CORAL—Control, Relations and Logic*

Raghu Ramakrishnan Divesh Srivastava S. Sudarshan
Computer Sciences Department,
University of Wisconsin-Madison, WI 53706, U.S.A.

Abstract

CORAL is a modular declarative query language/programming language that supports general Horn clauses with complex terms, set-grouping, aggregation, negation, and relations with tuples that contain (universally quantified) variables. Support for persistent relations is provided by using the EXODUS storage manager. A unique feature of CORAL is that it provides a wide range of evaluation strategies and allows users to — optionally — tailor execution of a program through high-level annotations. A CORAL program is organized as a collection of *modules*, and this structure is used as the basis for expressing control choices. CORAL has an interface to C++, and uses the class structure of C++ to provide extensibility. Finally, CORAL supports a *command sublanguage*, in which statements are evaluated in a user-specified order. The statements can be queries, updates, production-system style rules, or any command that can be typed in at the CORAL system prompt.

*This work was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award, with matching grants from Digital Equipment Corporation, Tandem and Xerox, and NSF grant IRI-9011563. An earlier version of this paper appeared at the NACLP '90 Workshop on Deductive Databases. The authors' e-mail addresses are {raghu,divesh,sudarshan}@cs.wisc.edu.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

1 Introduction

CORAL[1] is a database programming language being developed at the University of Wisconsin–Madison. It seeks to combine features of a database query language, such as efficient treatment of large relations, aggregate operations and declarative semantics, with those of a logic programming language, such as more powerful inference capabilities and support for incomplete and structured data. CORAL significantly extends the expressiveness of standard database query languages such as SQL, and differs from logic programming languages such as Prolog in supporting a modular and non-operational semantics. Applications in which large amounts of data must be extensively analyzed are likely to benefit from this combination of features. In particular, CORAL is attractive for sequence data analysis, natural language processing, temporal queries, and bill-of-materials and other transitive closure related applications.

To provide efficient support for novel applications CORAL allows the user to create new abstract data types and integrate them with the declarative query language in a simple and clean fashion. For instance, when dealing with DNA sequences, a type *sequence* that provides several built-in operations such as approximate subsequence matching, indexing etc. is very useful. Without such an abstract type definition facility, sequences would have to be simulated using, say, lists, which is inefficient.

A CORAL program is a collection of *modules*: at most one *imperative* module and any number of *declarative* and *command* modules. Modules can be compiled separately.

The declarative features of CORAL, which define the sublanguage permitted in declarative modules,

[1]The CORAL project was initiated in 88-89 — under the name Conlog — and an initial overview was presented in [RBSS90].

provide a powerful language that can be used to write complex view definitions or as a general purpose DBPL. CORAL supports tuples with variables (*non-ground terms*). While Prolog systems and some production rule systems support non-ground terms, none of the deductive database systems do so. This facility is useful in itself (for example, it allows the use of non-ground data structures like difference-lists and the use of variables to denote some incomplete information), and is essential for future extensions of the system to deal with "constraint facts," in a manner similar to CLP($\mathcal{R}$) [JMSY90]. The language used for imperative modules is currently C++ extended with several new types and constructs. Predicates defined in other modules can be queried from the imperative module; conversely, predicates defined in the imperative module can be used in other modules. The CORAL system also provides a command sublanguage that serves both as a shell for the CORAL interpreter, and as a set-oriented imperative language.

An important goal of the declarative language is to support efficient execution of queries. While CORAL does optimize queries without aid from the user, it allows the user to control the choice of evaluation techniques in order to improve efficiency of evaluation. In this, CORAL differs from other deductive database and logic programming systems. CORAL also differs from these systems in other ways, which we describe in more detail in Section 8.

1.1 An Example

We present an example in Figure 1 that illustrates declarative modules in CORAL. The program computes paths, with lists used to maintain the sequence of nodes on a path.[2] The example also illustrates the use of multiple modules to structure code — *Listroutines* could contain several other useful operations.

The module definition permits a subset of the defined predicates to be named as exported predicates, and other modules can pose queries over these predicates. The query forms permitted for each exported predicate are also indicated in the `export` declaration. For instance, the *path* module exports the predicate *path* (queries on this predicate that have the last three arguments free and the first argument either bound or free are permitted), and the *Listroutines* module in the above program exports the *append* predicate (in three permissible query forms—each corresponding to one of the three arguments being free and the other two bound). Note that queries that do not match the given form can be posed, but the CORAL system might use an inefficient technique to evaluate such queries, or might run into problems with predicates (such as arithmetic predicates) that require certain arguments to be ground.

[2]The use of *append* is for illustrative purposes. We can use *cons* instead to get the edges in the reverse order.

The current prototype of CORAL is optimized primarily for main-memory execution, but is interfaced with the EXODUS storage manager [CDRS86] to provide transparent (and quite efficient) access to data that is stored on disk. The CORAL system is in the public domain, and a copy of the software can be obtained by contacting the authors.

The rest of this paper is organized as follows. In Section 2, we present the declarative features of CORAL, and in the next section we discuss how the evaluation of a declarative module can (optionally) be guided by a user. In Section 4, we discuss the imperative features, and in Section 5, we present the command sublanguage. We discuss the CORAL system briefly from a user perspective in Section 6, including a preliminary comparison with the LDL system. Support for extensibility in CORAL, including the addition of new data types and operations and new relation and index implementations, is discussed in Section 7. We discuss related systems in Section 8, and outline future directions in Section 9.

2 Declarative Language Features

We assume the usual definitions of terms, literals, rules, etc., and in this section, we discuss more advanced features of the language. In particular, we discuss non-ground facts, and rules with negation and set-generation. CORAL supports a significantly larger class of programs with negation and set-generation than other systems. It is also the *only* deductive database system that supports non-ground facts.

2.1 Non-Ground Facts

Unlike Aditi ([VRK+90]), EKS-V1 ([VBKL90]), LDL ([NT89, CGK+90]), Glue-NAIL! ([MUVG86, PDR91]) and Starburst SQL ([MPR90]), which restrict the facts in a database to be ground, CORAL permits variables within facts. As an example, consider Figure 1. It is

```
module path.
export path(bfff, ffff).
path(X, Y, P1, C1)           :- path(X, Z, P, C), edge(Z, Y, EC),
                                append(P, [edge(Z, Y)], P1), C1 = C + EC.
path(X, Y, [edge(X, Y)], C) :- edge(X, Y, C).
end_module.

module Listroutines.
export append (bbf, bfb, fbb).
append([ ], L, L).
append([H | T], L, [H | L1]) :- append(T, L, L1).
end_module.
```

Figure 1: Program Path

possible to query *append* as follows:

Query: ?-$append([1, 2, 3, 4, X], [Y, Z], ANS)$.

and get the answer (a fact with variables in it)

$$ANS = [1, 2, 3, 4, X, Y, Z].$$

The interpretation of a variable in a fact is that the fact is true for every possible replacement of each variable by a ground term. Thus a fact with a variable in it represents a possibly infinite database. Such facts are often useful in knowledge representation, and could be particularly useful in a database that stores (and possibly manipulates) rules. There is another, possibly more important use of variables—namely to specify constraint facts.

Since we allow non-ground facts, we do not require rules to be range-restricted. Non-ground facts in the database are a special case of non-range-restricted rules where the body is empty.

2.2 Negation

The keyword not is used as a prefix to indicate a negated body literal. For instance, given a predicate *parent*, we can test if a is not a parent of b by using not $parent(a, b)$. Such a literal can be used in a query, or in the body of a rule.

CORAL supports a class of programs with negation that properly contains the class of non-floundering left-to-right modularly stratified programs ([Ros90]). A program is non-floundering if all variables in a negated literal are ground before the literal is evaluated (in the left-to-right rule order). Intuitively, a modularly stratified program is such that in the answers and subgoals generated for the program, there should be no cycles through negation.

The following example from [Ros90] illustrates the use of modularly stratified negation in a program. Suppose we have a complex mechanism constructed out of a number of components that may themselves be constructed from smaller components. Let the component-of relationship be expressed in the relation *part*. A component is known to be working either if it has been (successfully) tested or if it is constructed from smaller components, and all the smaller components are known to be working. This is expressed by the following program.

```
working(X)            :- tested(X).
working(X)            :- part(X, Y),
      not has_suspect_part(X).
has_suspect_part(X) :- part(X, Y), not working(Y).
```

Note that the predicate *working* is defined negatively in terms of itself. However, the *part* relation is acyclic, and hence the *working* status of a component is defined negatively in terms of subcomponents, but not negatively in terms of itself. CORAL provides an evaluation mechanism called Ordered Search [RSS92a] that evaluates programs with left-to-right modularly stratified negation efficiently.

2.3 Creating Sets and Multisets

Sets and multisets are allowed as values in CORAL; $\{1, 2, 3, f(a, b), a\}$ is an example of a set, $\{1, f(a), f(a)\}$ is an example of a multiset. Sets and multisets can contain arbitrary values as elements, and can themselves be used as arguments to functors (lists, sets or multisets). General matching or unification of sets (where one or both of the sets can have variables, respectively) is not supported. Although LDL supports set matching, we believe that most, if not all, uses of set matching as in LDL can be implemented naturally using the suite of functions that we provide on sets. Since we allow arbitrarily nested structures, we must define what the domain (or universe) of discourse is. In this we follow LDL, whose treatment of the universe is an extension of the *Herbrand universe* that is used as a standard in logic programming. The extended Herbrand universe is described in [BNST91].

There are two ways in which sets and multisets can be created using rules, namely, set-enumeration ({ }) and set-grouping (<>) as in LDL. However, the operations CORAL permits on sets are different from those supported by LDL, and we discuss the differences later. The following rule illustrates the use of set-generation:

$$p(X, < Y >) : - \ q(X, Y, Z).$$

This rule uses facts for q to generate a multiset S of instantiations for the variables X, Y, and Z. For each value x for X in this set it creates a fact $p(x, \pi_Y \sigma_{X=x} S)$, where π_Y is a multiset projection (i.e., it does not do duplicate elimination). Thus with facts $q(1,2,3), q(1,2,5)$ and $q(1,3,4)$ we get the fact $p(1, \{2,2,3\})$.

The use of the set-grouping construct in CORAL is similar to (but not exactly the same as) the grouping construct in LDL— set-grouping in CORAL is defined to construct a multiset, whereas it constructs a set in LDL. We can always obtain a set from the multiset using the *set* operator. In fact, with the following rule, the evaluation is optimized to create a set directly, rather than to first create a multiset and then perform duplicate elimination to convert it to a set.

$$p(X, set(< Y >)) : - \ q(X, Y, Z).$$

In several programs, the number of copies of an element is important, and the support for multiset semantics permits simple solutions. For example, to obtain the amount spent on employee salaries, the salary column can be projected out and grouped to generate the multiset of salaries, and then summed up. The projection and grouping in LDL yields a set of salaries, and if several employees have the same salary, the total amount spent on salaries is hard to compute.

We require that the use of the set-grouping operator be left-to-right modularly-stratified (in the same way as negation). This ensures that all derivable q facts with a given value x for X can be computed before a fact $p(x, \_)$ is created. Without such a restriction it is possible to write programs whose semantics is hard to define, or whose evaluation would be inefficient.[3]

2.4 Operations on Sets

We provide several standard operations on sets and multisets as built-in predicates. These include member, union, intersection, difference, multisetunion, cardinality, subset, and set. The multiset versions of these operations are carefully chosen to preserve the intuitive semantics of multisets. For reasons of efficiency, most of these are restricted to testing, and will not permit generation — for example, the subset predicate cannot be used to generate subsets of a set, but can be used to test if a given set is a subset of another. The predicate member is an exception in that it can be used to generate the members of a given set.

The treatment of set-terms in LDL generates a number of rules at compile time that is exponential in the size of the largest set-term in the program text. The use of set-matching is limited in CORAL to avoid this problem. A set term is restricted to be ground (as in LDL) and to match only another (identical) ground set term or variable. All the LDL rules with set terms that we have seen are easily translated into CORAL rules.

We allow several aggregate operations to be used on sets and multisets. The list of aggregate operators we support includes count, min, max, sum, product, average and any. Some of the aggregate operations can be combined directly with the set-generation operations for increased efficiency. For instance, the evaluation of the following rule is optimized to store only the maximum value during the evaluation of the rule, instead of generating a multiset and then selecting the maximum value.

$$maxgrade(Class, \mathsf{max}(< Grade >)) : - \\ student(S, Class), grade(S, Grade).$$

This optimization is also performed for count, min, sum and product.

The program in Figure 2 illustrates how to use aggregation to find shortest paths in a graph with edge weights. (The program as written is not efficient, and

[3] LDL imposes the more stringent restriction that uses of grouping be stratified. We note that while EKS-V1 does not support set-generation through grouping, it does support set-generation in conjunction with aggregate operations such as count, min and sum. Indeed, EKS-V1 allows recursion through uses of aggregation.

```
module shortest_path.
export shortest_path(bfff, ffff).
shortest_path(X, Y, P, C)       :- s_p_length(X, Y, C), path(X, Y, P, C).
s_p_length(X, Y, min(<C>))      :- path(X, Y, P, C).
path(X, Y, P1, C1)              :- path(X, Z, P, C), edge(Z, Y, EC),
                                   append([edge(Z, Y)], P, P1), C1 = C + EC.
path(X, Y, [edge(X, Y)], C)     :- edge(X, Y, C).
end_module.
```

Figure 2: Program Shortest_Path

may loop for ever; in Section 3.2 we describe how annotations may be used to get an efficient version of the program.) This program can be used, for example, to compute cheapest flights. The use of more complicated combinations of grouping and aggregation in CORAL is illustrated below.

$numofemps(M, \mathsf{count}(\mathsf{set}(< E >))) :-$
$\quad worksfor(E, M).$

This results in one tuple per manager with the second argument as the number of distinct employees working under her. The following example illustrates the use of member to generate the elements of a set.

$ok\_team(S) :- old\_team(S), \mathsf{count}(S, C), C \leq 3,$
$\quad \mathsf{member}(X, S), \mathsf{member}(Y, S), \mathsf{member}(Z, S),$
$\quad engineer(X), pilot(Y), doctor(Z).$

Each tuple in *old_team* consists of a set of people. An *ok_team* tuple additionally must contain an engineer, a pilot and a doctor. Note that a team containing a single member who is an engineer, a pilot and a doctor would qualify as an *ok_team*. This program is a translation into CORAL of an LDL program from [STZ92]; the semantics of the original LDL program required that a team contain at most three members. The addition of $\mathsf{count}(S, C), C \leq 3$ to the body of the rule ensures this.

2.5 Persistent Relations

The schema of a persistent relation must be declared, e.g., $schema(employee(string, int, float, string))$. Currently, tuples in a persistent relation are restricted to have fields of type string, int or float. Except for the points noted below (in Section 2.6), a persistent relation behaves just the same as a non-persistent relation. Indices can be declared, and are implemented as B+ tree indices.

CORAL uses the EXODUS storage manager to support persistent relations. EXODUS uses a client-server architecture; CORAL is the client process, and maintains buffers for persistent relations. If a requested tuple is not in a local buffer, a request is forwarded to the EXODUS server and the page with the requested tuple is retrieved. In the current implementation, the tuple is copied from the local buffer into the CORAL space. This is adequate when queries do not examine very large subsets of persistent relations, but is likely to cause problems otherwise. We are investigating alternative techniques wherein local copies of requested tuples are not created.

2.6 Multiple Databases

A *database* is a collection of relations, which can be either explicitly enumerated "base" relations or relations exported by a module. It is useful to think of a database as a workspace or environment. A user can have several named databases, copy relations between two databases (or simply make a relation in one database visible from another without copying), update relations in a database, or run queries against a database. It is also possible to save a database in a file between executions.

Persistent relations exist in a database called "db_rels", and can be made visible to other databases without copying. When a database that refers to a persistent relation is saved, only the name of the persistent relation—and not its current set of tuples—is saved.

3 Controlling the Evaluation of Declarative Modules

For (pure) declarative modules, CORAL evaluation (with occur checks) is guaranteed to be sound, i.e., if the system returns a fact as an answer to a query, that fact indeed follows from the semantics of the declarative program. The evaluation is also "complete" in a limited sense — as long as the execution terminates, all answers to a query are actually generated. It is possible however, to write queries that do not terminate; in some such cases (e.g., programs without negation or set-grouping) CORAL is still complete in that it enumerates all answers in the limit. (Of course, the use of choice, updates, aggregate selections and the absence of occur checks can result in incomplete or even unsound evaluation. These features should therefore be used with some care.)

During the evaluation of a rule r in module M, if we generate a query on a predicate exported by module N, a call is set up on module N. The answers to this query are used iteratively in rule r; each time a new answer to the query is required, rule r requests for a new tuple from the interface to module N. The interface to relations exported by a module makes no assumptions about the evaluation of the module. Module N may contain only base predicates, or may have rules that are evaluated in any of several different ways. The module may choose to cache answers between calls, or choose to recompute answers. All this is transparent to the calling module. Similarly, the evaluation of the called module N makes no assumptions about the evaluation of calling module M.

This orthogonality permits the free mixing of different evaluation techniques in different modules in CORAL and is central to how different executions in different modules are combined cleanly.

CORAL provides several rewriting transformations such as Magic Templates, Supplementary Magic Templates, Context Factoring, Existential Query Rewriting, etc. (see, e.g., [RSS92b]). By default, CORAL chooses a combination of rewriting transformations. However, other combinations might work better for some queries, and the expert user can choose an appropriate combination using annotations. The user can also control the execution in ways other than the choice of a rewriting strategy; we describe these options in the following sections.

3.1 Module Level Control

Materialization Vs. Pipelining:

Consider the following rules:

$$r(X,Y) :- p(X,Z), q(Z,Y).$$
$$p(X,Y) :- p1(X,Z), p2(Z,Y).$$

Materialized evaluation creates a relation for p and stores the generated tuples, whereas pipelined evaluation simply generates the p tuples and joins them with q tuples. The two approaches complement each other. If p is used many times, the cost of materialization is outweighed by the savings in avoiding recomputation. On the other hand, pipelining can be done very efficiently, and unless subqueries on p are indeed set up multiple times, the cost of storing the p tuples is avoided.

CORAL supports both materialization and pipelining. An interesting aspect of pipelining in CORAL is the treatment of recursive predicates. A subquery on the recursive predicate is solved by a recursive invocation of the same module, and each invocation pipelines the local results. The resulting computation is close to the evaluation strategy of a top-down implementation such as Prolog. (Of course, pipelined evaluation of recursive modules carries the same risks of potential incompleteness, and should be used with care.)

Controlling the Order of Deductions:

The use of facts computed during bottom-up evaluation can be prioritized. Consider the shortest path program from Figure 2, that uses the predicate $path(Source, Destination, Path\_list, Cost)$. For this program, it is better to explore paths of lesser cost first. This can be achieved by using *path* facts of lesser cost in preference to *path* facts of greater cost. *path* facts of greater cost are hidden when they are derived, and each time a fixpoint is reached, the *path* facts of lowest cost are exposed. This continues until there are no more hidden facts.

The user can specify that the evaluation prioritize the use of facts in this fashion, using an annotation of the following form:

$$@\ \mathsf{prioritize}\ path(X,Y,P,C)\ min(C).$$

This annotation is easily extended to prioritize facts for multiple predicates at the same time. We describe the benefits of this annotation in Section 3.2.

The Save Module Facility:

The module facility provides several important advantages. First, by moving many rules out of a module, the number of rules that are involved when performing an iteration on a module is reduced; this is particularly useful when computation in the higher module can proceed only after answers to subgoals on the lower module have been returned. Second, predicates defined in an external module are treated just like base predicates by the semi-naive rewriting algorithms — whenever there is a query (or set of queries) on such a predicate, a call to the module is made, and all the answers are evaluated. This has the benefit that the number of semi-naive rewritten rules decreases considerably if more predicates can be treated as base predicates. Third, in most cases, facts (other than answers to the query) computed during the evaluation of a module are best discarded to save space (since bottom-up evaluation stores many facts, space is generally at a premium). Module calls provide a convenient unit for discarding intermediate answers. By default, CORAL does precisely this - it discards all intermediate facts and subgoals computed by a module at the end of a call to the module.

However, there are some cases where the first two benefits of modules are desired, but the third feature is not a benefit at all, but instead leads to a significant amount of recomputation. This is especially so in cases where the same subgoal in a module is generated in many different invocations of the module. In such cases, the user can tell the CORAL system to maintain the state of the module (i.e., retain generated facts) in between calls to the module, and thereby avoid recomputation; we call this facility the *save module* facility.

In the interest of efficient implementation we have the following restriction on the use of the save module feature: *if a module uses the save module feature, it should not be invoked recursively.* We do not make any guarantees about correct evaluation should this happen at run-time. (Note that the predicates defined in the module can be recursive; this does not cause recursive invocations of the module).

3.2 Predicate Level Control

CORAL provides a variety of annotations at the level of predicates. By default, duplicate elimination is performed when inserting facts into a relation, so that a relation with only ground tuples consists of a set of facts.[4] An annotation allow_duplicates tells the system to not perform duplicate checks for any predicate in the module. This can also be done on a per-predicate basis. If a predicate in a program is declared to be multiset, CORAL guarantees that the number of copies of tuples in the answer to a goal on the predicate is equal to the number of derivations for the tuple in the original program. Some of the other predicate-level annotations are described below.

Indexing Relations:

CORAL supports two forms of indices: (1) *argument form indices*, and (2) *pattern form indices*. The first form creates an index on a subset of the arguments of a relation. The second form is more sophisticated, and allows us to retrieve precisely those facts that match a specified pattern that can contain variables. Such indices are of great use when dealing with complex objects created using functors. Suppose a relation *employee* had two arguments, the first a name and the second a complex term $address(Street, City)$. The following declaration then creates a pattern form index that can efficiently retrieve, for instance, employees named "John", who stay in "Madison", without knowing their street.

$$@\ \mathsf{make\_index}\ employee(Name, address(Street, City))(Name, City).$$

The Magic Templates rewriting stage generates annotations to create all indices that are needed for efficient evaluation. The user is allowed to specify additional indices, which is particularly useful if the Magic Templates rewriting stage is bypassed.

The Choice Operator:

CORAL provides a version of the choice operator of LDL, but with altogether different semantics [RBSS90]. The following example illustrates the use of choice in CORAL. Suppose with the *path* predicate from Figure 2, we are interested in just one path between each pair of nodes. (For instance, the user may want just one answer, or perhaps the predicate *path* is used in a computation that works equally well, irrespective of which path it gets, so long as it gets at least one path between each pair of nodes X, Y whenever such a path exists.) This can be specified using the following annotation:

$$@\mathsf{choice}\quad path(X, Y, P, C)(X, Y)(P, C).$$

The choice annotation says that for each value of the pair x, y, at most one fact $path(x, y, p, c)$ need be retained for *path*. If more than one fact $path(x, y, p, c)$ is generated by the program for any pair x, y, the system arbitrarily picks one of the facts to retain, and discards the rest. If we wish to retain a path for each pair of nodes and each path cost, we could use an annotation

$$@\mathsf{choice}\quad path(X, Y, P, C)(X, Y, C)(P).$$

[4] If facts contain variables, duplicate elimination requires subsumption checking. CORAL does not, however, guarantee that relations are maintained as irredundant sets of facts.

Unlike in LDL, the choice made is final — CORAL does not backtrack and try different ways to make the choice. We believe this semantics can be implemented more efficiently in a bottom-up evaluation than the LDL semantics. Giannotti et al. [GPSZ91] have investigated the connections between this "local" version of choice and stable models, and Greco et al. [GZG92] have shown that it is useful in a variety of "greedy" algorithms.

Aggregate Selections:

Consider the *shortest_path* program from Figure 2. To compute shortest paths between points, it suffices to use only the shortest path between pairs of points - path facts that are do not correspond to shortest paths are irrelevant. CORAL permits the user to specify an *aggregate selection* of the following form on the predicate *path*.

$$@\ \mathsf{aggregate\_selection}\ path(X,Y,P,C)(X,Y)min(C).$$

The system then checks (at run-time) if a path fact is such that there is a path fact of lesser cost C with the same value for X, Y (i.e., between the same pair of points), and if there is such a fact, the costlier path fact is discarded. This aggregate selection is extremely important for efficiency — without it the program may run for ever, generating cyclic paths of increasing length. With this aggregate selection, along with the choice annotation @choice $path(X,Y,P,C)(X,Y,C)(P)$, a single source query on the program runs in time $O(E \cdot V)$, where there are E edge facts, and V nodes in the graph.

Using facts in a prioritized fashion (described in Section 3.1) reduces the cost of evaluation of a single source shortest path problem from a worst case of $O(E \cdot V)$ to $O(E \cdot log(V))$[5] ([SR91]). This illustrates the importance of aggregate selections and prioritizing the use of facts in a bottom-up evaluation. [SR91] describes a technique to generate such aggregate selections automatically, but aggregate selections could also be specified by the user.

3.3 Rule Level Control

Join Orders:

CORAL uses a default left to right join order in the absence of information about relation sizes, except that for semi-naive rewritten rules the "delta" predicate is moved to the head of the join order. The user can override this default by specifying the join order on a per-rule or on a per-semi-naive-rewritten rule basis.

The Update Operator:

We allow a limited form of update in rule heads, which is illustrated by the following program.

$R1: path(X,Y,\infty).$
$R2: path(X,Y,\infty \rightarrow C) \quad :- edge(X,Y,C),$
$\quad path(X,Y,\infty).$
$R3: path(X,Y,C \rightarrow C1+C2) :- edge(X,Z,C1),$
$\quad path(Z,Y,C2), path(X,Y,C), C1+C2 < C.$
Query: ?-$path(a,b,C)$.

In the above program we have used ∞ to represent some value that is larger than the maximum length of acyclic paths in the *edge* relation. Note the difference in the structure of the head of rule $R3$. The notation $C \rightarrow C1 + C2$ says that on successfully instantiating the rule, any fact that matches $path(X,Y,C)$ should be replaced by a fact $path(X,Y,C1+C2)$. Under the semantics described above, an evaluation of this program stores only one fact $path(x,y,c)$ for each pair x, y, and computes shortest paths in the *edge* graph.

An alternate way of understanding this rule would be as follows.

$path(X,Y,C1+C2), \mathsf{delete}\ path(X,Y,C) :-$
$\quad edge(X,Z,C1), path(Z,Y,C2),$
$\quad path(X,Y,C), C1+C2 < C.$

Each successful rule instantiation deletes any facts that match the (instantiated version of) $path(X,Y,C)$, and inserts (the instantiated version of) $path(X,Y,C1+C2)$. In effect, here, the third field of the *path* fact is updated "in place."

The semantics of using this operation is, in the general case, non-deterministic. However, there are useful classes of programs when this is deterministic.

4 Imperative Modules

An imperative module is a program in an imperative language, which consists of C++ augmented by adding a layer of new types and constructs.

The basic object the C++ user needs to understand to be able to interface with CORAL is the *relation*, which can be treated as a set of tuples. Indices can be added to a relation by means of a procedure call. A C++ user can also directly access a database relation (not just get a copy of it) by providing the name of the relation and its arity. CORAL provides facilities to insert tuples into, and delete tuples from relations

[5] Assuming that the edge cost are non-negative.

using the "+ =" and "− =" operators. A procedure update_tuple to update tuples in a relation is also provided.

CORAL provides two iterative constructs for accessing the tuples of a relation (the tuples are returned in an arbitrary order). FOR_EACH_TUPLE successively instantiates its first argument to each tuple in the relation given by the second argument. FOR_EACH_MATCHING_TUPLE successively instantiates its first argument to each tuple in the relation given by the second argument that matches the pattern specified by the third argument. A variety of other functions are available to the imperative language programmer to manipulate relations. These include all the set and aggregate functions described earlier.

A C++ user can invoke a query on a relation that is defined declaratively (and exported by a declarative module), using a procedure call_coral. There are two variants of this procedure, one of which takes a single query, and the other a set of queries. In later versions of CORAL we plan to allow inline declaration of a declarative module within imperative code. This will provide a simpler syntax for calling declarative modules and could be interpreted as a direct extension of the C++ language. CORAL provides a simple convention for defining predicates using C++ code. We also provide a facility to define subclasses of tuple with typed and named attributes. Methods corresponding to the attribute names are created for the subclass, and these help in seamlessly converting types between C++ primitive types and CORAL's internal type system.

5 Command Modules

Command modules provide support for imperative programming without resorting to C++. Command modules provide sequencing and iteration as control constructs, and a set of atomic commands that includes any command that can be typed in at the CORAL prompt. Command modules can be parameterized. For example, if we wish to write a module that reads in a file and does some processing, the name of the file can be a parameter. Whereas — at least, in the absence of dynamic linking — C++ imperative modules must be compiled with the CORAL runtime system, command modules can be consulted from the CORAL prompt, just like declarative modules. We illustrate some features of command modules using the following example to sort a unary relation.

Example 5.1 If the relation *elem* is implemented as a heap (this can be specified by the CORAL user), the following program[6] would implement heap sort. This can be used elsewhere through the notation $HeapSort(in\_rel).sort(Sorted\_list)$. Note that in_rel could be a set or the name of a relation.

module HeapSort (in_rel).
export $sort$ (f).
$elem(X) :- in\_rel(X)$.
$sort([\,])$.
while ($elem(C)$) {
  $sort(B \rightarrow [A|B]) \ :- sort(B), A = max(elem.1)$.
  delete $elem(A) \ :- sort([A|B])$.
}
end_module.

$max(elem.1)$ is used to obtain the maximum element in the first column of the (current) *elem* relation. Consequently, the above program incrementally builds up sorted lists of the largest values in the *elem* relation. □

6 Using the CORAL System

The CORAL implementation, in contrast to LDL, does not do full compilation. Rather, there is a fixed runtime system that essentially interprets rules, and a user program is compiled into an optimized set of rules. This results in fast compilation, making CORAL suitable for interactive program development.[7] CORAL provides utilities to take a text file organized as a table, parse it into fields and records, and convert it into a relation. Similarly, utilities for output of relations in tabular form are also provided. The user can also execute any Unix command using the shell command from the CORAL prompt. CORAL provides a help facility that details the various commands available from the CORAL prompt.

Program Development Environment:

CORAL provides some basic facilities for debugging of programs. A trace facility is provided that does the

[6] The program is intended to illustrate the functionality of command modules, but some of the syntax is still tentative.

[7] However, if there is an imperative module in a program, the entire CORAL system must be re-compiled when the imperative module is compiled. This is because we do not use a dynamic linker.

following: (1) it lets the user know what rules are being evaluated, and (2) it prints out answers and subgoals as they are generated, to let the user know how the computation is proceeding. It is possible to trace individual predicates rather than tracing all predicates.

CORAL also provides some high-level profiling facilities. The unit of profiling is the unification operation. Unification of two atomic terms counts as one unification, while, for example, unification of $f(X, Y)$ and $f(a, b)$ counts as three unifications, one at the outer level and two at the inner level. Profiling also lets the user know how efficient the indexing is, by keeping counts of the number of tuples that the indexing operation tried to unify, and the number that actually unified and were retrieved. In addition, other counts such as number of successful applications of each rule, and the number of unsuccessful attempts at using a rule are also maintained. All this information put together gives users a fair idea of where their programs are spending the most time, and helps them optimize programs accordingly.

6.1 A Preliminary Performance Comparison of CORAL with LDL

In this section we present results of a brief comparison of the performance of CORAL with LDL. (EKS-V1 and Glue-NAIL! are not currently distributed publicly, and we were not able to compare their performance.) We emphasize that these results are preliminary, since CORAL is still in the process of being tuned, and our comparison is itself very limited. The goal is to give the reader some idea of how CORAL compares with other deductive database systems.

A main observation is that by virtue of being partially interpreted instead of fully compiled, CORAL is much faster than LDL in reading and compiling queries. In this respect, CORAL is comparable to Prolog systems. It is therefore very convenient for interactive program development.

We also compared execution times — Unix user cpu times on a lightly loaded Sun 4 workstation — on a work load that included a simple join rule, linear recursive programs (ancestor and same generation), non-linear programs (bi-linear ancestor), and structure manipulation (list append).[8] The data-sets included trees, chains, a (sparse) random graph and lists of varying lengths. Both LDL and CORAL allow the user some execution choices (e.g. whether or not to eliminate duplicates), and we used the best combinations for both systems.[9]

To summarize the results of the comparison, we found that the following observations generally held. LDL was about three times faster than CORAL on simple joins (10s vs. 30s on a join that generated — but did not materialize – an intermediate relation of 100,000 tuples). This is explained by the fact that LDL's compilation strategy allows for some optimizations on a per-rule basis that is not possible with CORAL's partial interpretation, and also by the fact that CORAL uses more abstract representations for terms since it has to support non-ground terms, unlike LDL. CORAL was typically much faster than LDL on the linear recursive queries. A selection of the numbers that we obtained illustrates this: on same generation with a 0.1% selection, 5.2s vs. 25.3s on a chain of length 1000 and 5.2s vs. 28.9s on a tree; and on right-linear ancestor with no selection, 37.4s vs. 265.8s on a chain of length 160. (The only exception that we found to this trend was on left-linear ancestor with no selections, where CORAL took 33.5s vs. 17.1s for LDL on a chain of length 160.) While there is no fundamental reason why this should be so, we conjecture that CORAL perhaps has better indexing. CORAL was also faster on bilinear queries, especially with selections. For instance, on bi-linear ancestor with a 1% selection, CORAL took 1.2s vs. 39.8s for LDL on a tree. The reason here is that LDL only implements a version of magic sets that deals with linear recursive queries. This is confirmed by the fact that LDL cannot run the takeuchi program (a standard Prolog benchmark), where bindings must be propagated through recursive literals to avoid unsafe calls on arithmetic predicates. Finally, CORAL is linear on append, whereas LDL is quadratic. For example, CORAL execution time went from 0.6s to 2.4s when we quadrupled the list length, while LDL went from 4s to 56s. Again, we conjecture that the difference is due to indexing, and for this example in particular, the treatment of structured terms.

We also ran these queries on a Prolog system, CLP($\mathcal{R}$) Version 1.1 from IBM. As expected, CORAL

[8] The programs and queries were chosen to preclude intelligent backtracking, factoring and other optimizations that apply to some, but not all, programs in order to get numbers that reflect the general case.

[9] For CORAL, pipelined execution is over twice as fast on append, but we give the numbers for bottom-up evaluation to provide a meaningful comparison.

was much faster on all but the append query, on which CLP($\mathcal{R}$) was much faster. On the append program, the overhead of memoing facts was wasted; on the other programs, with the exception of the join, it saved much repeated computation. (Incidentally, Prolog will not terminate on the left-linear and bi-linear versions of ancestor.) We note that there are Prolog systems such as BIM, Quintus and Sicstus Prolog that are much faster than CLP($\mathcal{R}$); we used CLP($\mathcal{R}$) since the others are not available to us currently.

Finally, we note that Prolog-style execution can be obtained in CORAL by using the pipelined mode of evaluation. Our implementation of pipelining is not as sophisticated as current Prolog implementations; execution is typically slower than CLP($\mathcal{R}$), but by a factor of less than 10. However, on many programs, e.g. append, it is faster than standard bottom-up evaluation with magic sets rewriting.

7 Extensibility in CORAL

The implementation of the declarative language of CORAL is designed to be extensible, i.e., the user can add new types to the system, and can add new implementations of relations and indices, without modifying or recompiling the rest of the system code. The user's program will, of course, have to be compiled and linked with the system code.

7.1 Adding Abstract Types to CORAL

To create a new type[10], the user must declare it as a subclass of the system class ConstArg. Several virtual functions (methods) must be defined for the new type. These include: an operator '==' (which takes an object of type Arg as parameter), printon (which takes a file as a parameter), hash which returns a hash value, copy which creates a copy of the object, and delete which is called when the system no longer needs the object.

For example, once we define a new type, say bitmap, we can create tuples with arguments that are bitmaps — employee records can now store photos of employees represented as a bitmap. The function printon controls how bitmaps are interpreted when they are printed. It can, for instance, create a window to display the bitmap.

[10] Objects of the user-defined type must be "constants", i.e., they cannot contain variables within them. New types that are not constants can be supported with minimal change to the system code. Later versions of CORAL may provide more direct support for such types, without the need to modify system code.

CORAL does not provide syntactic support for objects of new types within the declarative module. However, it is possible for the user to define built-in predicates that construct or retrieve subparts of objects of the new type. For example, a constructor for the data type *sequence* may take as parameter a list of elements, and convert it into whatever internal format is used for sequences. Clearly, bitmaps cannot be efficiently constructed thus, and must be created by imperative code written by the user. They can then be stored in relations, and manipulated just like other CORAL types.

Once objects of a user-defined type are created, presumably the user will want to manipulate them using rules. Builtin predicates on the user-defined type will probably be critical for this stage. CORAL provides the user with a very simple way of creating such builtin predicates, and hence this stage should not be a bottleneck in developing applications that use user-defined types.

The user has control over both the copy routine and the delete routine. The system never modifies a constant object, so the copy function can merely return a pointer to the old copy of the object, so long as the delete function is written keeping this in mind (perhaps using a reference count scheme). Such sharing is important for a type such as a bitmap that could use a lot of space.

7.2 Adding New Relation and Index Implementations

CORAL currently supports relations organized as linked lists, relations organized as hash tables, relations defined by rules, and relations defined by C++ functions. The interface code to relations makes no assumptions about the structure of relations, and is designed to make the task of adding new relation implementations easy.

Tuples in a relation can be accessed using the $get\_next\_tuple(TupleIterator)$ member function of the type $Relation$. This function takes as a parameter a $TupleIterator$ structure that contains a pattern, and each call to this function returns a tuple (from the relation) that matches the pattern. The code that searches the relation can save its state in a field of the $TupleIterator$ in between calls to get_next_tuple. Tuples can be inserted into relations using $insert(Tuple)$, and tuples can be deleted from relations using *delete*(*Tuple*), each of which are member functions of the type *Relation*, New implementations of relations can be created by making the implementation a subclass of *Relation*. The functions *insert*, *delete* and *get_next_tuple* are virtual functions, and can be redefined for the user-defined implementation of the relation.

Similarly, the user can create index structures as subclasses of type *Index*, along with *insert*(*Tuple*), *get_next_tuple*(*TupleIterator*) and *delete*(*Tuple*) function definitions. The user can store the indices in an *IndexSet* field of the relation, and can use the indices to make the *get_next_tuple* function on relations efficient. It is relatively straightforward to add, for instance, a B-tree index in this fashion.

8 Related Systems

There are many similarities between CORAL and deductive database systems such as Aditi ([VRK+90]), EKS-V1 ([VBKL90]), LDL ([NT89, CGK+90]), Glue-NAIL! ([MUVG86, PDR91]) and Starburst SQL ([MPR90]). However, there are several important differences, and CORAL extends all the above systems in the following ways:

1. CORAL supports a larger class of programs, including programs with non-ground facts and non-stratified negation and set-generation.
2. CORAL supports a wide range of evaluation techniques, and gives the user considerable control over the choice of techniques.
3. CORAL is extensible — new data and relation types and index implementations can be added without modifying the rest of the system.

With respect to EKS-V1, we note that it is the only system that supports integrity constraint checking. It also supports hypothetical reasoning. Aditi is unique in giving primary importance to disk-resident data.

LDL++, a successor to LDL under development at MCC Austin, is reportedly also moving in the direction taken by CORAL in many respects. It will be partially interpreted, support abstract data types, and use a local semantics for choice (Carlo Zaniolo, personal communication).

In comparison to logic programming systems, such as various implementations of Prolog, CORAL provides better indexing facilities and support for persistent data. Most importantly, the declarative intended model semantics is supported (for all positive Horn clause programs, and a large class of programs with negation and aggregation as well).

Modules serve as the units of compilation, and several evaluation choices can be specified on a per-module basis. Unlike Glue-NAIL! and LDL, where modules have only a compile-time meaning and no run-time meaning, modules in CORAL have important run-time semantics. Several run-time optimizations are done at the module level. For instance, modules provide a very useful unit for discarding intermediate facts—this is important with bottom-up computation, since facts that are computed are generally not discarded anywhere else, and would use excessive amounts of memory. Modules with run-time semantics are also available in several production rule systems (for example, RDL1 [KdMS90]).

9 Future Directions

A number of issues require further work. These include support for metaprogramming, constraints, disk-resident data, new data types and operations, user interfaces, inheritance and object orientation.

10 Acknowledgements

We would like to acknowledge our debt to LDL, NAIL!, SQL, Starburst, and various implementations of Prolog from which we have borrowed numerous ideas. We would like also to acknowledge the contributions of Per Bothner, who played a principal role in the implementation of the first prototype of CORAL, and Praveen Seshadri, who contributed significantly to the implementation of CORAL.

References

[BNST91] Catriel Beeri, Shamim Naqvi, Oded Shmueli, and Shalom Tsur. Set constructors in a logic database language. *The Journal of Logic Programming*, pages 181–232, 1991.

[CDRS86] Michael Carey, David DeWitt, Joel Richardson, and Eugene Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the International Conference on Very Large Databases*, August 1986.

[CGK+90] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):76–90, 1990.

[GPSZ91] Fosca Giannotti, Dino Pedreschi, Domenico Sacca, and Carlo Zaniolo. Non-determinism in deductive databases. In C. Delobel, M. Kifer, and Y. Masunaga, editors, *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases DOOD'91*, Munich, Germany, 1991. Springer-Verlag.

[GZG92] Sergio Greco, Carlo Zaniolo, and Sumit Ganguly. Greedy by choice. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1992.

[JMSY90] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. Technical report, IBM, T. J. Watson Research Center, 1990.

[KdMS90] G. Kiernan, C. de Maindreville, and E. Simon. Making deductive database a practical technology: a step forward. In *Proceedings of the ACM SIGMOD Conf. on Management of Data*, 1990.

[MPR90] Inderpal S. Mumick, Hamid Pirahesh, and Raghu Ramakrishnan. Duplicates and aggregates in deductive databases. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, August 1990.

[MUVG86] Katherine Morris, Jeffrey D. Ullman, and Allen Van Gelder. Design overview of the NAIL! system. In *Proceedings of the Third International Conference on Logic Programming*, 1986.

[NT89] Shamim Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases.* Principles of Computer Science. Computer Science Press, New York, 1989.

[PDR91] Geoffrey Phipps, Marcia A. Derr, and Kenneth A. Ross. Glue-NAIL!: A deductive database system. In *Proceedings of the ACM SIGMOD Conf. on Management of Data*, pages 308–317, 1991.

[RBSS90] Raghu Ramakrishnan, Per Bothner, Divesh Srivastava, and S. Sudarshan. CORAL: A database programming language. In Jan Chomicki, editor, *Proceedings of the NACLP '90 Workshop on Deductive Databases*, October 1990. Available as Report TR-CS-90-14, Department of Computing and Information Sciences, Kansas State University.

[Ros90] Kenneth Ross. Modular Stratification and Magic Sets for DATALOG programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 161–171, 1990.

[RSS92a] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. Manuscript, submitted for publication, 1992.

[RSS92b] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Efficient bottom-up evaluation of logic programs. In J. Vandewalle, editor, *The State of the Art in Computer Systems and Software Engineering.* Kluwer Academic Publishers, 1992.

[SR91] S. Sudarshan and Raghu Ramakrishnan. Aggregation and relevance in deductive databases. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, September 1991.

[STZ92] Oded Shmueli, Shalom Tsur, and Carlo Zaniolo. Compilation of set terms in the logic data language (LDL). *Journal of Logic Programming*, 12(1&2):89–120, 1992.

[VBKL90] L. Vieille, P. Bayer, V. Küchenhoff, and A. Lefebvre. EKS-V1, a short overview. In *AAAI-90 Workshop on Knowledge Base Management Systems*, 1990.

[VRK+90] Jayen Vaghani, Kotagiri Ramamohanarao, David Kemp, Zoltan Somogyi, and Peter Stuckey. The Aditi deductive database system. In *Proceedings of the NACLP'90 Workshop on Deductive Database Systems*, 1990.