Implementation of the CORAL Deductive Database System*

Raghu Ramakrishnan
Univ. of Wisconsin, Madison

Divesh Srivastava
Univ. of Wisconsin, Madison

S. Sudarshan †
AT&T Bell Labs, Murray Hill.

Praveen Seshadri
Univ. of Wisconsin, Madison

Abstract

CORAL is a deductive database system that provides a modular, declarative query language/programming language. CORAL is a deductive system that supports a rich declarative language, provides a wide range of evaluation methods, and allows a combination of declarative and imperative programming. The data can be persistent on disk or can reside in main-memory. We describe the architecture and implementation of CORAL.

There were two important goals in the design of the CORAL architecture: (1) to integrate the different optimization techniques in a reasonable fashion, and (2) to allow users to influence the evaluation strategies used so as to exploit the full power of the CORAL implementation. A CORAL declarative program can be organized as a collection of interacting modules and this module structure is the key to satisfying both these goals. The high level module interface allows modules with different evaluation techniques to interact in a transparent fashion. Further, users can optionally tailor the execution of a program by selecting from among a wide range of control choices at the level of each module.

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.

*This research was supported by a David and Lucile Packard Foundation Fellowship in Science and engineering, a Presidential Young Investigator Award, with matching grants from Digital Equipment
Corporation, Tandem and Xerox, and NSF grant IRI-9011563. The address 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,divesh,praveen}@cs.wisc.edu and sudarsha@research.att.com.

†The work of this author was performed largely while he was in the University of Wisconsin, Madison.

0

1 Introduction

In this paper, we discuss the design and implementation of the CORAL deductive database system. CORAL seeks to combine features of a database query language, such as efficient treatment of large relations, aggregate operations and declarative semantics, with those of a logic programming language, such as more powerful inference capabilities and support for incomplete and structured data. Support for persistent relations is provided by interfacing with the EXODUS storage manager [4]. A unique feature of CORAL is that it provides a wide range of evaluation strategies (such as top-down, bottom-up, and their variants) and allows users to optionally tailor execution of a program through high-level annotations. The language is described in [24]. Applications in which large amounts of data must be extensively analyzed are likely to benefit from this combination of features. In comparison to other deductive database systems such as Aditi [31], EKS-V1 [32], LDL [30], LOLA [6] and Nail-Glue [13], CORAL provides a more powerful language and supports a much wider range of optimization techniques.

We highlight several design decisions that allowed us to integrate (often unrelated) evaluation techniques and optimizations in a nearly seamless fashion. Specifically, we consider the following issues:

1. Data representation (e.g. constants, lists, terms).
2. Relation representation and implementation (e.g. in main-memory and disk-resident).
3. Index structures (e.g. hash-structures and B-trees).
4. Evaluation techniques (e.g. materialization and top-down)

In the CORAL implementation, we divide evaluation into a number of distinct subtasks, and provide a clean interface between the subtasks; relevant optimization techniques can be (almost) independently applied to each subtask. Extending database programming languages has received much attention lately, and we believe that the CORAL experience offers guidelines for resolving several common issues that go beyond the specific extensions that are addressed in

CORAL.

One of our goals was to allow users to exploit the full power of the implementation. CORAL supports a very rich language, and we believe that some user guidance is critical to effectively optimizing many sophisticated programs. The problem is to provide users with the ability to choose from the suite of optimizations supported by CORAL in a relatively orthogonal and high-level way, and to use a combination of optimizations for different parts of a program. The *module* structure was a key to solving this problem. The interface between modules is kept at a high level; evaluation techniques can be chosen on a per-module basis through (optional) *annotations*, and modules with different evaluation techniques can interact in a nearly transparent fashion.

An overview of the CORAL declarative language is presented in [24]. The query language supports general Horn clauses with complex terms, set-grouping, aggregate operations, negation and data that contains universally quantified variables. A number of user annotations can also be specified to guide the system in query evaluation and optimization. The details of the language are beyond the scope of the paper. Many features of the implementation ranging from low-level structures to the interactive system environment have also been omitted from the paper, due to shortage of space. The implementation details are found in [?]. In Sections 2- 5, we discuss some important aspects of the system implementation. Section 2 contains an overview of the CORAL system architecture. Section 3 explains the underlying representation of the data and Section 4 provides an overview of query evaluation and optimization. Section 5 is the main section that deals with implementation issues. It covers the basic strategies used in evaluating a module, as well as several important refinements. It also addresses user guidance of query optimization, and the interaction in the evaluation of different modules. The CORAL/C++ interface and support for extensibility in CORAL, including the addition of new data types and operations, and new relation and index implementations, are discussed in Sections 6 and 7. We discuss related systems in Section 8. Finally, we provide a retrospective discussion of the CORAL design and outline future research directions in Section 9.

2 Architecture of the CORAL System

The architecture of the CORAL deductive system is shown in Figure 1. CORAL is designed primarily as a single user database system, and can be used in a stand-alone mode. Persistent data is stored either in text files, or using the EXODUS storage manager [4], which has a client-server architecture. Each CORAL single-user process is a 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 toolkit, 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. Most of the effort of design and implementation in CORAL has concentrated on the single-user client, and the implementation has focused on operation out of main memory. While this is in keeping with Prolog-style systems, *there is no part of the design that is particularly biased towards a main-memory system*[1].

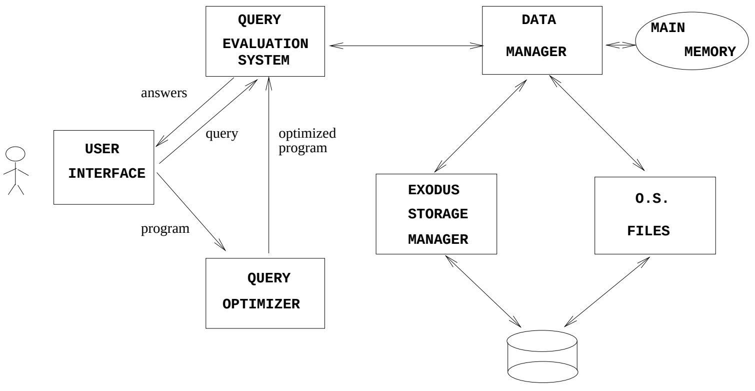

Figure 1: CORAL System Architecture

Data stored in text files can be 'consulted', at which point the data is converted into main-memory relations, with any specified indices; indices can also be created at a later time. 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 base relation, for instance) can be typed in at the user interface, and are not optimized. 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 doing rewriting transformations, 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.

[1]Certain optimizations like structure-sharing have however been implemented that optimize main-memory operation.

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 we generated a C++ program from each user program. (This is the approach taken by LDL [15, 5].) We found that this approach took a significantly longer time to compile programs, and the resulting gain in execution speed was minimal. We have therefore focused on the interpreted version; 'consulting' a program 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- or user-defined C++ code). 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 CORAL *does* manipulate data in a set-oriented fashion, and the 'get-next-tuple' interface is merely an abstraction provided to support modularity in the language. For example, a 'get-next-tuple' request on a persistent relation results in a page-level I/O request by the buffer manager of EXODUS.

CORAL supports an interface to C++, extended with several features that provide the abstraction of relations and tuples. C++ can be used to define new relations as well as manipulate relations computed using embedded declarative CORAL rules. The CORAL-C++ interface is intended to be used for the development of large applications.

3 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 straight-forward, 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, and this facilitates the discussion of evaluation techniques in the subsequent sections.

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 abstract 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, strings, or other abstract data-types are subclasses of Arg. The class Arg defines a set of virtual methodssuch as equals, hash, and print, which must be defined for each abstract data-type that is created. The class Tuple defines tuples of Args. A member of the class Relation is a set of tuples. The class Relation has a number of virtual methods 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[2]. The iterator is implemented using a member 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.

3.1 Representation of Terms

The primitive data types provided in the CORAL system include integers, doubles, strings, and arbitrary precision integers[3]. The current implementation restricts data that is stored using the EXODUS storage manager to be limited to terms of these primitive types. Such data is stored on disk in its machine representation, while in memory, the data types are implemented as subclasses of Arg.

The evaluation of rules in CORAL is based on the operation of *unification* that generates bindings for variables based on patterns in the rules 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 type can define how it generates unique identifiers, independent of how other types 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 types. This is very important for supporting extensibility and the creation of new abstract data types.

Terms can be built from a function symbol, or *functor*, and such terms are important for representing structured information. For instance, lists are a special type of functor term. A term $f(X, 10, Y)$ is represented by a record containing (1) the function symbol f, (2) an array of arguments, and (3) extra information to make unification of such terms efficient. The current implementation of CORAL uses a modified version of hash-consing [7, 5] that operates in a lazy fashion. 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, and these have to be handled differently.

Variables constitute a primitive type in CORAL, since

[2]This is analogous to the cursor notion in SQL.
[3]Arbitrary precision integers are supported using the BigNum package provided by DEC France.

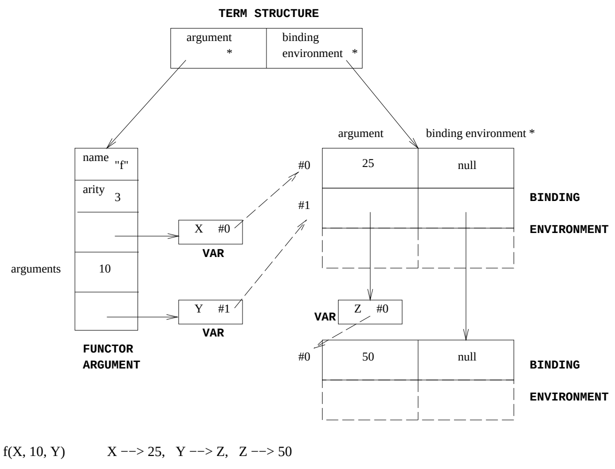

Figure 2: Representation of an Example Term

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 2.

3.2 Representation of Relations

CORAL currently supports in-memory hash-relations, as well as persistent relations (the latter by using the EXODUS storage manager [4]). 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 subclasses 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 get *marks* 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 5.3. 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 uses the EXODUS storage manager to support persistent relations. EXODUS uses a client-server architecture; CORAL is the client process, and maintains buffers for persistent relations. If a requested tuple is not in the client buffer pool, a request is forwarded to the EXODUS server and the page with the requested tuple is retrieved. While the architectural design does not require any copying of this data into other CORAL structures, the current implementation does perform some copying. This is an artifact of the basic implementation decision to share constants instead of copying their values. Currently, tuples in a persistent relation are restricted to have fields of primitive types only. At least in the case of constants of primitive types like integers, this has proven to be a poor decision, and we are in the process of modifying the implementation.

3.3 Implementation of Index Structures

Hash-based indices for in-memory relations and B-tree indices for persistent relations are currently available in the CORAL system. 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 a variable are hashed to a special value, denoted as `var`. The second form is more sophisticated, and allows us to retrieve precisely those facts that match a specified pattern, where the pattern can contain variables. Such indices are of great use when dealing with complex objects created using functors. One can retrieve, for example, those tuples in relation *append* that have as the first argument a list that matches $[X|[1, 2, 3]]$. A tuple $([5|[1, 2, 3]], [4], [5, 1, 2, 3, 4])$ would then be retrieved [26].

4 Overview of Query Evaluation

A number of query evaluation strategies have been developed for deductive databases, and each technique is particularly efficient for some classes of programs, but may perform relatively poorly on others. 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. 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 [24]; 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 5. 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.

4.1 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. Bindings are propagated by creating 'magic' facts that represent the appropriate binding values. 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.

The default rewriting technique is Supplementary Magic Templates [3, 18] ; see also [27, 28]. The rewriting can be tailored to propagate bindings across subgoals in a rule body using different subgoal 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 [18], Supplementary Magic With GoalId Indexing [26], and Context Factoring [16, 9]. Supplementary Magic is a good choice as a default, although each technique is superior to the rest for some programs. CORAL also supports Existential Query Rewriting [19], which seeks to propagate projections. This is applied by default in conjunction with a selection-pushing rewriting.

4.2 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), 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 whether to refine the basic nested-loops join with *intelligent backtracking*. These aspects are discussed in detail in [?].

The optimizer also decides on the subsumption checks to be carried out on each relation. The default is to do subsumption checks on all relations. A user can ask that a relation be treated as a *multiset*, with as many copies of a tuple as there are derivations for it in the original program[4]. This semantics is supported by carrying out duplicate checks only on the 'magic' predicates; some version of Magic Templates must used.

5 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. Modules *export* the predicates that they define; a predicate exported from one module is visible to all other modules, and can be used by them in rules. Since different modules may have widely varying evaluation strategies, some relatively high level interface is required for interaction between modules. The basic approach used by CORAL is outlined here.

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

Two basic evaluation approaches are supported, namely *pipelining* and *materialization*. Pipelining uses facts 'on-the-fly' and does not store them, at the potential cost of recomputation. Materialization stores facts and looks them up to avoid recomputation. Several variants of materialized evaluation are supported: Basic Semi-Naive, Predicate Semi-Naive [22], and Ordered Search [23].

5.1 Module and Rule Data Structures

The compilation of a materialized module generates an internal *module structure* that consists of a list of structures

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

corresponding to the strongly connected components (SCCs) of the module[5], and each SCC structure contains structures corresponding to semi-naive rewritten versions of rules. These *semi-naive rule structures* have fields that specify the argument lists of each body literal, and the predicates that they correspond to. Each semi-naive rule also contains evaluation order information, pre-computed backtrack points, and precomputed offsets into a table of relations.

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.

5.2 Pipelining

For *pipelining*, which is essentially top-down evaluation, the rule evaluation code is designed to work in a co-routining fashion — when rule evaluation is invoked, using the *get-next-tuple* interface, it generates an answer (if there is one) and transfers control back to the consumer of answers (the caller). Control is transferred back to the (suspended) rule evaluation when more answers are desired.

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 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.

There are three important points to note. Firstly, the implementation of pipelining, which is a radically different evaluation technique from bottom-up fixpoint evaluation, demonstrates the modularity of the CORAL implementation. Secondly, 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.) Thirdly, 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.

[5]An SCC is a maximal set of mutually recursive predicates.

5.3 Materialization

The variants of *materialization* are all bottom-up fixpoint evaluation methods. Bottom-up evaluation iterates on a set of rules, repeatedly evaluating them until a fixpoint is reached. In order to perform incremental evaluation of rules across multiple iterations, CORAL uses the semi-naive evaluation technique [2, 1, 22]. 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 in relations since the last iteration.) The evaluation part evaluates each rewritten rule once in each iteration, and performs some updates to the delta relations at the end of the iteration. An evaluation terminates when an iteration produces no new facts.

The optimizer analyzes the semi-naive rewritten rules and generates annotations to create any indexes that may be useful during the evaluation phase[6]. The basic join mechanism in CORAL is nested-loops with indexing. 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 nested-loops join considers the next tuple in any loop.

5.4 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 describe the implementation of some of these strategies.

5.4.1 Ordered Search

Ordered Search is an evaluation mechanism that orders the use of generated subgoals in a program and thereby provides an important strategy for handling programs with negation, set-grouping and aggregation, that are left-to-right modularly stratified. Full details of Ordered Search are not presented here, but the reader is referred to [23]. The principle of Ordered Search is that the computation is ordered by 'hiding' subgoals. This is achieved by maintaining a 'context' that stores subgoals in an ordered fashion, and that decides at each stage in the evaluation, which subgoal to make available for use next. The order in which generated subgoals are made available for use is somewhat similar to a top-down evaluation.

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 negated literals and rules that have grouping and aggregation. Second, the evaluation must add a goal ('magic' fact) to the corresponding 'done' predicate when (and only when) all answers to it have been generated. (The context mechanism is used

[6]Index generation also occurs for pipelined modules, but at the level of the original rules.

to determine the point at which a goal 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.

5.4.2 The Save Module Facility

In most cases, facts (other than answers to the query) computed during the evaluation of a module are best discarded to save space (since bottom-up evaluation stores many facts, space is generally at a premium). Module calls provide a convenient unit for discarding intermediate answers. By default, CORAL does precisely this — it discards all intermediate facts and subgoals computed by a module at the end of a call to the module. However, there are some cases where this leads to a significant amount of recomputation. This is especially so in cases where the same subgoal in a module is generated in many different invocations of the module. In such cases, the user can tell the CORAL system to maintain the state of the module (i.e., retain generated facts) in between calls to the module, and thereby avoid recomputation; we call this facility the *save module* facility.

In the interest of efficient implementation, we have the following restriction on the use of the save module feature: *if a module uses the save module feature, it should not be invoked recursively.* We do not make any guarantees about correct evaluation, should this happen at run-time. (Note that the predicates defined in the module can be recursive; this does not cause recursive invocations of the module). From an implementation point of view, the challenge is to ensure that no derivations are repeated *across multiple calls to the module.* This requires significant changes to semi-naive evaluation; while the details are omitted here for lack of space, they can be found in [25],

5.4.3 Lazy Evaluation

In the traditional approach to bottom-up evaluation, all answers to a query are computed by iterating over rules till a fixpoint is reached, and then returning all the answers. Lazy evaluation tries to return the answers at the end of every iteration, instead of at the end of computation. Lazy evaluation is implemented by storing the state of the computation at the end of an iteration, and returning the answer tuples generated in that iteration. The state is stored with the iterator that is created for the query (recall the 'get-next-tuple' iterative interface). The iterator then iterates over the tuples returned, and when it has stepped through all the tuples, it reactivates the 'frozen' computation that it has stored. 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.

5.5 Predicate Level Control

CORAL provides a variety of annotations at the level of individual predicates in a module. We discuss a couple of them in this section.

```
module s_p.
export s_p(bfff, ffff).
@aggregate_selection p(X, Y, P, C) (X, Y) min(C).
 s_p(X, Y, P, C)                  :- s_p_length(X, Y, C), p(X, Y, P, C
 s_p_length(X, Y, min(< C >))    :- p(X, Y, P, C).
 p(X, Y, P1, C1)                  :- p(X, Z, P, C), edge(Z, Y, EC),
                                     append([edge(Z, Y)], P, P1), C1
 p(X, Y, [edge(X, Y)], C)         :- edge(X, Y, C).
end_module.
```

Figure 3: Program Shortest_Path

5.5.1 Indexing Relations

As mentioned in Section 3.3, CORAL supports two forms of indices: (1) *argument form indices*, and (2) *pattern form indices.* The first form creates an index on a subset of the arguments of a relation. The second form is more sophisticated, and creates an index on a specified pattern that can contain variables. Suppose a relation *emp* had two arguments, the first a name and the second a complex term $addr(Street, City)$. The following declaration then creates a pattern form index that can efficiently retrieve employees named "John", who stay in "Madison", without knowing their street.

@ make_index $emp(Name, addr(Street, City))(Name, City)$.

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

5.5.2 Aggregate Selections

Consider the *shortest_path* program in Figure 3. 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 permits the user to specify an *aggregate selection* on the predicate *path* in the manner shown. The system then checks (at run-time) if a path fact is such that there is a path fact of lesser cost C with the same value for X, Y (i.e., between the same pair of points), and if there is such a fact, the costlier path fact is discarded. This aggregate selection is extremely important for efficiency — without it the program may run for ever, generating cyclic paths of increasing length. With this aggregate selection, along with the choice annotation @aggregate_selection $path(X, Y, P, C)(X, Y, C)any(P)$, a single source query on the program runs in time $O(E \cdot V)$, where there are E edge facts, and V nodes in the graph. CORAL's aggregate selection mechanism can also be used to provide a version of the choice operator of LDL, but with altogether different semantics [20].

5.6 Inter-Module Calls

The interaction between modules merits some discussion. Suppose that p is a predicate that appears in the body of a rule of module M2. Evaluation within a rule proceeds left-to-right[7] and can be thought of as a nested-loops join. (While this is not entirely accurate with respect to pipelined evaluation, it is an accurate enough description for our purposes.) When evaluation reaches the p literal, a scan is opened on p. A p tuple retrieved by the scan is used to instantiate the rule. When evaluation returns to the p literal on backtracking[8], the scan on p is advanced to get the next p tuple.

This 'get-next-tuple' interface to a relation p via a scan is the only interface presented to M2 by any relation, regardless of the nature of the relation. If p is defined in module M1 as a derived relation, the interface is still the same as if p were a base relation. We emphasize that the user need not be concerned about the details of how 'get-next-tuple' requests are generated. This is just an abstraction of how the evaluation proceeds, and is presented here for clarity of exposition.

An important consequence of this interface is that if M1 is materialized, then p is fully evaluated as evaluation repeatedly reaches the p literal upon backtracking. (More precisely, the part of p that is relevant to this p literal is fully evaluated.) Thus, the following rule governs inter-module calls:

> *The calling module will wait until the called module returns answers to the subquery. The called module presents a scan-like interface, and returns all answers to the subquery upon repeated 'get-next-tuple' requests.*

This is independent of the evaluation modes of the two modules involved. The point at which the called module returns answers, however, depends on its evaluation mode.

If the called module is pipelined, an answer is returned as soon as it is found, and the computation of the called module is suspended until another answer is requested by the caller. The use of certain features, such as 'save module' and 'aggregate selections' can result in all answers being computed before any answers are returned by the called module. Otherwise, answers are returned at the end of each fixpoint iteration in the called module; further iterations are carried out if more answers are requested by the calling module. At the level of the top-most query, this results in answers being available at the end of each iteration.

6 Interface with C++

The CORAL system has been integrated with C++ in order to support a combination of imperative and declarative programming styles. We have extended C++ by providing a collection of new classes (relations, tuples, args and scan descriptors) and a suite of associated methods. 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 in CORAL modules. The main program is compiled (after some preprocessing) 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 code can call extended C++ code and vice-versa. The above two modes are further discussed in the following sections.

6.1 Extensions to C++

C++ has been extended by adding a collection of classes and associated methods. The new classes are:

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 methods 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 methods 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.

In addition to the new classes, any sequence of commands that can be typed in at the CORAL interactive command interface can be embedded in C++ code, bracketed by special delimiters. A file containing C++ code with embedded CORAL code must first be passed through a CORAL preprocessor and then compiled using a standard C++ compiler. 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

[7] More generally, in a user specified order.

[8] Backtracking is only intra-rule unless evaluation in M2 is pipelined.

be provided as a basic abstraction, and this would make the interface rather complex.

6.2 Defining New Predicates

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 CORAL primitive types are the only types that can be used in a _coral_export declaration; user-defined types are not allowed. It is important to note that the CORAL preprocessor currently does *no* type checking, or even attempt to check if the exported function is defined in the file; it operates purely at a syntactic level.

The export mechanism makes it easy to pass values of these limited types between CORAL and C++ code. The predicate definition can use all features of 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 loaded into a newly allocated region in the data area of the executing CORAL system.[9] It is also possible to directly consult a pre-processed .C file or .o file, and avoid repeating the pre-processing and compilation steps.

7 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, or 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. We assume a set of standard operations on data types, and all abstract data types must provide these operations (as C++ virtual methods).

7.1 Extensibility of Data Types

The type system in CORAL is designed to be extensible; the class mechanism and virtual methods 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 methods 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 methods include the method *equals* that is used to check if two objects are equal, the method *print* for printing the object, the method *construct* that is used to create new objects from a list of arguments (used to re-create objects given a printed representation), *hash* to return a hash value, and some memory management functions. For a summary of the virtual methods that constitute the abstract data type interface, see [24, 21]. In addition to creating the abstract data type, the user can define predicates to manipulate (and possibly display in novel ways) objects belonging to the abstract data types. These predicates must be registered with the system; registration is accomplished by a single command.

[9]That is, the new code is incrementally loaded into CORAL.

7.2 Extensibilty of Access Structures

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 **??**).

8 Related Systems

There are many similarities between CORAL and deductive database systems such as Aditi [31], EKS-V1 [32], LDL [15, 5], Glue-NAIL! [13, 17], Starburst SQL [14], DECLARE [11], ConceptBase [8] and LOLA [6]. However, there are several important differences, and CORAL extends all the above systems in the following ways:

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

EKS-V1 supports integrity constraint checking, hypothetical reasoning and provides some support for non-stratified aggregation [12]. ConceptBase supports DATALOG, along with locally stratified negation (but no set-generation), 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 stratified programs, 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.

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. in that 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 [10]). LDL++, a successor to LDL under development at MCC Austin, is reportedly also moving in the direction taken by CORAL in many respects. It will be partially interpreted, support abstract data types, and use a local semantics for choice (Carlo Zaniolo, personal communication). XSB is a system being developed at SUNY, Stony Brook. It will support several features similar to CORAL, such as non-ground terms and modularly stratified set grouping and negation. Program evaluation in XSB will use OLDTNF, which has been implemented by modifying the WAM (David S. Warren, personal communication). DECLARE and SDS are early efforts to commercialize deductive database technology.

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

9 Conclusions

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. 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.

Modular Design : 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.

Extensibility : The decision to design an extensible system seems to have helped greatly in keeping our code clean and modular, in addition to its utility from an application development perspective.

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 toolkit. Thus CORAL could build on these 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 [29].

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. While this does make evaluation more efficient, it requires extensive memory management and garbage collection. Also, pointer based copying is performed even for primitive data types such as integers.

There are a number of directions in which CORAL could be, and in some cases needs to be, extended. These include better support for persistent data, improved memory management, enhanced C++ interface features, object-oriented extensions and support for constraints. 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 also needs to be performed.

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 some aspects of the set-manipulation code; Tarun Arora implemented several utilities and built-in libraries; Tom Ball implemented an early prototype of the semi-naive evaluation system; Lai-chong Chan did the initial implementation of existential query optimization; Sumeer Goyal implemented embedded CORAL constructs in C++; Vish Karra implemented pipelining; Robert Netzer did the initial implementation of Magic rewriting; and Bill Roth created test suites, implemented the Explanation tool along with Tarun, and added user-interface features.

References

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

[2] F. 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.

[3] C. Beeri and R. 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.

[4] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the International Conference on Very Large Databases*, Aug. 1986.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] W. Kiessling. DECLARE and SDS: Early efforts to commercialize deductive database technology. Submitted to the VLDB Journal., 1993.

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

[13] K. Morris, J. D. Ullman, and A. Van Gelder. Design overview of the NAIL! system. In *Proceedings of the Third International Conference on Logic Programming*, 1986.

[14] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. Duplicates and aggregates in deductive databases. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, Aug. 1990.

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

[16] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. 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.

[17] G. Phipps, M. A. Derr, and K. A. Ross. Glue-NAIL!: A deductive database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 308–317, 1991.

[18] R. 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.

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

[20] R. Ramakrishnan, P. Bothner, D. Srivastava, and S. Sudarshan. CORAL: A database programming language. In J. 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.

[21] R. Ramakrishnan, P. Seshadri, D. Srivastava, and S. Sudarshan. The CORAL user manual: A tutorial introduction to CORAL. Manuscript, 1993.

[22] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. *IEEE Transactions on Knowledge and Data Engineering (to appear).* A shorter version appeared in VLDB, 1990.

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

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

[25] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. The Save Module facility in CORAL. Manuscript, 1993.

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

[27] 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.

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

[29] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. CORAL++: Adding object-orientation to a logic database language. Submitted.

[30] S. Tsur and C. Zaniolo. LDL: A logic-based data-language. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 33–41, Kyoto, Japan, August 1986.

[31] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, and P. Stuckey. The Aditi deductive database system. In *Proceedings of the NACLP'90 Workshop on Deductive Database Systems*, 1990.

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