The CORAL Deductive System

Raghu Ramakrishnan
University of Wisconsin, Madison

Divesh Srivastava
AT&T Bell Laboratories, Murray Hill

S. Sudarshan
AT&T Bell Laboratories, Murray Hill

Praveen Seshadri
University of Wisconsin, Madison

Abstract

CORAL is a deductive system which supports a rich declarative language, and an interface to C++ which allows for a combination of declarative and imperative programming. The declarative query language supports general Horn clauses augmented with complex terms, set-grouping, aggregation, negation, and relations with tuples that contain (universally quantified) variables. A CORAL declarative program can be organized as a collection of interacting modules. The CORAL implementation supports a wide range of evaluation strategies, and automatically chooses an efficient evaluation strategy for each module in the program. In addition, users are permitted to guide query optimization, if desired, by selecting from among a wide range of control choices at the level of each module.

The CORAL system provides imperative constructs such as update, insert and delete rules. CORAL also has an interface with C++, and users can program in a combination of declarative CORAL and C++ extended with CORAL primitives. A high degree of extensibility is provided by allowing C++ programmers to use the class structure of C++ to enhance the CORAL implementation. CORAL provides support for both main-memory and disk-resident data. Disk-resident data is supported using the EXODUS storage manager, which also provides transaction management in a client-server environment.

The paper presents a comprehensive view of the system from broad design goals, the language, and the architecture of the system down to language interfaces and implementation details.

1 Introduction

We discuss the important aspects of the design and implementation of the CORAL deductive system. The goal was to combine the set-oriented data manipulation characteristics of relational systems with a powerful declarative programming language.

The CORAL deductive system[1] provides a powerful declarative language that can be used to express complex queries or view definitions on databases. CORAL combines features of database query languages, such as efficient treatment of large relations, aggregate operations and declarative semantics, with features of a logic programming language, such as powerful inference capabilities and support for structured and incomplete data. The CORAL declarative language significantly extends the expressiveness of standard database query languages such as SQL, and differs from logic programming languages such as Prolog in supporting a declarative semantics.

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 DEC, Tandem and Xerox, and NSF grant IRI-9011563.

The work of Divesh Srivastava and S. Sudarshan was performed while they were at the University of Wisconsin, Madison.

The addresses of the authors are Computer Sciences Department, University of Wisconsin, Madison, WI 53706, USA, and AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA. The authors email addresses are {raghu,praveen}@cs.wisc.edu and {divesh,sudarsha}@research.att.com.

Parts of this paper appeared in the Proceedings of the International Conference on Very Large Databases, 1992 ([RSS92b]), and in the Proceedings of the ACM SIGMOD International Conference on Management of Data, 1993 ([RSSS93b]).

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

Applications in which large amounts of data must be analyzed, and the analysis is too complex to be performed using a less expressive language such as SQL, are likely to benefit from the combination of features provided by CORAL. Examples of such applications include sequence queries, such as stock market analysis queries and DNA sequence analysis queries, and generalized transitive closure queries, such as bill-of-materials queries. We discuss several applications of CORAL in the paper.

Queries written in a declarative language do not specify how they should be evaluated. Since the database on which the queries are evaluated may be quite large, efficient execution of declarative queries is an important requirement of any deductive database system. We believe that no one evaluation technique is the best in all situations. Hence CORAL supports a wide range of evaluation strategies and optimization techniques ([Ram88, NRSU89, RSS94, KRS90, RS91, RSS92a], for instance). CORAL automatically chooses an efficient evaluation strategy for each program but, given the rich set of constructs in the language, we believe that some user guidance is critical to effectively optimizing many sophisticated programs. Several optimization techniques may be simultaneously applicable to the same program. Further, different optimization and evaluation techniques may be ideal for different parts of the program. A challenge faced by CORAL was how to effectively combine different optimization and evaluation techniques, and to provide users with the ability to choose, in a relatively orthogonal manner, from the suite of optimizations supported by CORAL. The module structure, described below, is the key to meeting this challenge.

A CORAL program is a collection of modules, each of which can be separately compiled (into CORAL internal data structures). Modules are the units of optimization and also the units of evaluation. Evaluation techniques can be chosen on a per-module basis, and different modules with different evaluation techniques can interact in a transparent fashion; the evaluation of each module is independent of the techniques used to evaluate other modules. In addition, the user can optionally specify high-level annotations at the level of each module, to guide query optimization. The ability to mix and match different evaluation strategies and optimization techniques in different modules greatly enhances the practical utility of the system, and reflects well upon the modularity of the system. CORAL is more flexible than other deductive database and logic programming systems in this respect. We consider the annotation-based approach to control, and some of the novel annotations supported, to be important contributions of the CORAL project.

While declarative languages can provide considerable ease of expression, users may want to code parts of their applications in an imperative language for reasons of efficiency, or for performing inherently imperative actions such as updates or user interaction. Hence, an important goal of the CORAL effort was to integrate the deductive system with a general purpose programming language, with minimal impedance mismatch. Since CORAL is implemented in C++ [Str91], this is the language with which CORAL has been interfaced. Users can program in a combination of declarative CORAL and C++, and the interface is bi-directional: (1) CORAL code can be embedded within C++ code and, further, data in the database can be manipulated from C++ directly, using high-level abstractions provided by the CORAL interface. (2) Declarative CORAL code can use predicates defined using C++ procedures.

To provide efficient support for novel applications, CORAL allows the user to create new C++ classes, and manipulate objects of these classes in the declarative query language. New implementations of relations and indices can also be added easily. Thus the CORAL deductive system is *extensible*. Extensibility has proven very useful in several CORAL applications.

The CORAL system utilizes the EXODUS client-server storage manager [CDRS86] to provide support for disk-resident relations; however, it can run in a stand-alone mode, without EXODUS, if all data is in main memory. The CORAL architecture thus supports an environment where one or more clients use copies of the CORAL system to execute queries, accessing data from a shared EXODUS server. Since the client-server interaction (including concurrency control, buffer management and recovery) is largely handled by EXODUS, much of the design effort has focussed on the architecture of the system at each client.

The CORAL deductive system is available from the University of Wisconsin along with an extensive user manual and a large suite of sample programs.[2] The actual implementation includes all the features described in

[2]CORAL is free software available by anonymous ftp from ftp.cs.wisc.edu.

this paper, unless otherwise stated.

The rest of this paper is structured as follows. In Section 2, we present the declarative features of CORAL; this is the primary focus of the system. We briefly discuss the interactive environment of the CORAL system in Section 3, touching upon some interesting features like data organization capabilities, transaction facilities, and debugging tools. In Sections 4–7, we discuss various aspects of the system implementation, from the CORAL system architecture, down to the underlying representation of data in the system. Section 4 contains an overview of the CORAL system architecture. Section 5 provides an overview of query evaluation and optimization. Section 6 covers the basic strategies used in evaluating a module, as well as several important refinements. This section also addresses user guidance of query optimization via annotations. Section 7 explains the underlying representation of data used in CORAL. The CORAL/C++ interface and support for extensibility in CORAL are discussed in Sections 8 and 9. In Section 10 we discuss the performance of the CORAL system using a few representative programs. In Section 11 we mention several applications that have been developed using the CORAL system, to illustrate the utility of deductive database systems in practice. We discuss related systems in Section 12. Finally, we provide a retrospective discussion of the CORAL design and outline future research directions in Section 13.

2 Declarative Language Features

We describe the declarative language provided by CORAL, informally presenting some concepts such as constants, variables, terms, facts and rules along the way. Formal definitions of these concepts may be found in logic programming texts such as [Llo87].

2.1 Declarative Language: Syntax and Semantics

The CORAL syntax is modeled largely after Prolog. Numbers, identifiers beginning with lower-case letters and quoted strings are constants, and identifiers that begin with an upper-case letter are variables.

Consider a database with an *employee* relation having three attributes: *name*, *department* and *salary*, and the following facts:

$$employee(\text{“}John\text{”}, \text{“}Toys\ for\ Tots\text{”}, 35000).$$
$$employee(\text{“}Joan\text{”}, \text{“}Toys\ for\ Tots\text{”}, 30000).$$

The above facts can be interpreted as follows: The first fact indicates that “*John*” is an employee in the “*Toys for Tots*” department and has a salary of 35000. The second fact indicates that “*Joan*” works for the same department and has a salary of 30000.

Constants and variables constitute simple terms. In order to express structured data, complex terms are required. Complex terms are constructed using *functors*, i.e., uninterpreted function symbols as record constructors. (Functors are represented using identifiers beginning with a lower-case letter.) Such terms can be arbitrarily nested. The following fact illustrates the use of complex terms:

$$address(\text{“}John\text{”}, residence(\text{“}Madison\text{”}, street\_add(\text{“}Oak\ Lane\text{”}, 3202), 53606)).$$

This fact can be interpreted as follows: The residence of “*John*” is “3202 *Oak Lane*” in the city of “*Madison*”, and has a zip of 53606. The function symbols *residence* and *street_add* are used as record constructors.

Rules in CORAL take the form:

$$p(\bar{t}) :- p_1(\overline{t_1}), \ldots, p_n(\overline{t_n}).$$

The semantics of CORAL rules is based on a declarative reading of the rules, unlike Prolog which has an operational reading of the rules. Informally, a rule is to be read as an assertion that for all assignments of terms to the variables that appear in the rule, the head is true if each literal in the body is true. (In particular, a fact

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

Figure 1: The Append Program

is just a rule with an empty body.) A CORAL program is a collection of rules. (These rules may be organized into modules, as we discuss later.)

It should be emphasized that a declarative language allows the programmer to express the *meaning* of a program, but offers no guarantee of execution strategy or order. This implies that declarative programs should not use features (like updates) that have side-effects[3].

In the deductive database literature, it is common to distinguish a set of facts as the EDB or *extensional database*, and to refer to the collection of rules as the IDB or the *intensional database*. The significance of the distinction is that at compile time, only the IDB, and possibly meta-information about the EDB, such as schema and functional dependency information, are examined; the actual contents of the EDB are assumed to be unavailable at compile time. Thus, the IDB is viewed as a program and the EDB as the input to the program.

A principal attraction of the logic programming paradigm is that there is a natural meaning associated with a program. As we have seen, each fact and rule can be read as a statement of the form "if <something is true> then <something else is also true>". In the absence of rules with negation, set-generation and aggregation, the meaning of a program can be understood by reading each of the rules in the program in this manner, with the further understanding that the only true facts are those that are either part of the input EDB or that follow from a repeated use of program rules. More formally, the semantics of CORAL programs is given by the least fixpoint (see Lloyd [Llo87], for example) of the program, with the EDB as the input to the program.

CORAL goes much further towards supporting this simple semantics than logic programming languages like Prolog. For programs with only constants and variables as terms and without negation, set-grouping or aggregation, i.e. DATALOG programs, this simple semantics is guaranteed. (More precisely, the default evaluation strategy is sound, complete and terminates for this class of programs.) It is possible that the set of relevant inferences is infinite in the presence of terms with function symbols; in this case, termination cannot be guaranteed, but the evaluation is still sound[4]; evaluation is also complete if it terminates.

In subsequent sections, we discuss more advanced features of the CORAL declarative language, such as non-ground facts, negation, set-generation and aggregation.

2.2 Modules

CORAL users can organize sets of rules and facts into *modules*. We introduce the syntax using a program to append two lists, given in Figure 1. This program illustrates the notion of modules, and CORAL's support for complex objects such as lists.[5]

Modules can *export* the predicates that they define; a predicate exported from one module is visible to all other modules. The *export* statements also define what forms of external queries are permitted on the module ("b" denotes an argument that must be bound in the query, and "f" an argument that can be free). For example, one

[3] CORAL however does offer one evaluation mode called *pipelining* that offers an explicit guarantee of a fixed evaluation strategy, and thus permits a meaningful use of such features within a program. This is discussed in greater detail in Section 6.

[4] The "occur check" has been omitted from the current implementation of CORAL, as in all Prolog systems, for reasons of efficiency. This compromises soundness for programs with non-ground terms and functors.

[5] The notation for list terms follows Prolog. A list is written in the form $[elem_1, elem_2, \ldots, elem_n]$; [ ] denotes the empty list. Given an element e and a list l, $[e|l]$ denotes the list obtained by adding e to the front of l. A list $[H|T]$ can be unified with a given non-empty list $[elem_1, elem_2, \ldots, elem_n]$ by binding H to $elem_1$, which is the head of the given list, and binding T to the list $[elem_2, \ldots, elem_n]$, which is the tail of the given list. The tail of a list of the form $[elem_1]$ is [ ].

can ask the following queries on the *listroutines* module in Figure 1: ?$append([1,2],[3,4],X)$, which corresponds to the bbf adornment, and ?$append([1,2],X,[1,2,3,4])$, which corresponds to the bfb adornment.

2.3 Non-Ground Facts

CORAL permits variables within facts. As an example, consider Figure 1. It is possible to query *append* as follows:[6]

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 meaning 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 set of ground facts. Such facts are often useful in knowledge representation, natural language processing and could be particularly useful in a database that stores (and possibly manipulates) rules. Non-ground facts may also be useful to specify constraint facts [Ram88, KKR90], although they are not supported currently in CORAL. Since CORAL allows non-ground facts, rules are not required to be range-restricted[7]. To the best of our knowledge, CORAL is the only deductive database system, other than XSB [SSW93], to support non-ground facts.

2.4 Negation

CORAL supports a class of programs with negation that properly contains the class of non-floundering left-to-right modularly stratified programs ([Bry89, Ros90]). A program is *non-floundering* if all variables in a negative literal are ground before the literal is evaluated (in the left-to-right rule order). Intuitively, a *modularly stratified* program is one in which the answers and sub-queries generated during program evaluation involve no cycles through negation. This class of programs properly includes the class of programs with *locally stratified* negation [Prz88]. For programs without negation, this semantics coincides with the least fixpoint semantics.

The keyword "*not*" is used as a prefix to indicate a negative 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.

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, all of which are known to be working. This is expressed by the following program.

$$\begin{array}{ll} working(X) & :-\ tested(X). \\ working(X) & :-\ part(X,Y), not\ has\_suspect\_part(X). \\ has\_suspect\_part(X) & :-\ part(X,Y), not\ working(Y). \end{array}$$

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 sub-components, 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. This is described further in Section 6.5.1.

[6] The current CORAL implementation by default performs certain optimizations that assume the absence of non-ground facts. These optimizations do not affect this query. In general, if non-ground facts might be generated during the evaluation of a module, these optimizations should be disabled, adding an annotation "@non_ground_facts +." to the module.

[7] A rule is *range-restricted* if every variable in the head of the rule also appears in the body. Non-ground facts in the database are actually a special case of non-range-restricted rules where the body is empty.

2.5 Sets and Multisets

Sets and multisets are allowed as values in CORAL. An example of a set is $\{1, 2, 3, f(a, b), a\}$, while $\{1, f(a), f(a)\}$ is an example of a multiset. Sets and multisets can contain arbitrary values as elements. Since CORAL allows arbitrarily nested structures, the universe of discourse is an extended Herbrand universe which includes sets [BNST91], as in LDL, and multisets, rather than the Herbrand universe which is standard in logic programming.

There are two ways in which sets and multisets can be created using rules, namely, set-enumeration ({ }) and set-grouping (<>); the syntax is borrowed from LDL [NT89], but there are some differences in semantics which we discuss later.

The following fact illustrates the use of set-enumeration:

$$children(john, \{mary, peter, paul\}).$$

The following rule illustrates the use of set-grouping:

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

This rule uses facts for q to generate a multiset S of instantiations for the variables X, Y, and Z. For each value x of 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, given facts $q(1, 2, 3), q(1, 2, 5)$ and $q(1, 3, 4)$ the above rule derives the fact $p(1, \{2, 2, 3\})$.

The use of the set-grouping construct in CORAL is similar to the grouping construct in LDL — however, 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 *makeset* 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.

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

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.

CORAL requires 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 meaning is hard to define, or whose evaluation would be inefficient.[8] The modularly stratified semantics [Ros90], although originally described for negation, can be easily extended to set-generation.

General matching or unification of sets (where one or both of the sets can have variables) is not supported in CORAL, unlike in LDL, which supports set-matching. The evaluation mechanism for set-matching 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 [STZ92]. 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 either another (identical) ground set-term or a variable. We believe that most, if not all, uses of set matching can be implemented naturally using the suite of functions (such as *member*), that CORAL provides on sets; we present an example in the next section.

[8] LDL imposes the more stringent restriction that uses of grouping be stratified. We note that while EKS-V1 [VBKL90] 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

2.6 Operations on Sets and Multisets

CORAL provides several standard operations on sets and multisets as system-defined predicates. These include $member, union, intersection, difference, multisetunion, cardinality, subset$, and $makeset$. For reasons of efficiency, most of these are restricted to testing, and will not permit generation — for example, the *subset* predicate can be used to test if a given set is a subset of another but cannot be used to generate subsets of a set. The predicate *member* is an exception in that it can be used to generate the members of a given set.

CORAL allows several *aggregate operations* to be used on sets and multisets: these include $count, min, max$, $sum, product, average$ 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.

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

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 weighted edges. (The program as written is not efficient, and may loop for ever on some data sets; in Section 6.6.3 we describe how annotations may be used to generate an efficient version of the program.)

The following example illustrates the use of *member* to generate the elements of a set.

$$\begin{aligned} ok\_team(S) :- \; & old\_team(S), count(S,C), C \leq 3, member(X,S), member(Y,S), \\ & member(Z,S), engineer(X), pilot(Y), doctor(Z). \end{aligned}$$

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 literals $count(S,C), C \leq 3$ in the body of the rule ensure this.

3 Interactive System Environment

CORAL presents users with an interactive environment for program development and ad-hoc querying.[9] This interface resembles the interface provided by typical Prolog interpreters. A number of utility commands that manipulate various system defaults that affect optimization and evaluation choices are available from this interface. We now describe some of these features.

[9] CORAL can also be accessed via its interface with C++, as described in Section 8.

3.1 Update Facilities

The CORAL system permits updates to base relations via imperative rules that can be evaluated at the command prompt. (Files containing a sequence of imperative rules can also be consulted from the command prompt.) These rules can be of one of the following forms:

- *head* $:=$ *body*. : assigns all qualifying tuples to head relation.
- *head* $+=$ *body*. : adds all qualifying tuples to head relation.
- *head* $-=$ *body*. : deletes all qualifying tuples from head relation.

The syntax of the head and the body of the rules is the same as in declarative rules within modules.

If the head predicate also appears in the body of the rule, and head facts corresponding to successful rule instantiations are immediately inserted/deleted, the result of the application of the imperative rule could become order-dependent, which is undesirable. To avoid this problem, CORAL uses a *delayed update* semantics: the body is fully evaluated, and (the multiset of) all head tuples from successful instantiations are inserted into, or deleted from, the appropriate relation.

CORAL supports transactions on disk-resident relations. Commands to initiate and terminate a transaction can be invoked from the CORAL prompt; at any time, only one transaction can be active within a single CORAL process.

3.2 Data Organization

In CORAL, data is stored as tuples in relations. Relations themselves can be organized into named workspaces. A *workspace* is a collection of relations, which can be either EDB relations or relations corresponding to predicates exported by modules. A user can have several named workspaces, copy relations from one workspace to another (or simply make a relation in one workspace visible from another without copying), update relations in a workspace, or run queries against a workspace. It is also possible to save a workspace as a text file between executions. There is always a current workspace, and new workspaces can be created interactively. Data can be loaded into the current workspace either by explicitly inserting facts into relations or by consulting text files that hold the data (as in Prolog systems).

Persistent relations exist in a special workspace and can be made visible to all other workspaces without copying. When a workspace that refers to a persistent relation is saved, only the name of the persistent relation — and not its current set of tuples — is saved.

3.3 Program Development

Some basic facilities are provided for debugging programs. A `trace` facility is provided that does the following: (1) It lets the user know which rules are being evaluated, and (2) It prints out answers and sub-queries (of specified predicates) as they are generated, to let the user know how the computation is proceeding.

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 about the efficiency of indexing, by keeping counts of the number of tuples that the indexing operation tried to unify, and the number that successfully unified and were retrieved. In addition, other counts such as the 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.

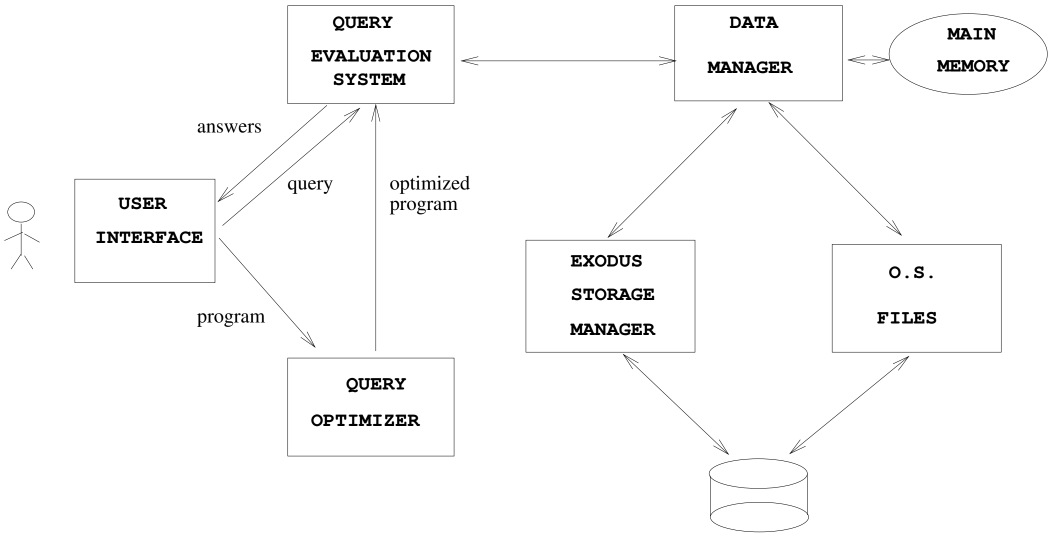

Figure 3: Architecture of the CORAL Deductive System

3.4 Explaining Program Execution

An explanation tool has been implemented that provides graphical explanations of the executions of declarative programs. The basis of this tool is that one can understand the meaning of a program in terms of the set of derivation trees of computed facts. Derivation trees can be "grown" and "pruned" dynamically on the screen, thereby providing a visual explanation of how facts were generated. The explanation mechanism can be enabled or disabled on a per-module basis. The explanation tool has been implemented as an application of the CORAL system; see Section 11 for further details.

We note that derivations are recorded in the exact form that they are carried out. Thus, if the user's program was rewritten by the system optimizer, the recorded derivations reflect the rewritten program, and it can sometimes be hard to see the mapping between the original and rewritten programs. However, with the default rewriting (Supplementary Magic rewriting) used by the CORAL system the mapping between the original program and the rewritten program is simple, and the user should be able to reason in terms of the original program when presented with derivations of the rewritten program.

4 Architecture of the CORAL System

The architecture of the CORAL deductive system is shown in Figure 3. CORAL is designed primarily as a single-user database system, and can be used in a stand-alone mode; however, data can be shared with other users via the EXODUS storage manager. Persistent data is stored either in text files, or using the EXODUS storage manager [CDRS86], which has a client-server architecture. Each CORAL process can act as an EXODUS client that accesses the common persistent data from the server. Multiple CORAL processes could interact by accessing persistent data stored using the EXODUS storage manager. Transactions and concurrency control are supported by the EXODUS storage manager, and thus by CORAL. However, within each CORAL process, all data that is *not* managed by the EXODUS storage manager is strictly local to the process, and no transactions are supported on such data.

Data stored in text files can be "consulted", at which point the data is converted into main-memory relations; indices can then be created. Data stored using the EXODUS storage manager is paged into EXODUS buffers on demand, making use of the indexing and scan facilities of the storage manager. The design of the system does not require that this data be collected into main-memory CORAL structures before being used; as is usual in database systems, the data can be accessed purely out of pages in the EXODUS buffer pool.

The query processing system consists of two main parts — a query optimizer and a query evaluation system. Simple queries (selecting facts from a single relation or multiple joined relations, for instance) can be typed in at the user interface. Such simple queries do not require rewriting transformations. Complex queries are typically defined in declarative "program modules" that export predicates (views) with associated "query forms" (i.e., specifications of what kinds of queries, or selections, are allowed on the predicate). The query optimizer takes a program module and a query form as input, and generates a rewritten program that is optimized for the specified query forms. In addition to performing source-to-source transformations[10], the optimizer adds several *control annotations* (to those, if any, specified by the user). The rewritten program is stored as a text file — which is useful as a debugging aid for the user — and is also converted into an internal representation that is used by the query evaluation system.

The query evaluation system takes as input annotated declarative programs (in an internal representation), and database relations. The annotations in the declarative programs provide execution hints and directives. The query evaluation system *interprets* the internal form of the optimized program. We also developed a *fully compiled* version of CORAL, in which a C++ program was generated from each user program. (This is similar to the approach taken by LDL [NT89, CGK$^+$90].) We found that this approach took a significantly longer time to compile programs, and the resulting gain in execution speed was minimal[11]. We have therefore focused on the interpreted version; "compiling" a program to CORAL internal structures takes very little time, and is comparable to Prolog systems. This makes CORAL very convenient for interactive program development.

The query evaluation system has a well defined "get-next-tuple" interface with the data manager for access to relations. This interface is independent of how the relation is defined (as a base relation, declaratively through rules, or through system-defined or user-defined C++ code), and is quite flexible. In conjunction with the modular nature of the CORAL language, such a high-level interface is very useful, since it allows the different modules to be evaluated using different strategies. It is important to stress that the "get-next-tuple" interface is merely an abstraction provided to support modularity in the language, and does not affect the ability to perform set-oriented computations.

While fundamental decisions such as using a bottom-up fixpoint computation are motivated by the potential for set-oriented evaluation, it is important to note the current implementation does not exploit this potential fully. For example, although the interface to EXODUS does page-level I/O, index nested-loops join is used even for disk-resident data; unless the index on the inner relation is a clustered index, performance may be poor. It would be a relatively straightforward matter to add more efficient external join methods such as blocked nested-loops or sort-merge, and we are currently working on such extensions. Further, the lack of a traditional cost-based query optimizer (e.g., for choosing a good join order in each rule) is another major gap in the current system, and again, this can be remedied with some effort. (However, some difficult issues remain little understood; for instance, should the join order be determined afresh on each iteration? Derr [Der93] suggests some heuristics.)

CORAL supports an interface to C++, and can be embedded in C++. C++ can be used to define new relations as well as manipulate relations computed using declarative CORAL rules. The CORAL/C++ interface is intended to be used for the development of large applications.

5 Overview of Query Evaluation and Optimization

A number of query evaluation strategies have been developed for deductive databases, and each technique is particularly efficient for some programs, but may perform relatively poorly on others. Thus any system that is tied to one evaluation strategy is bound to perform poorly on some programs. Indeed, this is also the case for relational systems such as SQL. However, given the greater complexity of a language like CORAL, it is harder to design a system in which different optimization techniques can be combined relatively orthogonally. Once this is done, a cost estimation package can be used to determine a "good" optimized version of a program. CORAL

[10] The query optimizer invokes several different program rewriting filters, which we discuss later.

[11] We note that the compiled version did not exploit various opportunities for optimization that do not exist with the interpreted approach. A more aggressive version of the compiler would probably be faster.

addresses the first task to a large extent, but currently uses heuristics instead of a cost estimation package to make choices of evaluation alternatives.

It is our premise that in such a powerful language, completely automatic optimization can only be an ideal; the programmer must be able to provide hints or *annotations* and occasionally even override the system's decisions in order to obtain good performance across a wide range of programs. Annotations control query evaluation and guide query optimization. Since they are expressed at a high level, they give the programmer the power to control optimization and evaluation in a relatively abstract manner. A detailed description of the annotations provided by CORAL may be found in [RSSS93a]; we mention some of them when discussing the query evaluation techniques.

The CORAL programmer decides (on a per-module basis) whether to use one of two basic evaluation approaches, namely *pipelining* or *materialization*, which are discussed in Section 6. Many other optimizations are dependent upon the choice of the basic evaluation mode. The optimizer generates annotations that govern many run-time actions, and, if materialization is chosen, does source-to-source rewriting of the user's program. We discuss these two major tasks of the optimizer below.

5.1 Source-to-Source Rewriting Techniques

Materialized evaluation in CORAL is essentially a fixpoint evaluation using a bottom-up iteration on the program rules. If this is done on the original program, selections in a query are not utilized. Several program transformations have been proposed to "propagate" such selections, and many of these are implemented in CORAL.

The desired selection pattern is specified using a query form, where a "bound" argument indicates that any binding in that argument position of the query is to be propagated. It is possible in fact that the query does not specify a ground binding in a "bound" position (it may specify no value, or a term with variables). This simply results in non-ground "magic facts". Thus, by specifying that all arguments are bound, binding propagation similar to Prolog is achieved (i.e., all available bindings are propagated). By specifying that all arguments are "free", in contrast, bindings in the query are ignored, except for a final selection. Bindings in certain arguments can be selectively propagated by choosing other query forms.

The default rewriting technique is Supplementary Magic Templates ([BR87b, Ram88]; see also [RLK86, Sek89]). The rewriting can be tailored to propagate bindings across sub-queries in a rule body using different body literal orderings; CORAL uses a left-to-right ordering within the body of a rule by default. Other selection-propagating rewriting techniques supported in CORAL include Magic Templates [Ram88], Supplementary Magic With GoalId Indexing [RS91], and Context Factoring [NRSU89, KRS90]. Supplementary Magic is a good choice as a default, although each technique is superior to the rest on some programs. For example, the GoalId Indexing variant is good for programs with many complex, especially non-ground, terms, and Context Factoring, while only applicable on some programs, is usually superior when applicable. The user can choose the rewriting to be applied through annotations. If no selection is specified, it may be appropriate to apply no rewriting, and the user can specify this as well.

CORAL also supports Existential Query Rewriting [RBK88], which seeks to propagate projections. This is applied by default in conjunction with a selection-pushing rewriting. A discussion of the relative merits of these rewriting techniques is not possible here due to shortage of space.

5.2 Decisions On Run-time Alternatives

In addition to choosing rewriting techniques for materialized evaluation, the optimizer makes a number of decisions that affect execution. The optimizer analyzes the (rewritten) program, and identifies some evaluation and optimization choices that appear appropriate.

The default fixpoint evaluation strategy is called Basic Semi-Naive evaluation (BSN), but a variant, called Predicate Semi-Naive evaluation (PSN) [RSS94], which is better for programs with many mutually recursive predicates, is also available. With respect to semi-naive evaluation, the optimizer is responsible for: (1) join order selection, (2) index selection, (3) deciding what forms of subsumption checks to use, and (4) deciding whether to refine the basic indexed nested-loops join with intelligent backtracking. The optimizer is also responsible for deciding whether to use variations of the fixpoint evaluation such as Lazy Evaluation or Ordered Search. We discuss these issues in Section 6.

6 Module Evaluation Strategies

The evaluation of a declarative CORAL program is divided into a number of distinct sub-computations by expressing the program as a collection of modules. Each module is a unit of compilation and its evaluation strategies are independent of the rest of the program. Since different modules may have widely varying evaluation strategies, a relatively high-level interface is required for interaction between modules.

Two basic evaluation approaches are supported, namely *materialization* and *pipelining*. Materialization stores facts and looks them up to avoid recomputation. Several variants of materialized evaluation are supported: Basic Semi-Naive, Predicate Semi-Naive [RSS94], and Ordered Search [RSS92a]. Pipelining uses facts "on-the-fly" and does not store them, at the potential cost of recomputation.

This section presents the interface between modules and the run-time data structures used. The various modes of evaluation of a module, and the ways in which the evaluation can be controlled by annotations from the user are then discussed.

6.1 Inter-Module Calls

Suppose that p is a predicate that appears in the body of a rule of module $M2$ and is not defined in $M2$. During the evaluation of $M2$, queries may be generated on p. If p is defined in module $M1$, then module $M2$ sets up an inter-module call on module $M1$ to solve the query. The interface to predicates exported by a module makes no assumptions about the evaluation of the module. Module $M1$ 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 $M1$ makes no assumptions about the evaluation of calling module $M2$. 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.

Inter-module calls are executed as follows: The calling module sets up a sub-query on the called module, and waits until the called module returns answers to the sub-query. The called module returns either no answer if the query has no answers, or returns one or more answers to the query. The called module may or may not return all answers immediately; repeated "get-next-tuple" calls must be used to get all answers to the call. The above interface is independent of the evaluation modes of the two modules involved. However, the order in which answers are returned on the initial call and on subsequent "get-next-tuple" requests depends on the evaluation mode of the called module.

The rationale behind the particular interface described above is as follows: the calling module may require only one answer, or may use only one answer at a time, e.g., if the inter-module call was generated by a literal involved in a nested-loops join. Early returning of answers to the user via Lazy Evaluation is also supported by this interface. The alternative interface of requiring all answers to be returned is less flexible.

6.2 Module and Rule Data Structures

The compilation of a materialized module generates an internal *module structure* that consists of a list of structures corresponding to the strongly connected components (SCCs) of the module.[12] Each SCC structure consists of *semi-naive rule structures* corresponding to semi-naive rewritten versions of the rules. These semi-naive rule structures have fields that specify the argument lists of each body literal, and the predicates they correspond to.

[12]An SCC in a module is a maximal set of mutually recursive predicates.

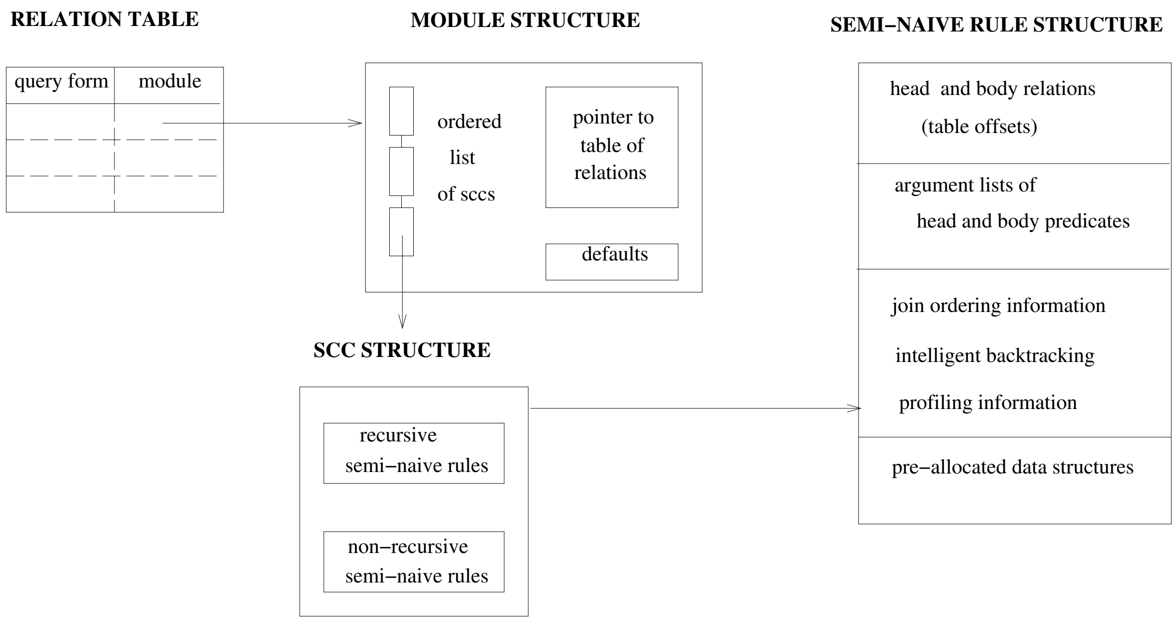

Figure 4: Important Run-time Data Structures

Each semi-naive rule structure also contains evaluation order information, pre-computed backtrack points, and pre-computed offsets into a table of relations. These structures are shown in Figure 4.

Offsets into a table of relations are used, instead of actual pointers to relations, because an important consideration was to keep rule and module evaluation re-entrant. This property is essential to support multiple concurrent invocations of the same module, as can happen with a cycle of inter-module invocations or Lazy Evaluation (which is discussed in Section 6.3.1). Information that changes with each invocation cannot therefore be stored with the rule structure. In particular, the actual relations involved change across invocations, and each module invocation has an associated table of (pointers to) relations. The order of relations corresponding to different predicates in this table can be determined, and the semi-naive rule structures refer to relations involved in the rule by offsets into this table.

A module to be evaluated using pipelining is stored as a list of predicates defined in the module. Associated with each predicate is a list of rules defining it (in the order they occur in the module definition), each rule being represented using structures like those used for semi-naive rules.

An attempt is made to compute all possible information at compile time, and store it along with the rule structures so as to make rule evaluation efficient. In a manner similar to Prolog, CORAL maintains a trail of variable bindings when a rule is evaluated; this is used to undo variable bindings when the indexed nested-loops join (or pipelining) considers the next tuple in any loop.

6.3 Materialization

The variants of *materialization* are all bottom-up fixpoint iteration methods, which repeatedly evaluate the rules until a fixpoint is reached. In order to perform incremental evaluation of rules across multiple iterations, CORAL uses semi-naive evaluation [Bay85, Ban85, BR87a, RSS94]. This technique consists of a rule rewriting part performed at compile time, which creates versions of rules with *delta relations*, and an evaluation part. (The delta relations contain changes to relations since the previous iteration.) The evaluation part evaluates each rewritten rule once in each iteration, and performs some updates to the relations at the end of each iteration. An evaluation terminates when an iteration produces no new facts.

The join order used in CORAL is currently left-to-right in the rule, with a simple reordering that moves delta relations to the front of the join order. The reordering is done with the expectation that the delta relations have a smaller number of tuples than the other relations. However, hooks have been provided to specify other join orders, and the optimizer can be modified to find good join orders.

The optimizer also analyzes the semi-naive rewritten rules and generates annotations to create any indices that may be useful during the evaluation phase.[13] The basic join mechanism in CORAL is nested-loops with indexing, and this knowledge is used by the index generation algorithm. For each semi-naive rule, index generation proceeds left-to-right in the join order of rule evaluation, and creates argument-form or pattern-form indices based on variables bound earlier in the join order, since these will act as selection arguments. For derived relations, these indices are created at module initialization time (run-time) and are destroyed, along with the relation, when module execution is completed. Indices can also be explicitly requested by a CORAL user.

For declarative modules, CORAL materialized 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.[14] 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. It is desirable to add a compile time check, based on sufficient conditions,[15] to determine if termination and completeness of CORAL evaluation can be guaranteed for a given program, but the current implementation does not support such a feature.

6.3.1 Lazy Evaluation

In the traditional approach to bottom-up evaluation, all answers to a query are computed by iterating over rules until a fixpoint is reached, and then returning all the answers. Lazy evaluation is the technique used by CORAL to return answers at the end of every iteration, instead of just at the end of the computation. The advantages of this approach are:

- It is useful for users who would rather have a steady stream of answers than wait until the end of the computation and then receive a burst of answers.
- It can be used in an interactive mode to look at the first few answers generated by a computation, and then (possibly) abort the rest of the computation.

A query on a relation has an iterator associated with it. Lazy evaluation is implemented by storing in the iterator the state of the computation at the end of an iteration of fixpoint evaluation, and returning to the iterator the answer tuples generated in that iteration. The iterator then iterates over the tuples returned, and when it has stepped through all the tuples, it reactivates the "frozen" computation, in order to get more answer tuples. This reactivation results in the execution of one more iteration of the rules, and the whole process is repeated until an iteration over the rules produces no new tuples.

6.4 Pipelining

Pipelining in CORAL is similar to top-down evaluation like Prolog. The rule evaluation code for pipelining is designed to work in a co-routining fashion — when rule evaluation is invoked, using the *get-next-tuple* interface, an answer is generated (if there is one) and control is transferred back to the consumer of the answers (the caller). When more answers are desired, control is transferred back to the (suspended) rule evaluation.

At module invocation, the first rule in the list associated with the queried predicate is evaluated. This could involve recursive calls on other rules within the module (which are also evaluated in a similar pipelined fashion). If the rule evaluation of the queried predicate succeeds, the state of the computation is frozen, and the generated answer is returned. A subsequent request for the next answer tuple results in the reactivation of the frozen computation, and processing continues until the next answer is returned. At any stage, if a rule fails to produce

[13] Index annotation generation also occurs for base relations used in pipelined modules, but at the level of the original rules.

[14] However, for reasons of efficiency, the current implementation does not perform occur checks by default.

[15] Sufficient conditions are needed since checking for termination is undecidable in general.

an answer, the next rule in the rule list for the head predicate is tried. When there are no more rules to try, the query on the predicate fails. When the topmost query fails, no further answers can be generated, and the pipelined module execution is terminated.

An interesting aspect of pipelining in CORAL is the treatment of recursive predicates. A sub-query on a 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 (although CORAL does not currently support all the extra-logical features of Prolog). Of course, pipelined evaluation of recursive modules carries the risks of potential incompleteness, and should be used with care.

We note that our implementation of pipelining handles recursive calls; in this, it differs from the "pipelining" used in LDL. The latter is essentially indexed nested-loops join without materialization within a bottom-up fixpoint iteration. Our implementation of pipelining can however be improved upon; indeed, state-of-the-art Prolog systems are much faster.

There are some important points to note regarding pipelining. First, the implementation of pipelining, which is a radically different evaluation technique from bottom-up fixpoint evaluation, demonstrates the modularity of the CORAL implementation. Second, from a language point of view, it demonstrates that the module mechanism allows a user to effectively combine bottom-up and top-down evaluation techniques in a single program. (Indeed, our implementation of pipelining could be replaced by an interface to a Prolog system.) Third, pipelining guarantees a particular evaluation strategy, and order of execution. While the program is no longer truly "declarative," programmers can exploit this guarantee and use predicates like updates that involve side-effects.

Materialization and pipelining complement each other. If facts in a relation are used many times, the cost of materialization (generating and storing facts) is outweighed by the savings in avoiding recomputation. Pipelining avoids these overheads of storing facts, and if sub-queries are not generated multiple times it is cheaper than materialization.

6.5 Module Level Control Choices

At the level of the module, a number of choices exist with respect to the evaluation strategy for the module, and the specific optimizations to be used. We have already seen the issue of materialization versus pipelining.

6.5.1 Ordered Search

CORAL uses the Ordered Search evaluation mechanism to order the use of generated sub-queries in a program. The reader is referred to [RSS92a] for details. However, some of the features of Ordered Search are mentioned here:

1. It maintains information about dependencies between sub-queries, and can be used to evaluate a large class of programs with negation, set-grouping and aggregation.

2. It provides an ordering to the computation by "hiding" sub-queries.

   When CORAL returns an answer to the user, the user has the option of terminating the computation, or having the system continue to find more answers. When a single answer to the query is all that is needed, there may be many sub-queries that are still hidden when an answer to the query is found, and the computation can terminate without ever using these sub-queries; thus a lot of redundant computation may be avoided.

In Ordered Search, sub-queries and answers to sub-queries are generated asynchronously, as in bottom-up evaluation of programs rewritten using Magic rewriting [BR87b, Ram88]. However, sub-queries are not made available for use immediately — the order in which generated sub-queries are made available for use is somewhat similar to a top-down evaluation. This is achieved by maintaining a "context" which stores sub-queries in an ordered fashion, and at each stage in the evaluation deciding which sub-query to make available for use next.

Ordered Search provides an important evaluation strategy for programs with negation, set-grouping and aggregation, that are left-to-right modularly stratified. Without Ordered Search, evaluation proceeds by setting up inter-module calls whenever a sub-query that has to be completely solved is encountered. This results in computation of such sub-queries proceeding independently, with no sharing of sub-computations. While this might be desired in some situations, it could result in considerable repeated computation in general. Ordered Search is used to evaluate left-to-right modularly stratified programs without inter-module calls; thus, sub-computations are shared, eliminating repeated derivations. Intuitively, all queries and answers are memoed, and enough dependency information between queries is maintained to ensure that any sub-query, for example a negative sub-query, that must be fully evaluated before any of its answers are used in further derivations, is indeed fully evaluated.

From an implementation perspective, in addition to maintaining the context, two changes have to be made. First, the rewriting phase, which must use a version of Magic in conjunction with Ordered Search fixpoint evaluation, must be modified to introduce "done" literals guarding negative literals and rules that have grouping and aggregation. Second, the evaluation must add a "magic" fact to the corresponding "done" predicate when (and only when) all answers to it have been generated. (The context mechanism is used to determine the point at which a query is considered done.) These changes ensure that rules involving negation, for example, are not applied until enough facts have been computed to reduce the negation to a set-difference operation.

6.5.2 The Save Module Facility

The module facility in CORAL provides several important advantages.

1. By moving many rules out of a module into another 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 sub-queries on the "lower" module have been returned.

2. 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 of decreasing the number of semi-naive rewritten rules considerably if many predicates can be treated as base predicates.

3. In many 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 sub-queries computed by a module at the end of a call to the module.

However, there are some cases where the first two benefits above 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 sub-query 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.

From an implementation viewpoint, the challenge is to ensure that no derivations are repeated across multiple calls to the module. Ensuring this requires some subtle changes to the semi-naive technique. Essentially, materialized views are to be incrementally maintained via semi-naive evaluation under inserts.

This requires significant changes to semi-naive evaluation: rules defined only in terms of base predicates must only be used in the first invocation of the module, certain predicates that were treated as "base" in the semi-naive rewriting because they appeared in lower SCCs must now be treated as derived predicates, and the updating of relations using deltas has to be modified to take into account tuples that were computed in previous calls to the module. While the details are omitted here for lack of space, they can be found in [RSS93]. In the interest of efficient implementation, CORAL requires that if a module uses the save module feature, it should not be invoked recursively. (Note that the predicates defined in the module can be recursive; this does not cause recursive invocations of the module).

6.6 Predicate Level Control

CORAL provides a variety of annotations at the level of individual predicates in a module. These annotations could affect the set of answers returned to a query.

6.6.1 Duplicate Elimination

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.[16] An annotation tells the system not to perform duplicate checks for all predicates in the module. This can also be done on a per-predicate basis. Further, a predicate in a program can be declared to be a *multiset*, i.e., with as many copies of a tuple as there are derivations for it.[17] CORAL then guarantees that the number of copies of tuples in the answer to a query on the predicate is correct according to the multiset semantics of the program [MPR90]. This semantics is supported by carrying out duplicate checks only on the "magic" predicates if any version of the Magic Templates rewriting is used.

6.6.2 Index Annotations

CORAL allows for the specification of two types of hash-based indices: (1) *argument form indices*, and (2) *pattern form indices*. The first form is the traditional multi-attribute hash index on a subset of the arguments of a relation. The hash function chosen works well on ground terms; however, all terms that contain variables are hashed to a special value. The second form is more sophisticated, and allows the retrieval of precisely those facts that match a specified pattern, where the pattern is a tuple of (possibly non-ground) terms. The "form" of the pattern must be specified when constructing the index. Such indices are of great use when dealing with complex objects created using functors. For example, suppose a relation *employee* had two arguments, the first a name and the second a complex term $address(Street, City)$. A pattern-form index can be used to efficiently retrieve employees named "$John$", who stay in "$Madison$", without knowing their street [RS91]. The following annotation can be used to create a pattern-form index as above:

@make_index $employee(Name, address(Street, City))$ $(Name, City)$.

Pattern-form indices are implemented in CORAL using hash-indices, and are almost as fast as argument-form indices.

A compile-time analysis of all rules is used to determine which indices need to be created for efficient rule evaluation. All such indices on in-memory relations are automatically created at run-time. However, indices are not automatically created on disk-resident relations. In addition, the user is allowed to specify indices using annotations within a module or using commands from the CORAL prompt.

6.6.3 Aggregate Selections

Consider the *shortest_path* program in Figure 5. This differs from the program in Figure 2 in that it has an additional annotation of the form:

@aggregate_selection $path(X, Y, P, C)$ (X, Y) $min(C)$.

To compute shortest paths between points, it suffices to use only the shortest paths between pairs of points — path facts that do not correspond to shortest paths are irrelevant. CORAL therefore permits the user to specify

[16] If facts contain variables, subsumption checking may be used, rather than just duplicate elimination. CORAL performs some subsumption checking but, for efficiency reasons, does not guarantee that relations are maintained as irredundant sets of facts [MR89].

[17] On non-recursive queries, this semantics is consistent with SQL when duplicate checks are omitted.

```
module shortest_path.
export shortest_path(bfff, ffff).
@aggregate_selection path(X,Y,P,C) (X,Y) min(C).
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 5: Program Shortest Path

an *aggregate selection* on the predicate *path* in the manner shown. The system then retains, for each X, Y pair, only the *path* facts with the lowest C value (among the currently known *path* facts). Without this aggregate selection, the program may run for ever, generating cyclic paths of increasing length; with it, the program is guaranteed to terminate. In the next section, we discuss how to further improve the efficiency of this program using additional annotations.

6.6.4 Using Aggregate Selections to Express Choice

CORAL's aggregate selection mechanism provides a version of the choice operator of LDL, but with a different semantics [RBSS90]. Consider again the shortest path program from Figure 5. If the user wishes to retain a single path for each pair of nodes and each path cost, this can be specified using the following annotation:

$$@aggregate\_selection\ path(X,Y,P,C)\ (X,Y,C)\ any(P).$$

The annotation says that for each value of the triple X, Y, C, at most one fact $path(X,Y,P,C)$ need be retained. If more than one fact $path(X,Y,P,C)$ is generated by the program for any triple X, Y, C, the system arbitrarily picks one of the facts to retain, and discards the rest.

Using a combination of the above two aggregate selections on the shortest path program (in conjunction with the default query evaluation technique using Magic rewriting), 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.

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.

6.6.5 Controlling the Order of Deductions

The use of facts computed during bottom-up evaluation can be prioritized.[18] Consider the shortest path program from Figure 5, which 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:

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

[18] The prioritization of facts is relevant to the evaluation mechanism, and is not used to order the answers returned to the used.

Prioritized relations in CORAL are implemented as priority queues. Evaluation with prioritized use of facts uses a simple extension of semi-naive evaluation, described in [SKGB87].

Using facts in a prioritized fashion reduces the cost of evaluation of a single source shortest path problem from a worst case of $O(E \cdot V)$ to $O(E \cdot log(V))$[19] [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 annotations automatically, but they could also be specified by the user.

6.7 Rule Level Control

6.7.1 Intelligent Backtracking

Both pipelining and materialization involve indexed nested-loops joins of relations. (For materialization, this is local to the joins in a single rule; for pipelining, it is effectively global, and over the sequence of all rules in the current execution path.) CORAL tries to perform intelligent backtracking (see, [CD85], for example) during rule execution in both cases.

Get-first-failure (or, get-first) backtracking provides the ability to "jump back" over several levels of nesting if no matching facts are found for a predicate in an inner level of a nested-loops join. It is used when there is no valid instantiation of a body literal the "first" time the literal is reached. At this stage, control within the rule backtracks to the last body literal that could generate new bindings for the literal that just failed. For example, consider the following rule with a left-to-right join order:

$$p(X,Y) :- q(X,Z), r(A,Y), s(Z,B), t(A,B).$$

If $s(Z,B)$ fails the first time it is reached with a particular binding for Z, the "get-first backtrack point" is the literal $q(X,Z)$, because this is the last point at which new bindings can be generated for Z.

Success backtracking provides the ability to "jump back" over several levels of nesting if an answer is generated and the number of times an answer is generated is irrelevant — the idea is that the loops jumped over would only produce more derivations of the same fact and not a new fact. When a rule execution is successful (i.e., there are valid instantiations of all the literals in the body of a rule), a head fact is generated. At this point, the control within the rule backtracks to the last body literal that could generate new bindings for the head literal. Consider the rule in the above example again. The "success-backtrack point" for this rule is the literal $r(A,Y)$, because $s(Z,B)$ and $t(A,B)$ cannot generate any bindings that will result in a new head fact. The logic for success backtracking in CORAL takes advantage of aggregate selections that express choice. For example, an annotation:

@aggregate_selection $p(X,Y)\ (X)\ any(Y)$.

could be added, with the meaning that for a given X value, a value for Y is chosen from a p tuple with the given X value, and p tuples with other Y values can be discarded. Adding the annotation would change the "success-backtrack point" of the above-mentioned rule to $q(X,Z)$.

CORAL automatically performs "get-first-failure" backtracking and "success" backtracking, unless there are non-ground facts. (By default, CORAL assumes that there are none; if it is possible that a program will generate non-ground facts, the user should indicate this through an annotation.) The analysis used for intelligent backtracking breaks down in the presence of such non-ground facts, although it can be extended to detect argument positions that are guaranteed to be ground and to take advantage of such argument positions.

Intelligent backtracking is implemented using an array of backtrack points, one for each body literal, and one success backtrack point for each rule. Intelligent backtracking for pipelined evaluation is also done on a per-rule basis, although there is the potential for doing it on a global basis. We note that LDL also implements a form of intelligent backtracking [CGK89].

[19] Assuming that the edge costs are non-negative.

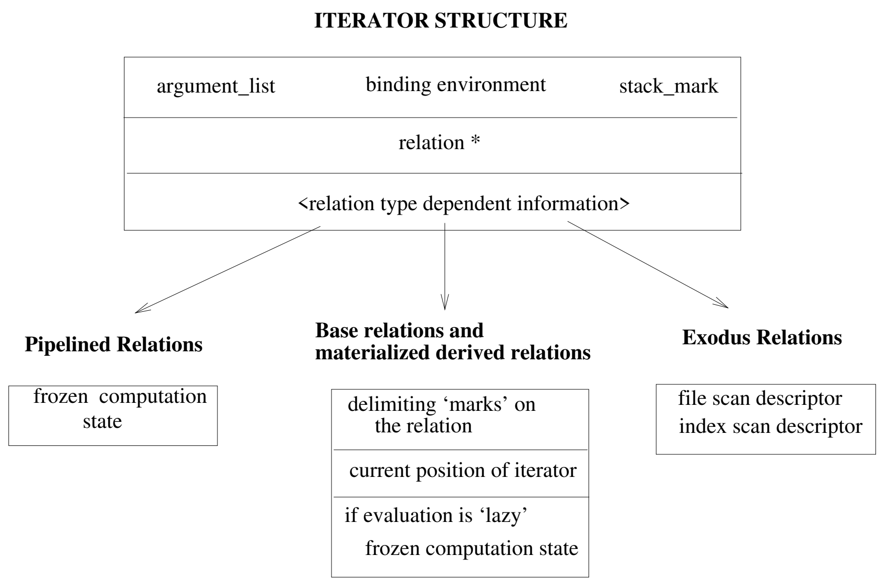

Figure 6: The TupleIterator Structure

6.7.2 Join Orders

CORAL uses a default left-to-right join order, except that for semi-naive rewritten rules the "delta" relation is moved to the beginning of the join order. This is generally a good heuristic, especially when Supplementary Magic rewriting has been used; in this case, moving the "delta" relation to the beginning of the join order does not introduce a cross-product. The user can override this default by specifying the join order on a per-rule or on a per-semi-naive-rewritten rule basis. However, this has not been implemented yet in CORAL.

7 The Data Manager

The data manager (DM) is responsible for maintaining and manipulating the data in relations. In discussing the DM, we also discuss the representation of the various data types. While the representation of simple types is straightforward, complex structural types and incomplete data present interesting challenges. The efficiency with which such data can be processed depends in large part on the manner in which it is represented in the system. This section therefore presents the data representation at a fairly detailed level.

The CORAL system is implemented in C++, and all data types are defined as C++ classes. *Extensibility* is an important goal of the CORAL system. In particular, we view support for user-defined data types as important. In order to provide this support, CORAL provides the generic class Arg that is the root of all CORAL data types; specific types such as integers and strings are sub-classes of Arg. The class Arg defines a set of virtual member functions[20] such as equals, hash, and print, which must be defined for each derived class that is created.

The class Tuple defines tuples of Args. An object of the class Relation is a collection of tuples. The class Relation has a number of virtual member functions defined on it. These include insert (Tuple*), delete (Tuple*), and an iterator interface that allows tuples to be fetched from the relation, one at a time.[21] The iterator is implemented using an object of a TupleIterator class that is used to store the state or position of a scan on the relation, and to allow multiple concurrent scans over the same relation. We show the structure of a TupleIterator in Figure 6. The details of the figure become meaningful as the description of the data structures proceeds.

[20] In C++, a virtual member function in a class is one that can be redefined in derived sub-classes of the class, and further, when the member function is invoked on an object, the member function corresponding to the most specific class to which the object belongs is invoked.

[21] This is analogous to the cursor notion in SQL.

7.1 Representation of Terms

The evaluation of rules in CORAL is based on the operation of *unification* that generates bindings for variables based on patterns in the rule and the data. An important feature of the CORAL implementation of data types is the support for unique identifiers to make unification of large terms very efficient. Such support is critical for efficient declarative program evaluation in the presence of large terms. In CORAL, each new type constructor can define how it generates unique identifiers, independent of how other type constructors construct their unique identifiers (if any); because of this orthogonality, no further integration is needed to generate unique identifiers for terms built using several different kinds of type constructors. This is very important for supporting extensibility and the creation of new user-defined data types. Specific issues on the construction of unique identifiers for several system defined types are detailed later in this section.

7.1.1 Constants

Constants in CORAL can be of one of the primitive data types provided in the CORAL system, such as integers, doubles and strings. The current implementation restricts data that is stored using the EXODUS storage manager to be limited to these primitive types. Such data is stored on disk in its machine representation, while in memory, the data types are implemented as sub-classes of Arg. Extra information is kept with strings to provide for efficient equality checking.

7.1.2 Functor Terms

An example of a term built from an uninterpreted function symbol, or functor, is $f(X, 10, Y)$. Such a term is represented by a record containing (1) the function symbol f, (2) an array of arguments, or pointers to the arguments, and (3) extra information to make unification of such terms efficient. Functor terms are important for representing structured information. For instance, lists (which are a special type of functor term) can be used to represent DNA sequences or stock quote sequences.

The current implementation of CORAL uses hash-consing [Got74] to speed up unification of functor terms. (LDL [CGK+90] also implements hash-consing.) Hash-consing assigns unique identifiers to each (ground) functor term, such that two (ground) functor terms unify if and only if their unique identifiers are the same. We note that such identifiers cannot be assigned to functor terms that contain free variables.

CORAL makes two modifications to the basic hash-consing scheme. First, it performs hash-consing in a lazy fashion, avoiding computing the unique identifiers if they are not used. Second, CORAL allows terms that contain variables, and therefore cannot be assigned unique identifiers. Such terms are tagged after the first attempt to assign them unique identifiers, and CORAL thereby avoids repeated attempts to compute unique identifiers for them.

7.1.3 Variables and Non-Ground Terms

Variables constitute a primitive type in CORAL, since CORAL allows facts (and not just rules) to contain variables; in this, CORAL differs from most other deductive database systems. The semantics of a variable in a fact is that the variable is universally quantified in the fact. Although the basic representation of variables is fairly simple, the representation is complicated by requirements of efficiency when using non-ground facts in rules. We describe the problems briefly.

Suppose we want to make an inference using a rule. Variables in the rule may get bound in the course of an inference. A naive scheme would replace every reference to the variable by its binding. It is more efficient however to record variable bindings in a *binding environment*, at least during the course of an inference. A binding environment (often referred to as a *bindenv*) is a structure that stores bindings for variables. Therefore, whenever a variable is accessed during an inference, a corresponding binding environment must be accessed to find if the variable has been bound. We show the representation of the term $f(X, 10, Y)$, where X is bound to 25 and Y is bound to Z, and Z is bound to 50 in a separate bindenv, in Figure 7.

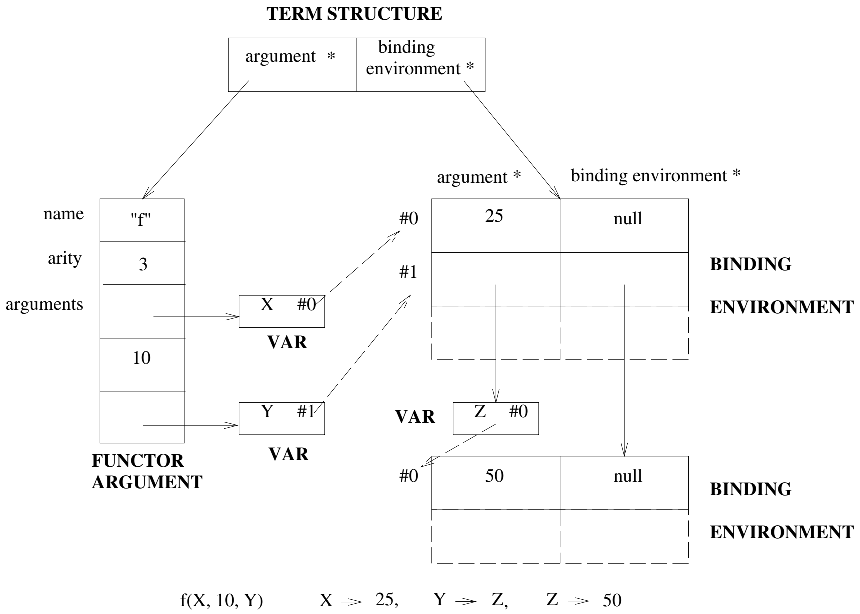

Figure 7: Representation of an Example Term

There is another complication to making inferences using facts that contain variables. The problem is that two facts (or a fact and the rule) may have variables of the same name, but the variables are independently universally quantified. To make an inference, variables in facts have to be renamed to remove name conflicts. Such a renaming could be expensive, and CORAL attempts to avoid renaming or postpone renaming as long as possible. (In many cases, the renaming can be avoided altogether if it is postponed.) To postpone renaming, CORAL maintains a binding environment for each fact that contains a variable, as well as a binding environment for the rule.

The top-down backtracking control strategy of Prolog permits optimizations whereby answers to sub-queries need not be renamed; such optimizations are not applicable to evaluation techniques, such as bottom-up evaluation, that perform memoization of facts. Techniques for avoiding renaming and optimizing unification in bottom-up evaluation, presented in [SR93], are implemented in CORAL.

7.1.4 Multisets

CORAL allows multiset-terms (and, as a special case, set-terms). Multisets are represented using the same data structures as relations, details of which are presented in Section 7.2. As with strings and functor terms, CORAL creates unique identifiers for the unification of multiset-terms; as with functor terms, this is done in a lazy fashion.

As noted earlier, the generation of unique identifiers for multisets is completely orthogonal to the generation of unique identifiers for functor terms, and complex terms built using both type constructors can be assigned unique identifiers without any problems.

7.2 Representation of Relations and Index Structures

Relations and indices are implemented as C++ classes that are sub-classes of `Arg`, and thus can be used just like other terms. However we have chosen to discuss them separately since the interface between the query evaluation system and the data manager is particularly important when it involves relations and indices.

7.2.1 Relations

CORAL currently supports in-memory hash-relations, as well as disk-resident relations (the latter by using the EXODUS storage manager [CDRS86]). Multiple indices can be created on relations, and can be added to existing relations. The relation interface is designed to make the addition of new relation implementations (as sub-classes of the generic class Relation) relatively easy.

CORAL relations (currently only the in-memory versions) support several features that are not provided by typical database systems. The first and most important extension is the ability to obtain *marks*[22] into a relation, and distinguish between facts inserted after a mark was obtained and facts inserted before the mark was obtained. This feature is important for the implementation of all variants of semi-naive evaluation described in Section 6.3. The implementation of this extension involves creating subsidiary relations, one corresponding to each interval between marks, and transparently providing the union of the subsidiary relations corresponding to the desired range of marks. A benefit of this organization is that it does not interfere with the indexing mechanisms used for the relation (the indexing mechanisms are used on each subsidiary relation).

CORAL can also "hide" facts in a relation and make them visible in a "prioritized" fashion. Such a feature is important for efficient evaluation of some programs, as discussed in Section 6.6.5.

7.2.2 Index Structures

Hash-based indices for in-memory relations and B-tree indices for disk-resident relations are currently available in the CORAL system. New index implementations can be created as sub-classes of the generic class Index and may be added to any relation implementation. For instance, pattern-form indices were added using this interface.

CORAL provides a uniform interface for the addition of any kind of index to a relation; tuples already in the relation are automatically added to the index. Scan operations on a relation may optionally specify an index to be used for the scan.[23] CORAL also has a mechanism for automatic selection of an index for a scan, in case an index is not specified.

7.3 Persistent Relations

CORAL uses the EXODUS storage manager to support persistent (or, disk-resident) relations. The schema of a CORAL disk-resident relation must be declared. For example, the schema of the employee relation (from Section 2.1) can be declared as $schema(employee(string, string, float))$. In the current CORAL implementation, tuples in a disk-resident relation are restricted to have fields of primitive types only. One way of allowing objects of arbitrary classes in such tuples would be to require the classes to provide member functions for encoding objects as bit-strings, and corresponding decoding functions. Future releases of CORAL may provide this facility.

Indices can be maintained, and are implemented as B+ trees. Both base relations as well as derived relations can be disk-resident. Derived relations that are materialized on disk during the evaluation of a query reside on a temporary disk volume so that they do not persist after the completion of the query. This is in contrast to persistent relations that reside on disk across query invocations and across invocations of CORAL.

The transaction mechanism provided by EXODUS is used to provide transaction semantics for actions on 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 the client buffer pool, a request is forwarded to the EXODUS server and the page with the requested tuple is retrieved.

As an artifact of the basic implementation decision to share constants instead of copying their values, there is some inefficient copying of data that occurs while handling disk-resident data. We are in the process of modifying

[22] A *mark* is a mechanism that makes it possible to recognize which tuples in the relation were added after the mark was obtained, and which tuples were already in the relation before the mark was taken. If a relation is conceptually viewed as a chain of tuples with new tuples appended to the end of the chain, obtaining a mark corresponds to recording a pointer to the current end of the chain.

[23] The current version of CORAL does not support scan predicates such as $X < 10$, but this feature will be supported in a future release of CORAL.

the implementation, at least in the case of constants of primitive types like integers.

8 Interface with C++

The CORAL system has been integrated with C++ in order to support a combination of declarative and imperative programming styles. The CORAL system provides a collection of new C++ classes (relations, tuples, args and scan descriptors) and a suite of associated member functions. In addition, there is a construct to embed CORAL commands in C++ code. This extended C++ can be used in conjunction with the declarative language features of CORAL in two distinct ways:

- Relations can be computed in a declarative style using declarative modules, and then manipulated in imperative fashion in extended C++ without breaking the relation abstraction.

  In this mode of usage, typically there is a main program written in C++ that calls upon CORAL for the evaluation of some relations defined using CORAL modules. The main program is compiled (after some pre-processing) and executed from the operating system command prompt; the CORAL interactive interface is not used.

- New predicates can be defined using extended C++. These predicates can be used in declarative CORAL code and are incrementally loaded from the CORAL interactive command interface. There are, however, some restrictions on the types of arguments that can be passed to the newly defined predicates.

Thus, declarative CORAL code can call extended C++ code and vice-versa. The above two modes are further discussed in the following sections.

8.1 CORAL Classes Visible to C++

The C++ classes defined in the CORAL system that are visible to the C++ programmer include:

Relation : This allows access to relations from C++. Relation values can be constructed through a series of explicit inserts and deletes, or through a call to a declarative CORAL module. The associated member functions allow manipulation of relation values from C++ without breaking the relation abstraction.

Tuple : A relation is a collection — set or multiset — of tuples.

Arg : A tuple, in turn, is a list of args (i.e., arguments). A number of functions are provided to construct and take apart arguments and argument lists.

C_ScanDesc : This abstraction supports relational scans in C++ code. A C_ScanDesc object is essentially a cursor over a relation.

The following example illustrates several classes (`Relation, C_ScanDesc, Tuple and Arg`) and functions which form part of the CORAL interface to C++:

```
int sum_first_args(char* rel_name, int rel_arity)
{
  Relation *rel = find_relation(rel_name, rel_arity);
  C_ScanDesc *scan = new C_ScanDesc(rel);

  Tuple *tuple; int sum = 0;

  /* Iterate over the tuples in the relation */
  for (tuple = scan->next_tuple(); !(scan->no_match()); tuple = scan->next_tuple()) {
```

```
    if (!is_int((*tuple)[0])) {
      error("non-integer first field !");    /* Print error message */
      exit 1;
    }
    sum += make_int((*tuple)[0]);   /* Sum up the first argument of each fact */
  }
  return (sum);
}
```

This example uses functions like `find_relation` and `is_int` which are part of the interface specification. The complete interface specification is provided in the user manual [RSSS93a]. However, this simple program demonstrates the fact that the C_ScanDesc abstraction, along with the Relation, Tuple and Arg abstractions, gives the C++ programmer a convenient way of accessing data stored in CORAL relations. Scans can be set up in a totally identical fashion on both base and derived relations.[24] A suite of routines is provided for converting CORAL terms into C++ values and vice-versa.

One restriction in the current interface is that a very limited abstraction of variables is presented to the user. Variables can be used as selections for a query (say, via repeated variables) or in a scan, but variables cannot be returned as answers (i.e., the presence of non-ground terms is hidden at the interface). Presenting the abstraction of non-ground terms would require that binding environments be provided as a basic abstraction, and this would make the interface rather complex.

8.2 Calling CORAL from C++

Any sequence of commands that can be typed in at the CORAL interactive command interface can be embedded in C++ code. However, the code must be bracketed by special delimiters. A file containing C++ code with embedded CORAL code must first be passed through the CORAL pre-processor and then compiled. The following program illustrates how to call declarative CORAL from imperative C++:

```
main(int argc, char**argv)
{
  int i = 2; double j = 4.23;
  init_coral(argv[0]);

  for (i = 0; i < 3; i++) {
    \[
      grows(($int)$i, 1).
      fixed(2, ($double)$j).
      ?grows(X,Y).
      ?fixed(X,Y).
    \]
  }
  exit_coral();
}
```

During the execution of the above program, each time through the loop, the variable `i` which is passed to the declarative CORAL code takes on a new value, and hence new facts get added to the relation "`grows`" on each iteration. The query `?grows(X,Y)` prints out a set of answers on each iteration, and the set of answers increases on successive iterations. Assuming duplicate elimination is performed, the relation "`fixed`" remains unchanged, as also the set of answers to the query `?fixed(X,Y)`.

[24] Note that it is easy to materialize a derived relation, if desired, by using an imperative rule with ":=".

8.3 Defining New Predicates in C++

As we have already seen, predicates exported from one CORAL module can be used freely in other modules. Sometimes, it may be desirable to define a predicate using extended C++, rather than the declarative language supported within CORAL modules. A *_coral_export* statement is used to declare the arguments of the predicate being defined. The definition can use full extended C++. The source file is pre-processed into a C++ file, and compiled to produce a .o file. If this file was consulted from the CORAL prompt, then it is incrementally loaded into a newly allocated region in the data area of the executing CORAL system. It is also possible to directly consult a pre-processed .C file or .o file, and avoid repeating the pre-processing and compilation steps.

Consider the following example of defining CORAL predicates using C++.

```
_coral_export double myfunc(double);
double myfunc(double x)
{
  return x*2 ;
}
```

The export statement defines a CORAL predicate `myfunc` based on the C++ function `myfunc`. The return value of the C++ function *myfunc* is automatically mapped into the second argument of the CORAL predicate *myfunc*. This predicate must be called with the first argument bound to a double; the second argument can be free or bound. If the second argument is bound, the computed value is compared with the given binding.

The CORAL primitive types are the only types that can be used in a _coral_export declaration; user-defined types are not allowed. An alternative mechanism is available to define more complex predicates using C++; details may be found in the CORAL manual [RSSS93a]. However, the export mechanism makes it very easy to define simple predicates using C++.

9 Extensibility in CORAL

The implementation of the declarative language of CORAL is designed to be extensible. The user can define new abstract data types, new relation implementations and new indexing methods, and use the query evaluation system with no (or in a few cases, minor) changes. The user's program will, of course, have to be compiled and linked with the system code. CORAL assumes a set of standard operations on data types is available, and all abstract data types must provide these operations (as C++ virtual member functions).

9.1 Extensibility of Data Types

The type system in CORAL is designed to be extensible; the class mechanism and virtual member functions provided by C++ help make extensibility clean and local. "Locality" refers to the ability to extend the type system by adding new code, without modifying existing system code — the changes are thus local to the code that is added. All abstract data types should have certain virtual member functions defined in their interface, and all system code that manipulates objects operates only via this interface. This ensures that the query evaluation system does not need to be modified or recompiled when a new abstract data type is defined. The required member functions include the member function *equals* which is used to check if two objects are equal, the member function *print* for printing the object, *hash* to return a hash value, and constructor and destructor functions. For a summary of the virtual member functions that constitute the abstract data type interface, see [RSS92b, RSSS93a]. The user can define predicates (using C++) to manipulate (and possibly display in novel ways) objects belonging to the abstract data types. These predicates must be registered with the system and can then be accessed from CORAL; registration is accomplished by a single command.

While the creation of new abstract data types in CORAL is quite straightforward, the definition of predicates to manipulate the abstract data types is a little more complicated since the creator must have some knowledge of the internal data representation in CORAL. Given this knowledge, defining a predicate is easy as well. For example, one of the authors implemented an array abstract data type along with a set of predicates to manipulate it, with about half a day's effort.[25]

9.2 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. The "get-next-tuple" interface between the query evaluation system and a relation is the basis for adding new relation implementations and index implementations in a clean fashion. The implementation of persistent relations using EXODUS illustrates the utility of such extensibility (Section 7.3).

10 Performance Results

The wide range of evaluation techniques available in CORAL and the ability to choose a combination easily offers much greater flexibility than is available in other systems. Here, we present some performance results that underscore the value of this flexibility by examining several programs and evaluating them using different execution alternatives in CORAL. For lack of space, we present only a few of the numbers that we generated, and just summarize our results here. We performed measurements on a lightly loaded Sparcstation 10/51, using Unix user cpu times. The programs chosen represent "typical" building blocks used in deductive applications: computing transitive closure and bill-of-materials relationships, appending lists, aggregating over sequences, and simple non-recursive rules that join several relations.

The purpose of providing these performance numbers is to give a feel for the efficiency of the system. We have not presented a comparison of CORAL with other systems. Such comparisons need to be carefully designed and are beyond the scope of this paper.

Ancestor and Bill of Materials

The first program that we examined was the well known *ancestor* example. We used a tree with a depth of 3 and a uniform fanout of 10 (having a total of 1110 edges), and a chain of length 160 as our data sets. The roots of the data sets were used as the selections on the first argument for each of the queries; thereby all nodes were present in the answers. The timings are shown in Table 1, and they do not include printing times. We present figures for three ways of writing the program (left-linear, right-linear and non-linear) in combination with two optimization techniques (Supplementary Magic, which is used by default in CORAL, and Factoring, which is applicable only to a certain class of programs and must be explicitly enabled).

The factoring rewriting of all the above forms of the ancestor program results in a program similar to (but slightly simpler than) the left-linear Supplementary Magic version of the program; hence their timings are similar. The poor performance of the non-linear version of the program with Supplementary Magic rewriting indicates that thought must be given to efficiency when writing declarative programs, unless the optimizer is sufficiently smart.

We also ran a bill-of-materials program, which computed all the basic parts needed to create an assembly, and summed up the total cost of the copies of each basic part required. We used a synthetic database of 100 different basic parts, and 15000 facts for the assemblies (each assembly consisting of three sub-assemblies/basic parts). A query on this database which resulted in all 100 basic parts being accessed ran in 12.36 seconds.

[25] This is available as part of the CORAL system. The additional code involved is included and discussed in the CORAL user manual [RSSS93a].

Program	Dataset	Rewriting	Timing
Left Linear Ancestor	Tree1110	Supplementary Magic	0.50s
		Factoring	0.40s
	Chain160	Supplementary Magic	0.09s
		Factoring	0.07s
Right Linear Ancestor	Tree1000	Supplementary Magic	1.65s
		Factoring	0.77s
	Chain160	Supplementary Magic	4.88s
		Factoring	0.11s
Non Linear Ancestor	Tree1000	Supplementary Magic	4.00s
		Factoring	0.74s
	Chain160	Supplementary Magic	149.25s
		Factoring	0.14s

Table 1: Ancestor Timings

List Length	Supp. Magic	Last-Call	Pipelined
100	0.19s	0.13	0.03
200	0.36s	0.27	0.06
400	0.71s	0.53s	0.14s

Table 2: Append Timings

Appending Lists

The second program that we examined was the standard program for appending two lists; the program was used primarily to demonstrate CORAL's support for complex data such as lists. We performed timing measurements for appending ground lists of different sizes; three evaluation techniques were used — pipelining, materialization using Supplementary Magic rewriting, and materialization using a version of Supplementary Magic rewriting with last-call optimization [Ros91]. Factoring is not applicable for this program,

The timings are shown in Table 2. As the data indicates, append runs in linear time on CORAL with all three evaluation techniques. What is interesting is that the rewritten program generated by the version of Supplementary Magic with last-call optimization actually generates non-ground facts, even though the append program itself does not generate non-ground facts on the queries we use. Without the non-ground fact related optimizations [SR93, Sud92], the evaluation of the above rewritten program would have taken quadratic time. With the non-ground fact optimization, not only did the program run in linear time, but it ran faster than the version of the program rewritten using Supplementary Magic rewriting. The version of Supplementary Magic with last-call optimization has not yet been implemented in the CORAL system, but we hand-coded the program to demonstrate the benefits of the non-ground fact optimizations implemented in CORAL, and the benefits of last-call optimization.

The timings for the append program clearly indicate that pipelining is the best evaluation mechanism for append; however, the exercise demonstrates that programs which need to be evaluated with materialization (for other reasons), can perform list operations with acceptable asymptotic bounds.

Moving Average Over a Sequence

The third program we studied computed the N-day moving average over a daily sequence of stock quotes. The average of the values of a sequence over a "window" of days prior to the current day are computed for each day in the sequence. With Supplementary Magic, on a sequence of length 1000, CORAL took about 0.74s regardless of window size. This reflects the fact that the optimized program essentially just scans the relation once, independent of the window size. Performance is linear in the size of the input sequence; e.g., CORAL takes 1.55 seconds on a sequence of length 2000, and 3.37 seconds on a sequence of length 4000,

Indexing and Joins

We measured the time taken to build indices on relations. For in-memory relations, indices could be constructed very efficiently; for example, creating an index on the first two columns of a ternary relation with 15000 tuples took just 0.24 seconds. This bears out our decision to construct indices on in-memory relations whenever the indices are required for evaluating a rule.

To get a feel for the performance of low-level evaluation mechanisms, we performed a simple join of the form "parent(X,Y), parent(X,Z), parent(Z,W)," with the tree data set having 1110 edges used as the input. We timed two variants of the program, both of which had an empty relation as the fourth relation in the join, so no actual facts were generated (the cost of materializing facts dominates the other join costs). The first variant had a relation with no arguments, and intelligent backtracking on finding the relation was empty the first time it was accessed would recognize that the rule would never generate any facts, and not perform the rest of the join. The time for detecting this was just around 0.05 seconds. The second variant was crafted to foil intelligent backtracking. Since the fanout of the tree data set is 10, the literal placed after the three parent literals is reached about 100,000 times in a nested-loops join. This variant of the join took 1.25 seconds to execute.

When the parent facts were stored in an EXODUS (persistent) relation, all the timings were approximately five times higher. This is primarily because of inefficiencies in the interface between CORAL and EXODUS, and the system is currently being modified to reduce some of the unnecessary overheads in this interface.

Based on our experience thus far, we can make a few observations: CORAL is not very efficient in its low-level implementation. For example, it is nearly three times slower than LDL on the basic join program, about eight to ten times slower than XSB, and about twenty-five times slower than WAM-based Prolog implementations. We believe that the difference in speed between LDL and CORAL on the basic join program is due to the fact that LDL is compiled into C, whereas CORAL is interpreted. However, CORAL performs better than LDL on many programs that are more complicated; we believe this is because, in general, CORAL has better program transformation and evaluation techniques. XSB and WAM-based Prolog implementations have paid a great deal of attention to low-level optimization techniques, such as unification and memory-management and this results in superior performance. However, these systems provide a fixed control strategy, and fixed data types, unlike CORAL which has a number of different control strategies and an extensible type system. We are currently studying how some of the WAM optimizations (see [AK91], for example) can be extended to support these CORAL features; This would require changes in, for example, internal representations of terms, memory management and code for rule evaluation. We believe that high-level optimizations in CORAL, such as the program transformations and semi-naive fixpoint evaluation, would not be substantially affected by such low-level optimizations.

11 Applications

It is widely accepted that developing significant applications currently represents the major challenge for deductive systems. We briefly describe major CORAL applications that we are aware of. In addition, we have developed and tested a large suite of programs; these are described in the user manual [RSSS93a]. The CORAL system has been retrieved by over 200 sites and has been used in research projects as well as in courses at several universities. Some substantial research applications of CORAL, developed by others, were described in the post-ILPS93 Workshop on Programming With Logic Databases [Ram93]. These include:

Support for visual querying of graph data : The Hy+/GraphLog group at the University of Toronto is using CORAL as a back-end for a system that allows users to pose queries through a visual interface [VW93]. A rich class of natural queries over graphs (e.g., flight queries, and class library dependency queries) is supported, and queries are evaluated by translation into CORAL queries.

Genome Sequence Analysis : A group at MIT's Genome Lab is considering the use of CORAL for maintaining and querying genome sequence data [GRS93]. They report that deductive databases are well-suited for their application since many of their queries are naturally recursive. They have used CORAL for posing some queries on their data, and mention that CORAL's features and performance in terms of speed are very good for their application.

In addition, several CORAL applications have been developed at the University of Wisconsin, which we now describe.

11.1 Applications Developed at the University of Wisconsin

The CORAL group has developed two substantial applications of CORAL.

The first application is the Mimsy package [RRS93] for analyzing sequence data in the stock market domain. Mimsy allows users to write queries using a menu-driven pseudo-English language interface (with no knowledge of CORAL). Mimsy queries are implemented by translating them into CORAL queries and passing them over a socket to a server executing CORAL. Data is stored in Unix files in the compressed format used by CRSP[26] for distribution. The relations needed to answer a query are loaded into memory on demand. Answers can be viewed as relations or as graphs; in addition to CORAL, software such as Ghostview and IPL is used in Mimsy.

An important feature of Mimsy is the special implementation of relations corresponding to stock histories — they are simply and efficiently implemented as C++ arrays, and CORAL's extensibility is used to "register" this new relation type. Adding such "array relations" was quite easy[27] and yielded a significant performance improvement, underscoring the importance of extensibility. We have tested the package on five years worth of data on 100 stocks; over 20 years worth of data on about 1600 stocks is available. It is therefore very important that the data be stored in compressed form. Nonetheless, an interesting point is that for virtually all queries of interest, all the data easily fits into memory — to see why this is so, note that the data for each stock can be represented in one double word, there are about 250 data items per stock per year, and that queries rarely involve over 10 stocks. Since data for a query is loaded on demand, the performance of a query is determined only by the data that it touches, rather than by the total amount of available data.

It is worth noting that many queries that Mimsy is designed to deal with are naturally recursive, and difficult to express in SQL. For example: "Find the N-day moving average of IBM" and "Find the longest increasing run in GM in 1993." Further, even queries that are non-recursive are more efficiently implemented than in a typical SQL system due to the light-weight implementation of relations containing stock series data. An example of a non-recursive query is: "Find the 4-day average of IBM whenever the price of DEC is up more than 5 per cent."

The Mimsy system is inspired by MIM [Log92, Lew92], which is a commercial package for analyzing stock market data. Mimsy has many of the features of MIM, which is a stand-alone package written in C, but Mimsy is not as fast as MIM. Nonetheless, it offers good interactive performance for typical queries. An important feature of Mimsy, not available in MIM, is the extensibility that comes from having CORAL available underneath. A sophisticated user can easily add new built-ins or predicate definitions using the CORAL deductive language and make these accessible to naive users from the pseudo-English language menu. This allows Mimsy to be customized with complex analysis strategies by individual users, which is a very desirable feature in an environment where many traders use sophisticated proprietary strategies for buying and selling securities.

A second application is an explanation and debugging package for CORAL called Explain [ARR$^+$93], which is itself implemented using CORAL. The implementation of Explain uses the C++ interface of CORAL extensively. However, the implementation of Explain does not utilize CORAL's deductive capabilities. Explain allows a user to browse through all derivation trees for facts generated during a CORAL run, using a graphical point-and-click interface. During a run, *derivation fragments* (rule instantiations) are generated and stored in a file. Subsequently, the browsing component of Explain can be invoked and the saved fragments are loaded into CORAL relations. The Explain commands to "grow" and "prune" derivation trees for "interesting" facts are

[26] Center for Research in Security Prices, University of Chicago.

[27] A graduate student who was moderately familiar with CORAL added and fully tested this code in under a week.

implemented using these relations, and the implementation is greatly simplified by leaving the management of these relations (including such concerns as indexing) to CORAL. Storing derivation fragments slows CORAL execution by a factor of about five, and loading in the saved relations can take tens of seconds for large dumps (some of our tests created over 60,000 fragments). Responses to browsing queries, once these relations are loaded, is very fast (much less than a second).

Also at the University of Wisconsin, Madison, Tom Reps has used CORAL to implement algorithms for *interprocedural slicing*, a technique for identifying program statements that can affect the value of a variable at a particular program point. An algorithm for slicing was reported in [HRB90], and implemented in about 5000 lines of C in eight months. The programs to be analyzed by slicing were encoded as a set of facts (corresponding to edges in the program dependency graph) in CORAL, and the algorithm for slicing was implemented in just about 100 lines of declarative CORAL code. A notable feature of the CORAL code was that complex recursive rules were frequently used, with the most common being variations of transitive closure. However, rules similar to the well-known same-generation rules (but more complex!) were also used; this is not surprising when one considers the need to "match" procedure calls and returns. This application clearly shows the utility of a declarative language capable of dealing with large numbers of facts; for example, one sample program of about 700 lines had an encoding consisting of over 10000 base facts. The number of recursive semi-naive rules in the largest recursive component was approximately 25, and tens of thousands of intermediate facts were generated. The application also brought out some limitations in the CORAL implementation. Memory management, join ordering and low-level rule implementation seem to be the main weaknesses, although a more careful evaluation remains to be done.

12 Related Systems

A number of other deductive database systems have been developed in the past few years. These include (in alphabetical order) Aditi [VRK+91], ConceptBase [JS93], EKS-V1 [VBKL90], Glue-NAIL! [MUVG86, PDR91], LDL [NT89, CGK+90], LDL++ [AO93], LOLA [FSS91], Starburst SQL [MPR90] and XSB [SSW93]. There are many similarities between CORAL and these systems. However, there are several important differences, and CORAL extends the above systems in the following ways:

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

EKS-V1 supports integrity constraint checking, hypothetical reasoning and provides some support for non-stratified aggregation [Lef91, Lef92]. ConceptBase supports several object-oriented features, integrity constraint checking, and provides a one-way interface to C/Prolog, i.e., the imperative language can call ConceptBase, but not vice versa. LOLA supports integrity constraints, several join strategies, and some support for type information. The host language of LOLA is Lisp, and it is linked to the TransBase relational database. Aditi gives primary importance to disk-resident data and supports several join strategies.

In CORAL, 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. Modules with run-time semantics are also available in several production rule systems (for example, RDL1 [KdMS90]).

LDL++ [AO93], a successor to LDL, has moved in the direction taken by CORAL in several respects. It is interpreted, supports abstract data types, and uses a local semantics for choice (Carlo Zaniolo, personal communication). XSB [SSW93] is a system being developed at SUNY, Stony Brook. It supports several features similar to CORAL, such as non-ground terms and modularly stratified negation. Program evaluation in XSB uses OLDTNF resolution, and has been implemented by modifying the WAM; this implementation performs basic operations such as unification very efficiently.

Unlike most logic programming systems, such as various implementations of Prolog, CORAL (like some of the other deductive database systems) supports declarative semantics for all positive Horn clause programs, and for a large class of programs with negation and aggregation as well, and provides better indexing facilities and support for persistent data.

13 Conclusions

13.1 Retrospective on the CORAL Design

The CORAL project is at a stage where one version of the system has been released in the public domain, and an enhanced version will soon be released. Looking back at the evolution of the system, the effects of several design decisions are becoming increasingly evident. On the positive side, most of the decisions we made seem to have paid off with respect to simplicity and ease of efficient implementation.

Modules : The concept of modules in CORAL was in many ways the key to the successful implementation of the system. Given the ambitious goal of combining many evaluation strategies controlled by user hints in an orthogonal fashion, the module mechanism appears to have been the ideal approach.

Annotations : It has been our experience in practice that often the discerning user is able to determine good control strategies that would be extremely difficult, if not impossible, for a system to do automatically. Hence the strategy of allowing the users to express control choices was a convenient approach to solving an otherwise difficult problem. It is important to emphasize that a good choice of default decisions is essential in order to shield a naive user from the need to learn about annotations. As the system becomes more sophisticated in making intelligent optimization choices, the need for user-specified annotations decreases. However, the ability to specify annotations when necessary is a valuable feature.

In retrospect, annotations such as "aggregate selections" have proven to be extremely useful, whereas other annotations such as "prioritize" have not been used much.

Extensibility : The decision to design an extensible system seems to have helped greatly in keeping our code clean and modular.

System Architecture : The architecture concentrated on the design of a single-user database system, leaving issues like transaction management, concurrency control and recovery to be handled by the EXODUS storage manager. Thus CORAL could build on facilities that were already available, and focus instead on the subtleties of deductive databases and logic rules. The overall architecture was reasonably successful in breaking the problem of query processing into relatively orthogonal tasks.

On the negative side, some poor decisions were made, and some issues were not addressed adequately.

Type Information : CORAL makes no effort to use type information in its processing. No type checking or inferencing is performed at compile-time, and errors due to type mismatches lead to subtle run-time errors. Typing is a desirable feature, especially if the language is to be used to develop large applications. This is one of the issues addressed by a proposed extension to CORAL [SRSS93].

Memory Management : In an effort to make the system as efficient as possible for main-memory operations, copying of data has largely been replaced by pointer sharing, even for primitive data types such as integers. While this does make evaluation more efficient, it requires extensive memory management and garbage collection. This is, in retrospect, the worst of our design decisions, and is currently being modified. It has implications for both garbage collection and the interface to persistent data.

Low-level Optimizations : The focus of the CORAL implementation was high-level optimizations such as rewriting algorithms and semi-naive evaluation. We have not attempted to fully optimize the basic operations in evaluating a rule. In the main-memory case, optimizations such as those pioneered in the Warren Abstract Machine (WAM) for Prolog systems could significantly improve performance. For disk-resident data, more efficient join methods and cost-based query optimization would greatly improve the system.

On the whole, however, CORAL has incorporated many features that are unique among deductive database systems.

1. As an architectural feature, the importance assigned to modules both as compile-time and run-time units distinguishes CORAL from other deductive database systems (as discussed in Section 12). The EXODUS-based architecture for persistent data and the C++ interface are also notable design decisions.

2. From a language viewpoint, CORAL supports a wide range of features that set it apart from other languages. In particular, the support for non-stratified negation and aggregation, non-ground facts, and the many control annotations make for a rich and powerful language.

3. From an implementation viewpoint, many of the strategies that are supported by CORAL were developed in the course of the project. These include program evaluation strategies like Magic Templates, Factoring, Ordered Search and Predicate Semi-Naive evaluation, as well as optimization techniques to handle non-ground facts efficiently, and techniques to implement Save Modules.

13.2 Future Directions

There are a number of directions in which CORAL could be, and in some cases needs to be, extended. This section discusses some issues that appear to be important, and that will involve significant effort. The issues discussed initially are those that involve enhancing the power or the performance of existing features of CORAL. Some other desirable enhancements that require extensive changes to the system are mentioned later.

- Further support of persistent data needs to be provided. Though CORAL uses the indexed nested-loops join strategy as its default, there is no reason to stick with this approach, especially for joins of persistent relations. The design of the system makes no assumption about individual join methods, and so persistent relations should be joined by the most efficient join method available. While the current system permits the storage of only primitive data types in EXODUS, this needs to be enhanced to allow the storage of structured data as well. This is an interesting direction of future research.

- The management of memory is probably the biggest drawback of the first version of the system. Extensive code modification is required to copy constants instead of sharing them. This process is partially completed at the time of publication.

- The interface with C++ needs to be enhanced with new abstractions that allow programmers greater abilities to use the power of CORAL. In particular, a more powerful abstraction of variables needs to be provided.

- While performance measurements of a preliminary nature have been made, an extensive performance evaluation of CORAL, both to evaluate various aspects of the system and to compare it with other systems needs to be performed.

- At present, the system presents a command prompt interactive interface, in much the same manner as most Prolog systems. There are many challenges with respect to user interfaces for declarative language systems that could be tackled in the CORAL context.

- Object-oriented features such as abstract data types, encapsulation and inheritance provide very rich data modeling capabilities. Incorporating such features into CORAL [SRSS93] would be crucial in developing large applications in CORAL. The key to this integration is the following observation: object-oriented features are essentially extensions of the data model, and a clean integration of these features into CORAL can be achieved by allowing the language to draw values from a richer set of domains.

- Currently, CORAL provides support for non-ground facts with universally quantified variables. In several applications, variables in facts are typically constrained by information from some domain. An important extension to CORAL is to support such constraint facts [KKR90], and we are considering linear arithmetic constraints as one of the constraint domains to incorporate into CORAL.

Acknowledgements

We would like to acknowledge our debt to Aditi, EKS-V1, LDL, NAIL!, SQL, Starburst, and various implementations of Prolog from which we have borrowed numerous ideas. We would like to acknowledge the contributions of the following people to the CORAL system. Per Bothner, who was largely responsible for the initial implementation of CORAL that served as the basis for subsequent development, was a major early contributor. Joseph Albert worked on many aspects of the set-manipulation code; Tarun Arora implemented several utilities and built-in/library routines in CORAL; Tom Ball implemented an early prototype of the semi-naive evaluation system; Jeff Cai provided developmental support; Lai-chong Chan did the initial implementation of existential query optimization; Manuvir Das implemented the Ingres interface; Sumeer Goyal implemented embedded CORAL constructs in C++; Vish Karra implemented pipelining; Robert Netzer did the initial implementation of Magic rewriting; and Bill Roth implemented the Mimsy system and helped build the Explain system.

References

[AK91] Hassan Ait-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction.* MIT Press, Cambridge, Massachusetts, 1991.

[AO93] Natraj Arni and Kayliang Ong. *The LDL++ User's Guide*, 2.0 edition, January 1993. MCC Technical Report Carnot-012-93(P).

[ARR+93] Tarun Arora, Raghu Ramakrishnan, William G. Roth, Praveen Seshadri, and Divesh Srivastava. Explaining program evaluation in deductive systems. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, 1993.

[Ban85] Francois Bancilhon. Naive evaluation of recursively defined relations. In Brodie and Mylopoulos, editors, *On Knowledge Base Management Systems — Integrating Database and AI Systems.* Springer-Verlag, 1985.

[Bay85] R. Bayer. Query evaluation and recursion in deductive database systems. Unpublished Memorandum, 1985.

[BNST91] Catriel Beeri, Shamim Naqvi, Oded Shmueli, and Shalom Tsur. Set constructors in a logic database language. *Journal of Logic Programming*, 10(3&4):181–232, 1991.

[BR87a] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3), September 1987.

[BR87b] Catriel Beeri and Raghu Ramakrishnan. On the power of Magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[Bry89] Francois Bry. Logic programming as constructivism: A formalization and its application to databases. In *Proceedings of the ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 34–50, Philadelphia, Pennsylvania, March 1989.

[CD85] J.H. Chang and A. M. Despain. Semi-intelligent backtracking of Prolog based on static data-dependency analysis. In *Proc. Symposium on Logic Programming*, pages 10–21, 1985.

[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.

[CGK89] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Abstract machine for LDL. Technical Report ACT-ST-268-89, MCC, Austin, TX, 1989.

[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.

[Der93] Marcia A. Derr. Adaptive optimization in a deductive database system. In Bharat Bhargava, Timothy Finin, and Yelena Yesha, editors, *Proceedings of the Second International Conference on Information and Knowledge Management*, pages 206–215, Arlington, VA, November 1993.

[FSS91] B. Freitag, H. Schűtz, and G. Specht. LOLA: A logic language for deductive databases and its implementation. In *Proceedings of 2nd International Symposium on Database Systems for Advanced Applications (DASFAA)*, 1991.

[Got74] E. Goto. Monocopy and associative algorithms in an extended lisp. Technical Report 74-03, Information Science Laboratory, Univ. of Tokyo, Tokyo, Japan, May 1974.

[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.

[GRS93] Nat Goodman, Steve Rozen, and Lincoln Stein. Requirements for a deductive query language in MapBase genome-mapping database. In Raghu Ramakrishnan, editor, *Proceedings of the Workshop on Programming with Logic Databases*, 1993.

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

[HRB90] S. Horowitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[JS93] M. Jeusfeld and M. Staudt. Query optimization in deductive object bases. In J.C. Freytag, G. Vossen, and D. Maier, editors, *Query Processing for Advanced Database Applications*. Morgan-Kaufmann, 1993.

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

[KKR90] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, April 1990.

[KRS90] D. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, left-, and multi-linear rule transformations that maintain context information. In *Proceedings of the International Conference on Very Large Databases*, pages 380–391, Brisbane, Australia, 1990.

[Lef91] Alexandre Lefebvre. Recursive aggregates in the EKS-V1 system. Technical Report TR KB 34, ECRC, February 1991.

[Lef92] Alexandre Lefebvre. Towards an efficient evaluation of recursive aggregates in deductive databases. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, June 1992.

[Lew92] Peter H. Lewis. A fast way to discover patterns in vast amounts of data. *The New York Times*, pages 16–17, August 1992.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[Log92] Logical Information Machines, 8920 Business Park Drive, Suite 372 Austin Texas 78759. *The XMIM Reference Guide*, 2.1.1 edition, July 1992. XMIM, Logical Information Machines and MIM are registered trademarks of Logical Information Machines, Inc.

[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.

[MR89] Michael J. Maher and Raghu Ramakrishnan. Dèjá vu in fixpoints of logic programs. In *Proceedings of the Symposium on Logic Programming*, Cleveland, Ohio, 1989.

[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.

[NRSU89] Jeffrey F. Naughton, Raghu Ramakrishnan, Yehoshua Sagiv, and Jeffrey D. Ullman. Argument reduction through factoring. In *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 173–182, Amsterdam, The Netherlands, August 1989.

[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 Conference on Management of Data*, pages 308–317, 1991.

[Prz88] T.C. Przymusinski. On the declarative semantics of stratified deductive databases. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216, 1988.

[Ram88] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.

[Ram93] Raghu Ramakrishnan, editor. *Proceedings of the Workshop on Programming with Logic Databases*, Vancouver, BC, Canada, October 1993. Available as Technical Report 1183, Computer Sciences Department, University of Wisconsin–Madison.

[RBK88] Raghu Ramakrishnan, Catriel Beeri, and Ravi Krishnamurthy. Optimizing existential Datalog queries. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 89–102, Austin, Texas, March 1988.

[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.

[RLK86] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The Alexander method — a technique for the processing of recursive axioms in deductive database queries. *New Generation Computing*, 4:522–528, 1986.

[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.

[Ros91] Kenneth Ross. Modular acyclicity and tail recursion in logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1991.

[RRS93] William G. Roth, Raghu Ramakrishnan, and Praveen Seshadri. MIMSY: A system for analyzing time series data in the stock market domain. In Raghu Ramakrishnan, editor, *Proceedings of the Workshop on Programming with Logic Databases*, 1993.

[RS91] Raghu Ramakrishnan and S. Sudarshan. Top-Down vs. Bottom-Up Revisited. In *Proceedings of the International Logic Programming Symposium*, 1991.

[RSS92a] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Controlling the search in bottom-up evaluation. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, 1992.

[RSS92b] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

[RSS93] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. The Save Module facility in CORAL. Manuscript, 1993.

[RSS94] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. *IEEE Transactions on Knowledge and Data Engineering*, August 1994. To appear. (A shorter version appeared in VLDB, 1990).

[RSSS93a] Raghu Ramakrishnan, Praveen Seshadri, Divesh Srivastava, and S. Sudarshan. The CORAL user manual: A tutorial introduction to CORAL. Manuscript, 1993.

[RSSS93b] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. Implementation of the CORAL deductive database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1993.

[Sek89] H. Seki. On the power of Alexander templates. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 150–159, 1989.

[SKGB87] H. Schmidt, W. Kiessling, U. Güntzer, and R. Bayer. Compiling exploratory and goal-directed deduction into sloppy delta iteration. In *IEEE International Symposium on Logic Programming*, pages 234–243, 1987.

[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.

[SR93] S. Sudarshan and Raghu Ramakrishnan. Optimizations of bottom-up evaluation with non-ground terms. In *Proceedings of the International Logic Programming Symposium*, 1993.

[SRSS93] Divesh Srivastava, Raghu Ramakrishnan, Praveen Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Proceedings of the International Conference on Very Large Databases*, Dublin, Ireland, 1993.

[SSW93] Konstantinos Sagonas, Terrance Swift, and David S. Warren. XSB: An overview of its use and implementation. Technical report, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York, 11794-4400, Oct 1993. Submitted for publication; available by anonymous FTP from cs.sunysb.edu in pub/TechReports/warren.

[Str91] Bjarne Stroustrup. *The C++ Programming Language (2nd Edition)*. Addison-Wesley, Reading, Massachusetts, 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.

[Sud92] S. Sudarshan. *Optimizing Bottom-Up Query Evaluation for Deductive Databases*. PhD thesis, University of Wisconsin, Madison, August 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+91] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, and Peter J. Stuckey. Design overview of the Aditi deductive database system. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 240–247, April 1991.

[VW93] Dimitra Vista and Peter Wood. Efficient visual queries for deductive databases. In Raghu Ramakrishnan, editor, *Proceedings of the Workshop on Programming with Logic Databases*, 1993.