choice of one can affect the others. For example, if a linear cost model is used, as in [KBZ86], then the search strategy can enumerate a quadratic space; on the other hand, an exponential space is enumerated if a general cost model is used, as in the case of commercial database management systems. Our discussion in this paper does not depend on either the execution space or the search strategy employed. However, it does depend on how the cost model is used in the optimization algorithm.

For brevity, we assume the exhaustive search as described in [KZ88]. As argued in that paper, the cost model for any execution can be composed of elementary cost functions for join, projection, and union, i.e., the traditional operations in relational queries. We can see how these cost functions are sufficient to model a recursive query by noticing that, in a bottom-up execution, recursion is implemented as a fixpoint iteration of a sequence of relational queries, which can be easily modelled using the traditional functions. Therefore, we limit the discussions in this paper to these cost functions.

The traditional cost functions use the description of the relations, e.g., cardinality or selectivity, to compute the cost. Observe that the operands for these functions may be intermediate relations, whose descriptions must be computed. We model the descriptor of a relation using the framework presented in [CP84], where it is called a profile. Such a descriptor encodes all the information about the relation that is needed for the cost functions.

The *descriptor* of an n-ary relation is a tuple containing $(b_i, c_i, i = 0, 1, \ldots, n)$, where b_i and c_i are the bag (the cardinality without removing duplicates) and cardinality of the i^{th} attribute of the relation, and b_0 and c_0 are the bag and cardinality of the relation itself. The redundancy in the descriptors is assumed for ease of exposition. For example, consider the employee relation, $emp(Name, Age)$, containing 100 tuples with 50 distinctly named employees in 20 distinct age groups. The descriptor for this relation is (100,100, 100,50, 100,20).

Let the set of all descriptors be $\mathcal{D}$, and let I be the set of cost values denoted by integers. We are interested in two functions for each operation, σ, $\text{COST}_\sigma : \mathcal{D} \times \mathcal{D} \rightarrow I$

and $\mathrm{DESC}_\sigma : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$. Intuitively, the COST_σ function computes the cost of applying the binary operation σ to two objects, and DESC_σ gives a descriptor for the resulting object. The functions for the unary operators are similarly defined. In this paper, we shall be mainly concerned with the DESC function.

3 Extending Datalog

In this section we incorporate the notions of modules and computed relations into Datalog to obtain Datalog^E and present well-formedness conditions for Datalog^E programs. Next, the semantics of Datalog^E programs is formalized. We then tackle the problem of optimizing Datalog^E programs by requiring schematic information for each computed predicate. It is shown that the traditional optimization framework can use this information to optimize Datalog^E programs.

3.1 Datalog^E

A Datalog^E program is characterized by the four-tuple (EDB, CDB, IDB, Q). The extensional database and the query are defined as before. The *computed database*, CDB, and the *intentional database*, IDB, are formalized below.

The *computed database* is a finite set of *infinite* relations. For example, $<, >$ and other arithmetic predicates fall into this category. Conceptually, any external procedure, e.g., a program written in C, can be viewed as an infinite relation with some constraints on the input and output arguments. Except for these constraints (to be discussed later), the usage of CDB predicates is identical to that of EDB predicates.

The *intentional database* is a finite set of pairs (RB, EQ), where RB is a rule base, and EQ is the set of exported predicates from RB. The exported predicates are query forms on the (derived) predicates defined in the RB. For convenience, we assume the exported predicates from any two RBs are disjoint. The exported predicates over all the RBs form the set of *global predicates* for the program. Any (derived) predicate in an RB that is not a base, computed or global predicate is called a *local predicate*. Without loss of generality, we assume that the set of predicates in the four categories, i.e., base, computed, global and local, partitions the set of all predicates in the entire program. Let us also redefine $DB = EDB \cup IDB \cup CDB$.

Global and computed predicates are referred to as *external predicates*, giving the rationale for Datalog^E.

3.2 Well–Formed Datalog^E Programs

We now introduce a class of Datalog^E programs. Our goal is to identify the programs that can be executed without generating infinite relations for external predicates and that can be compiled separately.

First, we recall the notion of *finiteness constraint* (FC) from [KRS88]. Intuitively, this captures the notion of the input/output requirements of an external predicate. A computed predicate p is said to satisfy a *finiteness constraint* of the form $\overline{X} \rightarrow \overline{Y}$ if and only if for each tuple T in p, the set of tuples $\{S[\overline{Y}] \mid S \in p \text{ and } S[\overline{X}] = T[\overline{X}]\}$, is finite.[2]

The finiteness constraint must be explicitly stated in the case of a computed predicate. This is done by specifying the input arguments $\overline{X}$ of the procedure corresponding to the computed predicate. It is assumed that this procedure can compute the (finite) set of values for $\overline{Y}$ in finite time, given a value for $\overline{X}$. It is the responsibility of the Datalog^E environment to invoke the procedure only after a value for $\overline{X}$ is known. This imposes an ordering on the predicate occurrences of a Datalog^E program. Consider the following example that uses the external predicates `mult` and `add`.

$$\mathtt{p(X,Y,Z)} \leftarrow \mathtt{mult(X,X,XX), add(XX,YY,Z), mult(Y,Y,YY), b1(X), b2(Y).}$$

We assume the FC $\{X, Y\} \rightarrow \{Z\}$ for both `mult(X,Y,Z)` and `add(X,Y,Z)`. Thus, none of the procedures can be invoked until some values are taken from the base relations, `b1` and `b2`. Moreover, the `add` procedure must be invoked after both occurrences of `mult`. The order of the `mult` invocations, however, can be chosen arbitrarily.

It is possible that the predicate occurrences of a Datalog^E program cannot be ordered in such a way that the input bindings required by an external predicate are supplied. Consider the following example:

$$\mathtt{p(X)} \leftarrow \mathtt{add(X,Y,Z).}$$

In such a case, the Datalog^E program is not well–formed. This can be recognized at compile time by using the following *covering condition*.

Given the query Q and an ordering per rule of the predicate occurrences, we construct an adorned version of the program as in [Ull85]. In an adorned program, every argument of each predicate occurrence (as well as the head predicate) is marked as bound or free. A program is *covered* if there exists an adorned program, i.e., an ordering per rule of the predicate occurrences, in which the input arguments of each external (both computed and global) predicate is marked as bound.

The FC need not be specified for global predicates, since it can be inferred during compilation. Briefly, the input bindings are specified in the query form for which the program is compiled. During compilation, safety analysis guarantees that the resulting execution will produce a finite result when applied to a finite database[KRS88]. In other words, if the arguments $x_1, x_2, \ldots, x_k$ are used for input and $y_1, y_2, \ldots, y_m$ are the output arguments specified, the compiler guarantees

[2]Note this notion is strictly weaker than that of functional dependency.

the FC $\{x_1, x_2, \ldots, x_k\} \rightarrow \{y_1, y_2, \ldots, y_m\}$. This leads to our earlier claim that the arguments of the global predicates in a Datalog^E program must also be marked as bound for the program to be covered.

In order to compile a global predicate separately, we must ensure that all information necessary in a global compilation scheme is available. In particular, we would like to guarantee the safety of the execution chosen by the system. Consider the following example involving two mutually recursive predicates.

```
p(X,Y) ← b(X,Y).
p(X,Y) ← q(X,Z), b(Z,Y).
q(X,Y) ← p(X,Z), b(Z,Y).
```

Clearly, a safe execution can be found if the predicates are defined in the same module. However, separate compilation could result in an unsafe execution, i.e., it could result in an infinite left–recursive descent.

Hence, we disallow mutually recursive global predicates in well–formed Datalog^E programs. A *well-partitioned* Datalog^E program is one in which any two mutually recursive global predicates are defined in the same RB.

We can now define the class of well–formed Datalog^E programs. A Datalog^E program $P = (EDB, CDB, IDB, Q)$ is *well–formed* if and only if it is covered and well-partitioned.

3.3 Semantics

In this section, we recall the declarative model semantics for Datalog programs and show that it is unaffected by the addition of external predicates. For the sake of brevity, our discussion in this section is informal.

A *substitution* is a non-empty finite set of ordered pairs $\{X_1/v_1, \ldots, X_n/v_n\}$ such that $(\forall 1 \leq i \leq n)$ X_i is a distinct variable, v_i is an atom. We view a substitution as a mapping on variables. If σ is a substitution and X a variable, the result of *applying* σ to X is defined as

$$X\sigma = \begin{cases} v, & \text{if } (X/v) \in \sigma \\ X, & \text{otherwise} \end{cases}$$

We informally explain the notion of the universe, U, of a Datalog (and Datalog^E) program P. Initially we take U_0 to be the set of all atomic objects in P. U_1 is defined as the set of all possible predicate occurrences that can be formed from the elements in U_0. U is then defined to be U_1 unioned with U_0.

We now define the satisfaction of a rule. Consider a rule of the form $p(\ldots) \leftarrow p_1(\overline{X_1}), \ldots, p_m(\overline{X_m})$ Let σ be a substitution and $I \subseteq U$. The rule is satisfied if either there is some $[p_i(\overline{X_i})\sigma] \notin I$, or if whenever $[p_i(\overline{X_i})\sigma] \in I, \forall i \in \{1, \ldots, m\}$ then $p(\ldots)\sigma$ is also in I.

Given a collection of rules RB, $I \subseteq U$ is a model of RB if I satisfies all the rules in RB. A model M of a given collection of rules RB is said to be minimal if no proper subset of M is also a model of RB. It has been shown that Datalog programs have a unique minimal model. This unique model is defined to be the meaning of the program.

Note that in the semantics defined above, finiteness of the base relations is irrelevant. Since we view external predicates as infinite relations, and use them interchangeably with base predicates in the programs, the same semantics defined above carry over to Datalog^E. Therefore, we use the same unique minimal model to be the meaning of Datalog^E programs.

The semantic definition is traditionally complemented with a constructed model semantics, that, in some sense, provides an operational semantics for the program. The constructed model semantics is not relevant to the topics discussed in the remainder of this paper and is, therefore, omitted.

3.4 Optimization of Datalog^E

The execution space and search strategy used in the optimization algorithm for Datalog can also be used for Datalog^E, since the usage and semantics of base and external predicates are very similar. We need only redefine the cost functions to be capable of evaluating the cost of operations on external predicates. Observe that the descriptor for a relation cannot model an infinite relation; therefore, we must define a new descriptor for the infinite relation. We then formalize the two functions to compute the cost and resulting descriptor for each basic operation, i.e., join, union and projection.

3.4.1 Descriptors for Infinite Relations

Let us denote the descriptor on infinite (finite) relations as *i-descriptors* (*f-descriptors*). An f-descriptor was previously defined. An i-descriptor is defined as a pair (SE,FE), where the SE and FE are *selectivity* and *fan-out expressions*. SE and FE are arithmetic expressions in which the operands are constants, cardinality, and bag of the input arguments[3]; and the operators are +, -, *, /, max, and min.[4] SE and FE are arithmetic expressions that evaluate to positive numbers less than and greater than 1, respectively. Intuitively, the SE represents the probability that a given binding for the input arguments will select a nonempty subset of the infinite relation corresponding to the computed predicate. The FE is the expected number of tuples in the selected subset of the infinite relation, conditioned on the fact that a nonempty subset was chosen.[5]

[3] We have assumed that *card* and *bag* are the only schematic information of interest. Any other information can be incorporated in the same way.

[4] Max and min are binary prefix operators, whereas the others are infix operators.

[5] Note that the traditional assumptions of uniformity and independence are implicit in the definition of the SE and FE.

Pred	Bound	SE	FE
$lt(X,Y)$	X,Y	$\frac{1}{3}$	1
$add(X,Y,Z)$	X,Y	1	1
$sqrt(X,Y)$	X	$\frac{1}{2}$	2
$eq(X,Y)$	X,Y	$\frac{1}{max[card(@1),card(@2)]}$	1

Figure 1: SE and FE for some computed predicates.

A list of SE and FE for some computed predicates has been given in Figure 1. Note that the predicate *lt* is truly a predicate in the sense that it does not return any values, whereas *add* and *sqrt* are single-valued and multi-valued functions, respectively. The *lt* and *add* are very selective in the sense that all values for the input arguments choose a nonempty subset of their respective infinite relation. On the other hand, the *sqrt* predicate selects a nonempty subset only if the input argument is positive. As a result, the SE for this predicate is $\frac{1}{2}$. The use of *max* and *card* are exemplified in the descriptor for the *eq* predicate. The SE for this predicate is motivated by the desire to treat the following two rules identically.

```
p(X,Y) ← b1(X,X1),b2(X1,Y).
p(X,Y) ← b1(X,X1),b2(X2,Y),eq(X1,X2).
```

Note that card(@n) refers to the cardinality of the argument where the variable in the n^{th} input argument is bound. Thus, the definition of card(@1) for *eq* is the cardinality of the second argument of *b*1, i.e., where the 1^{st} input argument of *eq*, namely $X1$, is bound. Any references to the descriptor information of an input argument, e.g., *card* or *bag*, are resolved in this manner.

Note that in the above example for *eq*, the SE evaluates to a value depending upon the *card* of *b*1 and *b*2. This underscores the need to have an expression, not a constant, for SE and FE.

An i-descriptor can be used to model an f-descriptor with an FC restriction. Consider the $emp(Name, Age)$ relation, containing 100 tuples with 50 distinct names and 20 distinct ages, i.e., the descriptor is $(100, 100, 100, 50, 100, 20)$. Also consider the query $?emp(N, 30)$. Using the traditional assumptions regarding the uniformity of the distribution of values and the independence of attributes, we can estimate the result to contain 5 tuples with 5 distinct names and exactly one age, namely 30. Obviously, the result can be represented by an f-descriptor. We can also represent this result using the following i-descriptor, with finiteness constraint $Age \rightarrow Name$.

$$SE = min(1, \frac{20}{card(@2)}), \qquad FE = 5.$$

The selectivity expression is unity if the cardinality of the Age attribute, i.e., 20, is larger than the number of distinct values in the argument where the variable occurring in the Age attribute is bound. Otherwise, only 20 of those tuples will join with the *emp* relation, resulting in the selectivity $20/card(@2)$. The FE is five, since an age that exists in the relation has, on the average, five names associated with it. This example can be generalized to the following transformation.

f-to-i transformation:

Given a relation $r(A1, A2)$ with the f-descriptor $(b_0, c_0, b_1, c_1, b_2, c_2)$, the i-descriptor corresponding to r with respect to the FC $A1 \rightarrow A2$, is as follows:

$$SE = min(1, \frac{c_1}{card(@1)}), \qquad FE = \frac{c_0}{c_1}.$$

Intuitively, the SE and FE model the subset of the relation having a single value for the first argument. We shall use the f-to-i transformation in the definition of the cost and descriptor functions. Note that $A1$ and $A2$ could be vectors, and the above result could be extrapolated in the obvious way.

To completely model an f-descriptor of an n-ary relation with i-descriptors, we need to have an i-descriptor for each set of input arguments, i.e., 2^n i-descriptors. Therefore, an f-descriptor can be seen as a more concise representation of 2^n i-descriptors.

In summary, we have infinite and finite relations, with their respective descriptors. We redefine the set of all descriptors $\mathcal{D}$ to consist of the union of the set of all i-descriptors, $\mathcal{D}_i$, and the set of all f-descriptors, $\mathcal{D}_f$.

3.4.2 Cost and Descriptor Functions

In this subsection, we provide cost and descriptor functions capable of mapping from objects in $\mathcal{D}_i$. We assume that the schema has the following information for each computed predicate:

1. Name of the predicate;
2. FC, i.e., input arguments;
3. SE;
4. FE;
5. Cost of computing the answer for a value for each input argument.
6. Descriptor expression for output arguments.

As all of this information is needed to compute the cost and the resulting descriptor, we shall redefine the descriptor for an infinite object to consist of the above hextuple. The cost in the descriptor is the cost of computing the answer, using this external procedure, for a given value for each input argument. This cost is based on the unit cost which is used (by the optimizer) for the operations on non-external predicates. In $\mathcal{LDL}$ we use the cost of comparing a value as the unit cost. The database administrator who sets up the schema with the above information is expected to guesstimate the cost using the same units. The accuracy of this information is important only to an extent, as evident from the following desiderata used by most designers of query optimizers: *It is more important to avoid the worst execution than to obtain the best execution.* The rationale for this desiderata is that the spectrum of the cost of executions spans many orders of magnitude. Therefore,

the cost supplied in the descriptor need only be a reasonable estimate. Note that we shall not need the descriptor expression for output arguments until the next section, when we extend the language to include complex objects such as sets, etc.

The operations are canonically viewed as follows:

1. Join of a and b:
   $$p(X,Y,Z) \leftarrow a(X,Y), b(Y,Z).$$
2. Union of a and b:
   $$p(X,Y) \leftarrow a(X,Y).$$
   $$p(X,Y) \leftarrow b(X,Y).$$
3. Projection of c:
   $$p(X,Y) \leftarrow c(X,Y,Z).$$

For each of the above operations, we describe the functions using the following finite and infinite relations:

1. infinite relations: $e1(X,Y)$ and $e2(X,Y)$ with $X \rightarrow Y$ as their FC; SE1 and SE2, FE1 and FE2, Cost1 and Cost2 as their SE, FE and Cost, respectively.
2. finite relations: $rx(X,Y)$ and $ry(X,Y)$ with (bx_0, cx_0, bx_1, cx_1, bx_2, cx_2) and (by_0, cy_0, by_1, cy_1, by_2, cy_2) as their descriptors.

The functions are defined below, using a generic cost function for operations on two finite relations, e.g., Jcost(rx,ry) as the cost of joining relations rx and ry. Even though these functions are for operations on the above binary predicates, the implementation of the $\mathcal{LDL}$ optimizer has successfully used this approach for predicates of arbitrary arity. For convenience, we define the following multiplication factors:

$$mf1 = SE1 * FE1; \qquad mf2 = SE2 * FE2.$$

1. Join:

- $\mathcal{D}_f \times \mathcal{D}_f$: Join of rx and ry:
  Assume Cost = Jcost(rx, ry).
  Assume Desc: Jdesc(rx, ry).

- $\mathcal{D}_i \times \mathcal{D}_i$: Join of $e1$ and $e2$:
  Cost = Cost1 + $mf1$*Cost2.
  Desc: SE = SE1 *ne; FE = FE1 *$mf2/ne$,
  where ne =(1 - (1-SE2)^{FE1}) is the prob. of non-empty join of FE1 tuples with $e2$.

- $\mathcal{D}_i \times \mathcal{D}_f$: Join of $e1$ and ry:
  Cost = Cost1 + Jcost(ry, t),
  where t is a $mf1$-tuple relation.
  Desc: (B,C, B,1, B,mf1, B,min(mf1,cy_2)),
  where B = by_0*mf, C = bc_0*mf.

- $\mathcal{D}_f \times \mathcal{D}_i$: Join of rx and $e2$:
  Cost = bx_2 * Cost2 + Jcost(rx, t),
  where t is a $mf2$-tuple relation.
  Desc: similar to the $\mathcal{D}_i \times \mathcal{D}_f$ case.

2. Union

- $\mathcal{D}_f \times \mathcal{D}_f$: Union of rx and ry:
  Assume Cost = Ucost(rx, ry).
  Assume Desc: Udesc(rx, ry).

- $\mathcal{D}_i \times \mathcal{D}_i$: Union of $e1$ and $e2$:
  Cost = Cost1 + Cost2.
  Desc: SE = SE1 + SE2 - SE1 * SE2;
  FE = ($mf1$ + $mf2$)/SE.

- $\mathcal{D}_i \times \mathcal{D}_f$ and $\mathcal{D}_f \times \mathcal{D}_i$: Union of $e1$ and ry:
  Cost = Cost1 + Ucost(ry,t).
  Desc: SE = SE1 + min[1 , (c_1/$card$(@1))];
  FE = FE1 + (c_0/c_1).

3. Projection:

- $\mathcal{D}_f$: Pcost(rx), Pdesc(rx);

- $\mathcal{D}_i$: SE = SE1; FE = FE1; Cost = Pcost(t),
  where t is a $mf1$-tuple relation.

For brevity, we omit the detailed derivations of these formulae. Here we provide the intuitions for a few of the less obvious formulae and use these formulae to illustrate the impact on the optimizer.

The result of join is finite except in the join of $e1$ and $e2$. In this case, the SE for the result is the probability of selecting a nonempty result, which is the product of SE1 and the probability that at least one of the FE1 tuples from $e1$ join with $e2$; the FE reflects the fact that the fanout expression is conditional (i.e, the result is non-empty).

Union results in a finite relation only when both operands are also finite relations. The descriptor function for the union of $e1$ and ry uses the f-to-i transformation to convert the f-descriptor to an i-descriptor. The two i-descriptors are then unioned.

Since the result of the above functions may be an i-descriptor, some of the (local) derived predicates may also have to be modelled as i-descriptors. Therefore, these functions are applicable whenever two predicates with the appropriate descriptors are operated on.

The computation of the descriptor function that results in an i-descriptor is symbolic. Consider the case of joining $e1$ and $e2$, in which the resulting SE is an expression that is formed by multiplying the expressions symbolically. Such a symbolic manipulation of descriptor information was not needed in the traditional cost functions for optimizing relational queries. Such an expression is evaluated only when it is 'applied' to a base relation, as in the $\mathcal{D}_i \times \mathcal{D}_f$, and the $\mathcal{D}_f \times \mathcal{D}_i$ cases.

Given the above formulae for join, union and projection, it is natural to address the comprehensiveness of the descriptors for infinite relations. To do so, we must determine whether the information in the descriptors is sufficient for any Datalog program and whether the estimation is 'reasonably' accurate. The former is obvious from our observation that any bottom-up processing is composed of a sequence of these operations. The latter requires the validation of the formulae and has not yet been done. In this paper, the formulae have been provided simply to illustrate the approach. The derivations and validations will be addressed in a more complete paper.

In summary, we have proposed a model for the i-descriptor and shown that it can be used to define the cost and descriptor functions, which are then used in the global optimization algorithm. Note that the model is sufficiently general in the sense that the expressions can be extended to operate on any new schematic information that may be of interest. In that sense, we have presented a new framework for integrating external predicates into Datalog.

3.5 Implementation Issues

In order for the reader to understand the proposal in the proper context, we briefly outline the relevant implementation issues. The compiler translates each Datalog^E global predicate into a *C* procedure with references to the external procedures for all other external predicates. These references are resolved in the linking phase to produce an executable. An answer is produced when the executable is invoked with appropriate input parameters.

Some mechanism must be provided to convert objects between the Datalog^E and external representations. Moreover, an external procedure should be able to return (receive) one or more results to (from) a Datalog^E program. In the following section, we outline a more general implementation in the context of $\mathcal{LDL}$, which subsumes Datalog^E.

4 Extending $\mathcal{LDL}$

$\mathcal{LDL}$ has a more general data model than Datalog, in that it allows complex objects, such as functors, tuples and sets. The $\mathcal{LDL}$ semantics are presented in depth in [NT89]. As with Datalog^E, these semantics do not change with the addition of modules and externals. The syntax of modules and externals is presented in [CGK89a]. In the remainder of this paper, we concentrate on the implementation issues involved for the communication and optimization problems.

4.1 Implementation Issues

There are two orthogonal problems that must be solved in order to allow an $\mathcal{LDL}$ program to interact with a conventional language, such as *C*. There is an inherent incompatibility between the data types traditionally associated with procedural languages and the rich data types of $\mathcal{LDL}$. Moreover, procedural languages are ill-suited to the multiple solution paradigm basic to declarative languages, such as $\mathcal{LDL}$. In this section, we outline a solution which addresses both of these problems.

As mentioned in the background section, the complex data in $\mathcal{LDL}$ can be recursively defined using a set-tuple model. Basically, any object at the highest level is either a set, a tuple or an atom. So, we provide a set of primitives to compose and dissect any complex object partitioned along these three categories. The composing primitives include *create_atom, create_tuple, create_set* and *add_to_set* with appropriate parameters to define the constructed object. The *add_to_set* primitive is used to construct a set after it is initialized to the empty set using *create_set*. The dissecting primitives for atoms are conversion routines from the internal Datalog^E environment to the *C* environment. A tuple is dissected using the *get_ith_attr* and *get_name* primitives. Sets are dissected using the primitives *open_cursor* and *get_next_elem*, which can be used in a `while` loop of a *C* program.

In order to receive multiple solutions from a conventional subprogram, we must provide a facility to collect all the results with a single subroutine call. To do this, the calling $\mathcal{LDL}$ program initializes a temporary relation which is passed to the subprogram. The results are then stored into this relation by the subprogram. The calling program can read the tuples in this temporary relation as it would any other relation. In particular, the various $\mathcal{LDL}$ execution models [CGK89b], such as pipelined or materialized executions, apply, as do the various compile-time (local) optimizations, such as existential query optimization [RBK88] and intelligent backtracking. The following example shows how a subroutine can use these primitives to return multiple arguments.

```
procedure square_roots(Rel, x);
        Compute ±√x and add_to_set(Rel);
return;
```

$\mathcal{LDL}$ uses this same method when calling a global procedure. Hence, a conventional procedure can use this paradigm to call $\mathcal{LDL}$ by creating a temporary relation, calling the $\mathcal{LDL}$ compiled procedure, and, subsequently, reading the answers from the relation. We illustrate this in the following example.

```
procedure process_ancestors(x);
    create_set(Rel, 2);
    call LDL ancestor procedure with Rel and x;
    open_cursor(Rel);
    while (more tuples in Rel)
        let t be get_next_elem(Rel);
        /* Process t */
    end while;
return;
```

Note that the *Process t* above may include dissecting the tuple returned and processing the attributes of the tuple.

Observe that if a procedure returns a single value, then the above paradigm requires it to be circumscribed by a routine that puts the result in a relation and returns it to the $\mathcal{LDL}$ environment. We recognize that many *C* procedures that are single valued functions can be integrated in a simpler fashion; i.e., use them as functions to

compute the answer. For these functions, the $\mathcal{LDL}$ environment takes care of the necessary conversions and totally avoids the intermediate step of putting the result in a relation. Note that the user is still required to declare this function in the schema and provide the descriptor information.

4.2 Optimization of $\mathcal{LDL}$

As in the case of Datalog^E, the addition of external predicates to $\mathcal{LDL}$ does not affect the execution space or the search strategy. The effect is limited to the cost functions. Even in the cost functions, only the computation of the descriptor function is affected.

The addition of complex objects to Datalog^E necessitates the extension of the i- and f-descriptors. The f-descriptor must be capable of describing the complex object. The i-descriptor is modified by adding information required to compute the descriptors for the output arguments. Note that the FC, SE and FE remain unchanged as these are not affected by the addition of complex objects to Datalog^E. In this section, we make the simplifying assumption that sets cannot have heterogeneous objects; e.g., a set cannot contain both f(X,Y) and g(X). Thus, we can model the complex object as nested sets. Note that the implementation of the $\mathcal{LDL}$ optimizer does not make this assumption.

We define the representation of a complex object using a grammar-like[6] formalism.

$$S \rightarrow \{\ S\ \} \mid [S, S, \ldots, S] \mid atom$$

Intuitively, the representation describes the nesting with {...} and [...] denoting a set and a tuple, respectively.

A nested set is associated with an f-descriptor similar to the one for flat relations. Each object/subobject is attributed with a bag and cardinality for a corresponding 'flat' relation. This relation is constructed as follows:

> First, identify the immediate set containing the subobject. For this set, append a unique number corresponding to each of the parents of this set to its elements and then union all the occurrences of this set. The bag and cardinality of this subobject are the corresponding values in the constructed set, i.e., relation.

Consider the following object:

{ [dno, loc, { [ename, { [cname, cage] }] }] }

This is a set of departments, each containing the department number (dno), location (loc) and the set of employees. Each employee has a name (ename) and a set of children. Each child has a name (cname) and age (cage). The bag and cardinality of the employee set is computed as if it were an atomic object in the department relation. The bag and cardinality for ename is computed for the set of all employees over all departments. Similarly, the bag and cardinality for the subobject, the children set, is computed over all departments. The bag and cardinality for cname is computed over all departments and over all employees. Intuitively, these numbers reflect the bag and cardinality of the universal relation corresponding to the above hierarchical schema. This allows the optimizer to estimate the result of joining two complex subobjects, especially at different levels.

Cost functions must be changed due to both the additional operations on complex objects and the revised set of f-descriptors. These changes are orthogonal to the problems incurred due to the addition of external predicates. Therefore, we avoid any discussion on this topic, except to note that the cost function extensions are based on the operations viewed as if they were on the conceptual universal relation mentioned above.

The addition of complex objects to the language also allows the external predicates to return complex objects for output arguments. Consequently, the optimizer must be provided with the structure of the output object. This may not be independent of the context, as evident from the following example. Consider the *eq* predicate, with only the first argument as input, i.e., an assignment predicate. The structure of the second argument may be atom, tuple or any complex object depending on the input argument. As another example, consider the predicate that returns the first argument of a tuple. The structure of this argument cannot be known a priori. So, once again, we use an expression for each output argument to describe its structure and to enable the optimizer to compute its descriptor correctly. The following grammar-like notation describes the expression syntax:

$$\begin{aligned} S \ \rightarrow\ & \{\ S\ \} \mid [S, S, \ldots, S] \mid atom \mid \\ & @I \mid element(S) \mid attr(S, I) \end{aligned}$$

where I is an integer; and *atom*, '@', *element*, *attr* are terminal symbols of the grammar. Note that the first three rules provide the mechanism to construct more complex objects. The fourth rule provides a reference to an input argument, i.e., I^{th} argument. This means that the descriptor is identical to the I^{th} argument, conditioned on the fact that a nonempty subset of the infinite relation is chosen. Applying this condition is useful in modulating the parameters of the descriptor for the input argument. The last two rules provide a way to dissect the complex object of an input argument.

For example, consider the output expression for the two predicates in Figure 2. The second argument is defined to be identical to the first for *eq* and to be an element of the set in *member*. In short, an expression is capable of referencing the descriptor of an input argument with the use of @I and then dissecting it (e.g., element, attr) or composing from it (e.g., {...}) to result

[6] Note that the we have not been precise in the rule for tuple, as it needs another recursive rule to represent the arbitrary number of attributes.

Predicate	Bound	Output exp
$eq(X,Y)$	X	@1
$member(X,S)$	X	$element$(@2)

Figure 2: Output for some computed predicates.

in any conceivable descriptor. Once again, note that the computation here is symbolic.

In summary, the descriptor function is enhanced so that it can compute the descriptor for each output argument of the external predicates. These descriptors will, in turn, be used in later cost computations.

5 Conclusions

We have extended $\mathcal{LDL}$ to include modules and external procedures, and have presented a framework for optimizing extended $\mathcal{LDL}$. The notion of well-formedness was introduced to identify the class of programs that could be separately compiled and could be executed without generating infinite relations. We addressed the inherent problems in merging a declarative language such as $\mathcal{LDL}$ with procedural languages. In particular we showed how an external procedure could manipulate complex $\mathcal{LDL}$ objects and handle the multiple solution paradigm of $\mathcal{LDL}$. We provided a set of functions to facilitate the above.

We viewed external procedures as infinite relations, and defined cost descriptors for these relations, called *i-descriptors*, based on the notions of fanout and selectivity. We extended the descriptor and cost functions to operate on *i-descriptors*. The traditional optimization scheme, using the new descriptors and extended functions, proved sufficient for optimizing extended $\mathcal{LDL}$ programs. The results reported here form the basis for the implementation of externals and modules in the $\mathcal{LDL}$ compiler and optimizer developed at MCC.

$\mathcal{LDL}$ contains constructs not addressed in this paper. Updates are of particular interest, since the $\mathcal{LDL}$ optimizer must preserve their relative ordering in a program[NK88]. The optimizer must be made aware of externals that have side-effects, as they must be treated in the same manner as updates. For example, this could be used to guarantee that a graphics device is initialized before anything is actually displayed.

References

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

[CGK89b] Chimenti, D., R. Gamboa, and R. Krishnamurthy. "Execution Models in $\mathcal{LDL}$," MCC Technical Report in preparation.

[CP84] Ceri, S. and G. Pelagatti. *Distributed Databases: Principles & Systems*, McGraw-Hill Book Company, 1984.

[KBZ86] Krishnamurthy, R., H. Boral, and C. Zaniolo. "Optimization of Non-Recursive Queries," in *Proc. of the Conference on Very Large Data Bases (VLDB)*, Kyoto, Japan, 1986.

[KN88a] Krishnamurthy, R., and S.A. Naqvi. "Non-Deterministic Choice in Datalog Programs," in *International Conference on Databases*, Jerusalem, 1988.

[KRS88] Krishnamurthy, R., R. Ramakrishnan, O. Shmueli. "Framework for Testing Safety and Effective Computability of Extended Datalog," in *SIGMOD*, Chicago, 1988.

[KZ88] Krishnamurthy, R., and C. Zaniolo. "Optimization in a Logic Based Language for Knowledge and Data Intensive Applications," *Extending Data Base Technology*, Venice, 1988.

[Ll84] Lloyd, J.W. *Foundations of Logic Programming*, Springer Verlag, 1984.

[MW81] Maier, D., D. S. Warren "Incorporating Computed Relations in Relational Databases", in *Proc. SIGMOD*, Ann Arbor, MI, 1981.

[NK88] Naqvi, S. A., R. Krishnamurthy. "Database Updates in Logic Programming", in *Proc. SIGACT-SIGMOD Principles of Database Systems Conference (PODS)*, Austin, April 1988.

[NT89] Naqvi, S. A., S. Tsur. *A Language for Data and Knowledge Bases*, W.H.Freeman, 1989.

[RBK88] Ramakrishnan, R., C. Beeri, and R. Krishnamurthy. "Optimizing Existential Queries," in *Proc. SIGACT-SIGMOD Principles of Database Systems Conference (PODS)*, Austin, April 1988.

[Sell79] Sellinger, P.G. et al., "Access Path Selection in a Relational Database Management System," in *Proc. SIGMOD Intl. Conf on Mgt. of Data*, 1979.

[Ull85] Ullman, J. "Implementation of Logical Query Languages for Databases," in TODS, Vol. 10, No. 3, pp. 289-321, 1985.