*VLDB Journal*, 3, 123-160 (1994), Kotgiri Ramamohanarao, Editor



# The Glue-Nail Deductive Database System: Design, Implementation, and Evaluation

**Marcia A. Derr, Shinichi Morishita, and Geoffrey Phipps**

*Received April, 1993; revised version accepted December, 1993.*

**Abstract.** We describe the design and implementation of the Glue-Nail deductive database system. Nail is a purely declarative query language; Glue is a procedural language used for non-query activities. The two languages combined are sufficient to write a complete application. Nail and Glue code are both compiled into the target language IGlue. The Nail compiler uses variants of the magic sets algorithm and supports well-founded models. The Glue compiler's static optimizer uses peephole techniques and data flow analysis to improve code. The IGlue interpreter features a run-time adaptive optimizer that reoptimizes queries and automatically selects indexes. We also describe the Glue-Nail benchmark suite, a set of applications developed to evaluate the Glue-Nail language and to measure the performance of the system.

**Key Words.** Language, performance, query optimization.

## 1. Introduction

A current focus of database systems research is the design of programming languages and systems to support non-traditional database applications such as computer aided design, software engineering, and financial analysis. The Glue-Nail database system (Phipps et al., 1991; Derr et al., 1993), which was developed at Stanford University, provides two complementary languages for programming such applications. The Glue procedural language (Phipps, 1990, 1992) augments relational-style queries

---

Part of this article was presented at the ACM SIGMOD International Conference on Management of Data, Washington, DC, 1993.

Marcia A. Derr, Ph.D., is Technical Staffmember, AT&T Bell Laboratories, 600 Mountain Avenue, Room 2B430, Murray Hill, NJ 07974-0636 USA; Shinichi Morishita, Ph.D., is Advisory Researcher, IBM Japan, Tokyo Research Laboratory, 5-19, Sanban-cho, Chiyoda-ku, Tokyo 102, Japan; Geoffrey Phipps, Ph.D., is Technical Staffmember, Sun Microsystems Laboratories, Inc., 2550 Garcia Avenue, MTV 28-112, Mountain View, CA 94043-1100 USA. Much of this research was done while the authors were at Stanford University, Stanford, California, USA.

with control structures, update operations, and I/O. The Nail declarative language (Morris et al., 1986, 1987) provides rules for expressing complex recursive queries or views.

The purpose of this article is to describe the design and implementation of the Glue-Nail database system. In particular we focus on how we optimized the output or the performance of each major component of the system. We describe a set of benchmark application programs and present performance results that demonstrate the synergetic effects of these optimizations. We also compare a Glue-Nail application with a version written in C and evaluate the design of Glue based on our experiences with the system.

We begin by reviewing the background and underlying design philosophy of the Glue-Nail system. Glue-Nail evolved from NAIL! (Morris et al., 1986, 1987), a deductive database system that featured a logic-based query language, Nail.[1] Logic-based query languages such as Nail have proved to be powerful query languages, but have weaknesses as well as strengths. Because logic is side-effect free and declarative (i.e., the execution order is unspecified), queries can be expressed clearly and optimized easily. But the logical basis is also a weakness because there are operations, such as updating the database and performing I/O, which *do* have side effects, and hence require a procedural language (i.e., a language where the execution order *is* specified). To become a useful database language, Nail needs procedural operations, yet these very same operations are at odds with the semantics of Nail.

Our solution is the two language architecture of Glue-Nail. Nail provides all the strengths of a logic-based query language. Glue complements Nail with procedural features. The problem with this approach is that it involves the design of yet another programming language. The new language must offer significant advantages over existing languages, C/C++ and Prolog being the main contenders. Glue (Phipps, 1992) was designed to offer such advantages by reducing the impedance mismatch problem with Nail. Glue is much closer in semantics and syntax to Nail than C++. Glue has one advantage over Prolog, notably that both Glue and Nail are set-oriented, whereas Prolog is tuple-oriented.

A major part of Glue is the *Nailog* term syntax system (a variant of HiLog; Chen et al., 1989). The Nailog term syntax allows a subgoal to have a variable as its predicate name. For example, the term Z(X,Y) has the variable Z as its predicate name.[2] In most other logic-based languages, the predicate name must be known at compile time. The Nailog system gives the programmer additional power and flexibility.

---

1. We use "NAIL!" to denote the system and "Nail" to denote the language.

2. We use the usual logic programming convention whereby variables begin with upper case letters, and constants start with lower case letters.

Another weakness of the original NAIL! system was the loose coupling between the front end and the back end of the system. The front end of the system translated a Nail program into an intermediate language called ICODE. The back end of the system was an ICODE interpreter that generated SQL statements, which were executed by an underlying commercial relational database system. Typical Nail programs were compiled into code that created many temporary relations. These temporary relations were managed by the same facilities that managed persistent shared relations on disk. Consequently, temporary relations, which were usually small, short-lived, and did not need to be shared, incurred the same overhead as persistent relations. Another problem was that there was no way to control query optimization in the commercial database system. Finally, the loosely-coupled configuration was too slow, because it involved multiple levels of interpretation and retrieved answers one tuple at a time.

Our solution to the architecture problem was to design a complete system tailored to the characteristics of Glue and Nail. The three major components of the system are the Glue compiler, the Nail compiler, and the IGlue interpreter. In this approach, Nail rules and Glue code are both compiled into a target language called IGlue.[3] IGlue code is executed by the IGlue interpreter, which manages all relations and indexes in main memory. One of the advantages of this architecture is the opportunity it provides for various kinds of optimizations. The Glue compiler includes a static code optimizer that uses peephole techniques and data flow analysis. The Nail compiler performs recursive query optimizations. The IGlue interpreter provides an adaptive optimizer that optimizes queries at run time.

The remainder of this article is organized as follows. Section 2 describes the Glue-Nail language pair. Section 3 gives an overview of the system architecture. Sections 4, 5, and 6 describe the major components of the system: the Nail compiler, the Glue compiler, and the IGlue interpreter. Section 7 describes the Glue-Nail application benchmark and presents performance results and an evaluation of the Glue language system. Section 8 compares Glue-Nail to several other deductive database systems. Finally, Section 9 presents some conclusions.

## 2. The Glue-Nail Language

Glue and Nail are two complementary languages that together enable a programmer to write a complete database application. We describe features of the two languages. To facilitate the description we present an example Glue-Nail program in Figure 1. This program computes a *bill of materials*; that is, for a hierarchy of parts, it computes the quantity of each basic part required to build a complex part. Basic parts are described by the persistent or *Extensional Database*

3. Pronounced "igloo."

**Figure 1. Example Glue-Nail program that computes a bill of materials**

```
 0)    module bill;
 1)    export main(:);
 2)    from io import read(:X), write(X:);
 3)
 4)    edb     part_cost(BasicPart, Supplier, Cost, Time),
 5)            assembly(Part, SubPart, Qty);
 6)
 7)    %-----------Glue procedures-----------
 8)
 9)    prog main(:)
10)    rels answer;
11)
12)       answer :=
13)            read(P) &
14)            bom(P,B,Q) &
15)            A = P // '\t'  // B  // '\t'  // Q  // '\n' &
16)            write(A).
17)       return(:).
18)
19)    end
20)
21)    proc bom(Root: Raw, Q)
22)    rels unknown(P), p(P,SP,Q), notyet(P), anymore;
23)
24)       unknown(Root) := in(bom(Root)).
25)       unknown(P) += in(bom(Root)) & partstc(Root, P).
26)       p(P, P, 1) := unknown(P) & part_cost(P,_,_,_) & --unknown(P).
27)       repeat
28)          notyet(P) := unknown(P) & assembly(P,Child,Q) & unknown(Child).
29)          p(P, Raw, Q) +=
30)             unknown(P) &
31)             ! notyet(P) &
32)             assembly(P, SP, M) &
33)             p(SP, Raw, N) &
34)             S = N*M &
35)             group_by(P, Raw) &
36)             Q = sum(S) &
37)             --unknown(P).
38)          anymore := in(bom(Root)) & unknown(Root).
39)       until !anymore;
40)       return(Root: Raw, Q) := p(Root, Raw, Q).
41)    end
42)
43)
44)    %-----------Nail rules------------
45)
46)    partstc(X,Y) :- assembly(X,Y,_).
47)    partstc(X,Z) :- assembly(X,Y,_) & partstc(Y,Z).
48)
49)    end
```

(EDB) relation `part_cost(BasicPart, Supplier, Cost, Time)`. The EDB relation `assembly(Part, SubPart, Qty)` describes the part-subpart hierarchy. Each tuple describes a complex part, one of its immediate subparts, and the quantity of that subpart.

### 2.1 Nail Rules

Nail is a declarative language in which the user can define views or derived relations in terms of logical rules. These views are also referred to as *Intensional Database* (IDB) relations. For example, the recursive rules in lines 46–47 of Figure 1 define the IDB relation, `partstc(X,Y)`, which is the transitive closure of the EDB relation, `assembly(X,Y,Q)`. As with EDB relations, this relation can be queried in several ways by providing a set of bindings for one or more variables. For example, one may ask if the set of tuples {`(bicycle, spoke)`, `(bicycle, wheel)`} is in the `partstc(X,Y)` relation. In this case, both arguments are bound, and the query can be described as `partstc(X,Y)`$^{bb}$. The Nail query `partstc(Root,P)` in line 25, asks for all subparts `P` that are in the transitive closure of the bound argument `Root`. Because the first argument is bound and the second argument is free, this query is described as `partstc(X,Y)`$^{bf}$. As will be described in Section 5, the Nail compiler uses the binding pattern of a query to determine how to evaluate a set of Nail rules.

### 2.2 Glue Assignment Statements

The basic instruction element of Glue is the assignment statement. An assignment statement performs joins over relations, Glue procedures, and Nail predicates, and assigns the result to a relation. Glue assignment statements are not logical rules, they are operational directives. Assignment statements do not define tuples, they create or destroy tuples. Consider the example assignment statement in line 28 of Figure 1. The effect of executing this statement is to join relations `unknown(P)`, `assembly(P,Child,Q)`, and `unknown(Child)`; project the set of tuples `(P)` from the result of the join; and assign this set to the relation `notyet(P)`.

In their basic form, Glue assignment statements have a single head relation, and a conjunction[4] of subgoals in the body. The body of the assignment statement is evaluated and produces a set of tuples over the variables in the body. The tuples are used to modify the head relation. Glue allows subgoals to be negated. It also allows update operators to be applied to subgoals. In Figure 1, line 31 is an example of a negated subgoal, and line 37 is an example of a subgoal with a delete operator.

The semantics of the Glue assignment statement are defined by a left-to-right evaluation order, where all solutions are found for each subgoal before evaluating the next subgoal. The evaluation order is fixed for purposes of side-effects and

4. The body can contain other control operators, such as OR and a form of implication, but space precludes their inclusion in this paper.

aggregation. The underlying implementation of the system, however, is free to reorder subgoals that have no side-effects.

There are three assignment operators in Glue:

[ := ] Clearing assignment. The head relation is overwritten by the result of the body.

[ += ] Insertion assignment. The tuples from the body are added to the head relation.

[ -= ] Deletion assignment. The tuples from the body are removed from the head relation.

Several examples of clearing and insert assignment operations can be found in the Glue code in Figure 1.

### 2.3 Relations and Terms

There are two kinds of relations in Glue: EDB relations and local relations. EDB relations persist beyond the execution of any single program. Local relations are defined within the scope of Glue procedures, and have a lifetime equal to the lifetime of a procedure call stack frame.

An attribute of a tuple is represented by a ground (variable-free) Nailog term. Nailog is an extension of the usual logic programming term syntax and semantics, and is a subset of HiLog (Chen et al., 1989). HiLog and Nailog have second order syntax, but first order semantics (Lloyd, 1984). While HiLog places no restrictions on the use of terms as subgoals, Nailog does make restrictions for the sake of efficiency (see below). Nailog provides an elegant computational model for meta-programming and sets (see Section 2.4).

A Nailog term can denote a string, a number, a variable, or a compound term. The functor of a compound can itself be an arbitrary term. A tuple that illustrates a variety of Nailog terms is shown below.

```
(foo,'Jane Doe', 37, 14.5, f(a,b), g(h)(1,2), X(1), p(X)(Y), X(Y)(Z))
```

The first two terms are strings, the third and fourth terms are numbers, and the remaining five are compound terms. The last four terms are not legal terms in traditional logical term syntaxes.

Only tuples containing ground terms can be stored in Glue relations. In the tuple above, only the first six terms are ground. The remaining three terms contain variables and hence could not be stored in a relation. This ground-only restriction allows the IGlue interpreter to use only matching when comparing subgoals against a relation, rather than using full unification. When matching two terms, at most one of the terms can contain variables, the other can only contain constants. Hence only the variables in one term need to have their bindings updated. When unifying two terms, both of the terms can contain variables. Hence both terms need to have

their bindings updated. Complex feedback loops can exist between the variables in both terms, further complicating the process. Matching is (in general) much faster than unification.

In Nailog, variables range over predicate *names*, not over predicate *extensions* (values). This distinction is important, because the set of predicate names is finite, whereas the set of possible predicate extensions is infinite. The scoping rules of Glue's modules and procedures provide the compiler with a list of the predicate names with which a subgoal variable could possibly unify, so most of the predicate selection analysis can be done statically at compile time. Hence much of the cost of meta-programming is avoided.

All Glue subgoals must have a completely bound predicate name at run time. For example, the following are all legal Glue subgoals, assuming that variable `Y` is bound: `f(G,J)`, `Y(K)`, and `f(Y)(K)`. There is one specific exception in Nailog: a subgoal of a single variable (e.g., `Y`) is illegal. The meaning of such a subgoal is ambiguous and the most obvious meanings are computationally very expensive. These two restrictions are the only differences between HiLog and Nailog. They are designed to allow programmers to write the programs that they need to write, without paying the penalty for programs that never need to be written.

Notice that both compound terms and predicate terms can have arbitrary terms as their functors, rather than being limited to atoms as in standard first order logic-based languages. It is important that compound terms and predicate terms both have the same syntax for two reasons. First, the language model is made cleaner by the existence of a single syntax. Second, it would make it impossible to store complicated Nailog predicate names as terms in a tuple. For example, the predicate term `wafer(metal)(layer2)(X,Y)` has a Nailog term `wafer(metal)(layer2)` as its principal functor. If compound terms could not use Nailog syntax, then we could not store the name of this predicate in a relation.

### 2.4 Sets and Meta-programming

The limitations of first normal form for representing attributes are widely recognized. It is often more natural to express an attribute as a set than to flatten it into first normal form (Wiederhold, 1986). Allowing set-valued attributes also solves many null-value problems (Makinouchi, 1977).

A common problem with adding set-valued attributes to relational or deductive databases is that sets are often introduced as an entirely new data type. The new set data type needs special operators, such as set-membership, set-insertion, set-deletion, and set-unification. While this approach may work, it unnecessarily complicates the language model. A programmer should be able to use the relational semantics to handle sets, because relations are nothing more than sets of tuples. This is the approach taken by Glue and Nail.

Sets in Glue are manipulated by storing the name of a predicate (i.e., the name of a set or relation), rather than the value (members) of a set. Sets are therefore regular relations. Remember that subgoals may have variables for their predicate

**Figure 2. Example of set construction in Glue**

```
class_info( ID, Ins, Room, tas(ID), students(ID) ):=
    class_instructor( ID, Ins ) &
    class_room( ID, Room ).
tas(ID)(Grad_student):=
    class_subject( ID, Subject ) &
    failed_exam( Grad_student, Subject ).
students(ID)(S):=
    attends( S, ID ).
```

names. Therefore we can store the name of a relation in a tuple, then extract it using a variable and use that variable as a subgoal name. For example:

```
dept_employees( toy, E_set ) & E_set( Emp_name ) & ...
```

The second attribute of the `dept_employees` relation is the name of the relation which holds the employees in the toy department.

An example of set definition is shown in Figure 2. The relation `class_info(_,_,_,_,_)` contains information about a class: its identifying code, instructor, set of teaching assistants (TAs), and set of students. The relation `tas(ID)(_)` defines the TAs for course `ID`, notably those graduate students who failed the graduate qualifying exam in the course's subject area. Observe that the name of this relation is a compound term. The relation `students(ID)(_)` contains the names of the students who are taking course `ID`. The other relations are defined elsewhere. A typical use of the `class_info` predicate might be:

```
class_info(C,I,R,T,S) & T(TA) & S(Student)
```

That is, `class_info` provides bindings for variables `T` and `S`, which are names of sets (i.e., relations).

The foregoing argument is only concerned with the language model that is presented to the programmer. Note that nothing has been said about the implementation of sets or relations. The arbitrary distinction between sets-of-tuples (relations) and sets-of-other-things (sets) has been removed. The number of concepts in the language has been reduced to have a cleaner computational model. A cleaner model should lead to more efficient coding, and fewer errors. A system with a computational model that distinguishes between sets-of-tuples and sets-of-other-things could have special optimizations to deal with the latter. However, such a design instantly raises the question: "Why not provide these optimizations for sets-of-tuples as well?" On the other hand, if the two types of sets share the same implementation, but are distinguished in the computational model, then nothing has been gained by complicating the programmer's computational model.

### 2.5 Aggregation

Aggregation in Glue occurs at the subgoal level. The aggregation operators are Glue subgoals, and they operate over the tuples that have been produced in the assignment statement body up to that point. For example, consider the following statement that finds the shortest stick in a relation of sticks:

```
shortest(Name,L):= stick(Name,Length) & L = min(Length) & L = Length.
```

The relation `stick(Name,Length)` contains the name and lengths of a set of sticks. Suppose that its contents are:

| Name | Length |
|---|---|
| stick_1 | 12 |
| stick_2 | 8 |
| stick_3 | 9 |

If we evaluate the above statement, then the bindings for the variables after each subgoal will be:

| Name | Length |
|---|---|
| stick_1 | 12 |
| stick_2 | 8 |
| stick_3 | 9 |

| Name | Length | L |
|---|---|---|
| stick_1 | 12 | 8 |
| stick_2 | 8 | 8 |
| stick_3 | 9 | 8 |

| Name | Length | L |
|---|---|---|
| stick_2 | 8 | 8 |

Notice that the final subgoal joins variables `L` and `Length`, eliminating all but the shortest sticks. The relation `shortest(Name,L)` will be assigned the the single tuple `(stick_2,8)`.

At first glance, it appears that our treatment of aggregation is inherently inefficient. In the example above, the aggregate `L = min(Length)` is appended to every `(Name, Length)` tuple because all three values are needed later in the evaluation of the query. This is a property of that particular Glue statement, not of the method for computing aggregates. In the following variation

```
minlength(L) := stick(Name,Length) & L = min(Length).
```

the aggregate result `L` can be projected directly onto the head relation, and the set of tuples `(Name, Length, L)` is never materialized.

The assignment statement in lines 29–37 of Figure 1 illustrates aggregation with grouping. The term `group_by(P,Raw)` partitions the set of tuples `(P,SP,M,Raw,N,S)` by the grouping key `(P,Raw)`. Then for each partition, the aggregate operator `sum(S)` computes the sum of variable `S`. The sum is assigned to variable `Q`.

## 2.6 Control Statements

Glue offers several control constructs, which have been borrowed from familiar procedural languages. These include three different loop constructs and an if-then-else construct. The condition tests for control statements are Glue assignment statement bodies. A test is true if it returns at least one tuple; it is false otherwise. In other words, condition tests are existence tests. Lines 27–39 in Figure 1 illustrate a `repeat-until` loop. The condition test is the subgoal body `!anymore`, which it evaluates to true if local relation `anymore` is empty. The loop body always executes at least once, because the condition test is evaluated after the loop body.

## 2.7 Glue Procedures

Glue procedures are analogous to Nail rules in that they compute a set of tuples from the current state of the EDB. Glue procedures differ from Nail predicates in two ways. First, the operational semantics of a Glue procedure are specified by the programmer, whereas a set of Nail rules has no operational semantics, only declarative semantics. Second, Nail rules can be called with any binding pattern, whereas Glue procedures have a set of arguments which must be bound when the procedure is called.

We will explain the structure of Glue procedures using the code in lines 21–41 of Figure 1. The name of this procedure is `bom(_:_,_)`. Informally, procedure `bom(Root:Raw,Q)` computes the quantity `Q` of each basic part `Raw` that is used to construct complex part `Root`. The procedure's arity is one-to-two; given one bound input argument, it produces ternary tuples. The colon is used to separate the arguments that must be bound from the arguments that can be bound or free. The arguments to the left of the colon must be bound. Whenever `bom(Root,Raw,Q)` is used as a subgoal, its first argument must be bound. The second and third arguments may be either bound or free. More correctly, given a set of unary tuples (over attribute `Root`), the procedure `bom(Root:Raw,Q)` extends these tuples to be a set of ternary tuples `(Root,Raw,Q)`, such that `Raw` is a basic part that appears `Q` times in part `Root`.

The procedure has several local relations, which are declared in line 22. Procedures may be called recursively. Each invocation of a procedure has its own copies of its local relations. Declarations of local relations "hide" the declarations of other predicates with which they unify.

All procedures have two special relations, `in` and `return`. The relation `in` holds the procedure's input tuples. The relation `in` contains a single term argument whose functor is the name of the procedure and whose arity is equal to the bound arity of the procedure (i.e., the arity to the left of the colon in the procedure definition). The relation `return` holds the procedure's output tuples. Assigning to this relation also has the effect of exiting the procedure. The `return` relation has the same arity as the procedure. An assignment statement that assigns to the `return` relation has an implicit `in` subgoal as its first subgoal. For example, below we see line 40,

rewritten to include the implicit `in` relation:

```
return(Root:Raw,Q):= in(bom(Root)) & p(Root,Raw,Q).
```

The implicit `in` relation has a natural meaning; it restricts the return relation to only those tuples which extend the input relation.

When a Glue procedure is used as a subgoal it is called once on all of the bindings for its input arguments, rather than being called many times (i.e., once for each tuple in the binding set).

### 2.8 Modules

Both logic programming and deductive database languages have had problems "programming in the large," partly due to their lack of large scale code organization structures. Hence, in common with several other languages, Glue-Nail has a module system. Modules provide statically scoped naming contexts, as they do in languages like Modula-3. Unlike CORAL modules, Glue modules do not have any dynamic effect on scope. CORAL modules can be "called" with predicate arguments to allow meta-programming. Glue and Nail achieve the same result by using Nailog, where any procedure or rule can use variables for predicate names. Besides offering the usual advantages of separate compilation and modularity, the module design of Glue also gives the compiler valuable information concerning which predicates are visible at any point in a program. This information can be used to perform much of the predicate dereferencing at compile time instead of run time.

Modules have:

- A name,
- A list of imported EDB predicates,
- A list of imported Nail predicates and Glue procedures,
- A list of exported Nail predicates and Glue procedures, and
- Code, both for Glue procedures and Nail rules.

Notice that a module can contain both Glue procedures and Nail rules, thus allowing the programmer to group predicates by function, rather than by language type. Glue also provides a predefined input/output module from which procedures can be imported. For example, in line 2 of Figure 1, module `bill` imports procedures `read(:X)` and `write(X:)` from module `io`.

### 2.9 Using Nail without Glue

Although Glue was designed to be used with Nail, Nail could be embedded in languages like C and COBOL. Huge amounts of legacy code exist in the world. The embedding could be done in a similar fashion to embedded SQL. A Nail call would reference existing database tables, and would return another table as its result. Nailog term syntax would only be supported if the existing database supported it, which is unlikely. Hence Nail would be reduced to just datalog (constants, no

function symbols). No attempt has been made to implement such a port, we mention it merely for interest.

## 3. System Overview

The Glue-Nail database system was designed as a memory-resident system that supports single-user applications. This design targets small to medium-sized applications where the database does not need to be shared or where portions of database may be checked out for relatively long periods of time. All queries and updates operate on main memory representations of relations. Between executions of programs, the EDB relations reside on disk.

The Glue-Nail system architecture consists of the Glue compiler, the Nail compiler, the static optimizer, the linker, and the IGlue interpreter. The configuration of these components is shown in Figure 3. The input to the system is a Glue-Nail program, which consists of one or more code modules. Each module can be compiled separately.

The Glue compiler separates the Glue code from the Nail rules in module. It compiles the Glue code into the target language, IGlue, which will be described in Section 6.1. The Glue compiler also passes each Nail query and its associated set of rules to the Nail compiler. The Nail compiler transforms the Nail query and rules into an IGlue procedure that computes the answer to the query. The linker collects all relevant IGlue code into a single file. The IGlue code is optionally analyzed and transformed by the static optimizer.

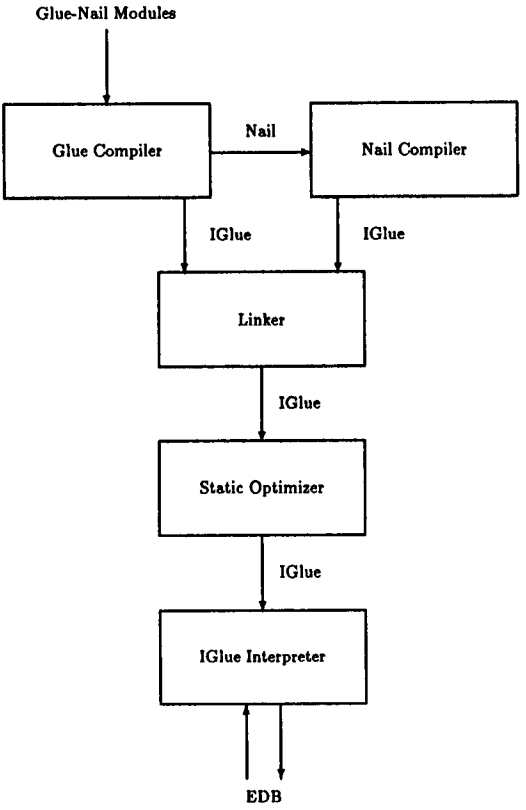
The IGlue interpreter reads the IGlue program, and loads into memory the disk-resident EDB relations that the program will access. As the interpreter executes the IGlue program, it calls the run-time optimizer to adapt query plans to changing parameters of the database. When the interpreter halts, it writes to disk any EDB relations that have been updated.

The Glue compiler is written in Prolog and C. The Nail compiler is written in Prolog. The linker is written in C. The static optimizer and the IGlue interpreter are written in C++. The system was developed on a DEC5000 and workstations with 16 to 32 megabytes of main memory. It has been ported to SPARC 2 and SPARC 10 workstations, running both SunOS 4.1.x and Solaris 2.x. While the current hardware presents limitations on the size of programs this approach can handle, we believe that trends toward larger main memories make this approach feasible for a variety of applications. Departmental server machines with gigabytes of main memory are now available.

## 4. The Glue Compiler

Compiling Glue into IGlue takes place in four phases: parsing, code generation, linking, and static code optimization. Parsing and linking are straightforward and

**Figure 3. Glue-nail system architecture**




will not be discussed any further. Instead we will concentrate on how the code generator and static optimizer produce quality IGlue code.

As with any compiler, the major technical problem faced in building the Glue compiler was to be able to produce code that is both correct and efficient. The Glue compiler is responsible for static optimization of Glue programs. One way to provide for efficient execution is to reduce the number of IGlue operations and relations. The Nailog semantics of Glue and Nail were particularly difficult to handle efficiently, because in general the predicate to which a Glue subgoal refers can only be determined at run time.

Three strategies are used by the Glue compiler: early identification of procedure calls, reduction of compiler-generated storage space, and removal of redundant operations. The first two problems were solved by careful design of the code generator, the latter problem was solved by an IGlue-to-IGlue static optimizer. This optimizer is also capable of improving the IGlue code produced by the Nail compiler. We will elaborate on these compiler optimization strategies here.

### 4.1 Optimizations in the Code Generator

Two optimizations are performed in the code generator. These optimizations were not present in the first implementation of the code generator, but were added when their need became apparent.

*Predicate Class Analysis and Procedure Calls.* As mentioned earlier, the Nailog term system allows subgoals to have variables as their predicate names. Predicate selection is the process of associating a subgoal term with a particular relation, or procedure, or Nail predicate. Predicate selection for Nailog subgoals with variables can in general only be resolved at run time. The simplest solution would be to perform predicate selection at run time, when the functor is fully bound. However, that would mean treating every subgoal as a potential procedure call, which is expensive. Hence the Glue compiler does as much of the predicate name resolution as possible at compile time. Subgoals are unified against the predicates that are visible in the current scope. For subgoals with ground predicate names there can be at most one match. For subgoals with variable predicate names, the number of possible matches is usually reduced (and can never be increased). Hence the amount of checking to be done at run time is reduced. In particular we can often prove that a subgoal cannot match a procedure call, so the less expensive relation look-up can be used. Compile-time predicate analysis was found to increase the speed of the PATH benchmark (described in Section 7) by 315%. The unoptimized version of the code treated all subgoals as potential procedure calls. The optimized code performed compile-time analysis to distinguish between relation subgoals and procedure call subgoals.

*Temporary Relation Compression.* Temporary relation compression involves analysis of the variables that are stored in compiler-generated temporary relations. The compiler uses these relations to store variable bindings between IGlue instructions. The simple approach that was first implemented was to record all known variable bindings in the temporary relations. Compression ensures that the only variable bindings that are recorded in the temporary relations are those that will be subsequently used. The effect of this optimization on programs from the benchmark suite is shown in Table 1. The absolute error in the code speed improvements is one percentage point. We note that this optimization has been present in most relational databases since System R, but we mention it here because it was the most effective static optimization performed by the Glue compiler. As so often happens in compiler optimization, the obvious and simple ideas fix most of the problems.

### 4.2 Static Optimizations

The Glue static optimizer is an IGlue to IGlue code transformer. It is a static (compile time) optimizer and should not be confused with the dynamic (run time) optimizer in the IGlue interpreter. A number of different optimization techniques are used. The general algorithm is shown in Figure 4. Peephole analysis uses only

**Table 1. Temporary relation compression**

| Program | BILL | CIFE | SG | SPAN | CAR | $THOR_p$ | $THOR_s$ | OAG |
|---|---|---|---|---|---|---|---|---|
| Total arity, before | 53 | 36 | 36 | 60 | 248 | 1422 | 723 | 281 |
| Total arity, after | 29 | 31 | 8 | 42 | 121 | 951 | 408 | 214 |
| % arity reduction | 45 | 14 | 78 | 30 | 51 | 33 | 44 | 24 |
| % speed-up | 13.5 | 11.6 | 26 | 4.7 | 35 | 9.2 | 53 | 3.5 |

**Figure 4. Static optimization algorithm**

```
peephole();
do {
     perform_data_flow_analysis();
     constant_propagation();
     copy_propagation();
     peephole();
} while changes;
```

local information. Constant and copy propagation require data flow analysis. The aim of these optimizations is to reduce the number of operations on relations, or to remove a relation entirely.

In addition to the above optimizations, a cardinality analysis algorithm has been developed. It identifies relations that contain at most one tuple (known as "singleton relations"). These relations can be replaced by simpler data structures that can hold only a single value. The required changes to the IGlue language and interpreter have not been made, so performance numbers are not available. However, the analysis algorithm is very effective at locating singleton relations. The algorithm relies on the programmer's declaring that some relations are singleton. Abstract interpretation using fixpoints is employed to compute the maximum cardinalities of all the other relations. Relation sizes are limited to $\{0, 1, \mathit{LARGE}\}$. Arithmetic over this set is closed, and repeated operations reach a fixpoint very quickly (usually *LARGE*). Hence the algorithm as a whole terminates, because any relations involved in loops reach their fixpoint cardinality very quickly. In one particular case in the THOR parser (described in Section 7), the analyzer algorithm was able to prove that 65 out of 66 local and temporary relations were single-tuple relations after being given one declaration by the programmer.

*Peephole Techniques on Joins.* Most of the work in a database takes place inside the joins, so improving the join code is important. Glue static optimization can only reduce the number of relations and variables within the join. The static optimizer

**Table 2. Effect of DFA**

| Program | BILL | CIFE | SG | SPAN | CAR | $THOR_p$ | $THOR_s$ | OAG |
|---|---|---|---|---|---|---|---|---|
| Code size before | 60 | 151 | 41 | 49 | 78 | 938 | 429 | 116 |
| Code size after | 52 | 139 | 35 | 39 | 56 | 713 | 307 | 93 |
| % size reduction | 13 | 7.9 | 9.8 | 20 | 28 | 24 | 28 | 20 |
| % speed-up | 3.8 | 2.4 | 3.5 | 5.4 | 14.1 | 5.5 | 5.6 | 6.1 |

analyzes the use of variables within the join. The optimizer converts existential variables into "don't care" variables, performs any unifications that can be done at compile time, and removes joins that do not update any relation.

These peephole techniques are quick to execute because they use only local information about a join. Note that the other optimizations may change the code so that the peephole techniques may be reapplicable. Hence the peephole optimizer is run after each pass through the main optimizer loop.

Contrary to expectations, it was found that peephole optimization had only a marginal effect on execution speeds. Speed-ups for the application suite varied between 0% and 3%.

*Data Flow Analysis.* Data Flow Analysis (DFA) of IGlue code is similar to DFA in imperative languages with single-valued variables, but there are some important differences. First, DFA in IGlue is concerned with relations, not variables. Second, IGlue has more opportunities for aliasing than traditional three-address code.

The optimization techniques that depend on DFA are:

- Copy propagation,
- Constant propagation, and
- Reduction in strength of the join operation.

These three techniques are variants on their traditional forms. For example, DFA can often prove that a relation has a known value at some point in an IGlue program. Relations can be proven to be empty, or to contain a set of known tuples. If a relation is provably empty, then references to it may be replaced by `FALSE`. If a relation has a single known tuple, then the value of that tuple can replace the uses of that relation at compile time. If the relation is known to possess more than one tuple, then its uses can be replaced by the multiple tuples, although in practice this has not proven to be cost effective. DFA also identifies operations that do not change the value of a relation (such as clearing an empty relation).

The results of running the DFA optimizations on the benchmark suite (Section 7) are shown in Table 2. The absolute error in the code speed improvements is one percentage point.

This table shows the reduction in the number of IGlue statements achieved by DFA. The speed improvements were not as large as expected. The reason that the improvements in speed were much less marked than the improvements in code size is that all IGlue statements are not created equally. The static optimizer is very good at identifying redundant operations (such as unnecessary data copying and movement). Unfortunately, these joins are usually quick to execute because they *are* redundant. For example, the IGlue interpreter implements a `_MOVE` operation by simple pointer changes, rather than tuple copying. Clearing an empty relation is also very fast. Hence removing redundant operations does little to improve program speed. Most of the IGlue interpreter's execution time is spent computing nonredundant joins. The static optimizer can do very little to improve the internals of these joins, so the peephole optimizations are all that apply. The static optimizer can do more with small joins and copying operations.

For the reasons given above, the run-time optimizer (see Section 6.3) has proven to be effective exactly when the static optimizer is ineffective, and vice versa. The two optimizers achieve a useful synergy.

## 5. The Nail Compiler

The Nail compiler translates a Nail query and its associated set of rules into an IGlue procedure. The research in developing the Nail compiler focused on efficiently evaluating recursive queries.

Among query evaluation methods for deductive databases, the magic-sets transformation (Beeri and Ramakrishnan, 1987; Ullman, 1989) is the most established one, because of its generality and efficiency. The magic-sets transformation generates a program that simulates top-down evaluation with memoing and restricts the search space of subgoals to a subspace of those relevant to the query. We employ variants of the magic-sets transformation developed in the literature of deductive databases.

To optimize recursive query evaluation, the Nail compiler applies one of two variants of the magic-sets transformation to the Nail program. The compiler then chooses a strategy to evaluate the transformed program, depending on whether the program is negation-free, stratified or unstratified. Then the compiler generates the IGlue code that encodes the selected evaluation strategy. The Nail compiler does not choose join orders or select indexes for each IGlue query that it generates. The best plan for evaluating each query may also vary, because the cardinality of each relation may vary at run time. Hence, in Glue-Nail, query optimization is performed adaptively at run time.

Now we give the details of the magic-sets transformation strategy and the evaluation strategy. First the Nail compiler applies one of two variants of the magic-sets transformation as follows:

**if** the input Nail program is negation-free and right-linear **then**
apply the context right-linear transformation (Kemp et al., 1990; Mumick and Pirahesh, 1991);
**else**
apply the supplementary magic-sets transformation (Ramakrishnan, 1988; Ullman, 1989);

Then the compiler chooses one of three evaluation strategies as follows:
**if** the transformed program is negation-free **then**
evaluate it using semi-naive bottom-up evaluation (Bancilhon, 1986);
**else**
**if** the transformed program is stratified **then**
evaluate it by Kerisit-Pugin's (1988) method;
**else**
evaluate it by the alternating fixpoint tailored to magic programs (Morishita, 1993);

The Nail compiler makes these selections at compile time, because all conditions in the above algorithms can be tested by checking only the program syntax. The method proposed by Kerisit-Pugin (1988) and the alternating fixpoint technique both perform bottom-up evaluation in a semi-naive fashion. The Nail compiler generates the IGlue code that implements the selected evaluation strategy and passes it to the static optimizer and the linker.

The previous Nail compiler applied the method of Ross (1990) to every Nail program. We replaced the previous method with several better strategies for the following reasons:

- For negation-free programs it is more efficient to apply the supplementary magic-sets transformation to the input and evaluate the transformed program using the semi-naive bottom-up method. Furthermore, for the class of right-linear programs, which includes many common recursions such as transitive closure, the context right-linear transformation is much more efficient than the magic-sets.

- We can easily decide whether a set of rules is stratified just by looking at the syntax of rules. For stratified programs, several methods that make use of the stratification have been proposed. It is more efficient to evaluate magic programs by using one of those methods. Because of its simplicity, we employed the method of Kerisit-Pugin (1988).

- The previous compiler handled only modularly stratified Datalog programs, a subclass of general programs that have two-valued well-founded models. The problem with modularly stratified programs is that it is recursively unsolvable to determine for an arbitrary program whether that program is modularly stratified for all EDBs (Ross, 1991). Put another way, it is

not possible to decide syntactically whether a set of rules with negation is modularly stratified. Although there are some sufficient conditions for modular stratification available (Ross, 1991), in general the programmer must guarantee that the given program is modularly stratified to get correct answers using methods for modularly stratified programs. Furthermore, it could be a difficult task for the programmer to ensure this property for complex programs.

The last limitation motivated us to look for a robust algorithm that works for fully general Datalog with negation and with three-valued, well-founded models. In this general setting, however, we discovered that the well-founded model of the magic program may not agree with the well-founded model of the original program. To fix this problem, Kemp et al. (1992) developed a method that tends to generate magic facts and therefore may not restrict the search space well. By slightly tailoring the alternating fixpoint technique (a standard method to compute well-founded models; Van Gelder, 1989), we created a novel method for magic programs. This approach computes the correct answer to the query and always generates fewer (and in some cases significantly fewer) magic facts than the method of Kemp et al. (1992). Its formal presentation, correctness and theoretical properties can be found in Morishita (1993). We implemented this method for fully general Datalog with negation.

Now we show performance results of the new method. Let us use the following win program as a test program, in which `win` is an IDB predicate and `move` is an EDB predicate.

```
win(X) :- move(X,Y) & !win(Y).
```

First we consider modularly stratified instances of the win program and compare the new method with the old one. The above program is modularly stratified if the EDB for `move` is acyclic. We use two different `move` relations that are acyclic. The first relation represents a linear list using tuples of the form: (1,2),. . .,($N$-1,$N$). The second represents a complete binary tree of height $H$. Given query `win(1)`, we compare the code generated by the new compiler with the code created by the previous one. Both programs are executed by the IGlue interpreter under the same optimization conditions. The programs were executed on a SPARC 10/41 with 128 megabytes of memory and running SunOS 4.1.3. Tables 3 and 4 show the evaluation times.

The results show that, although one can write a modularly stratified program for which the new method runs slower than the previous method does, this is not always the case. There are cases where the previous compiler works better than the new one, and cases where the new one is superior to the old, depending on the properties of the EDB relations. The reader might feel that there is no advantage to employing the new method for modularly stratified programs. It should be remembered that methods for modularly stratified programs ask the programmer to guarantee that a

**Table 3. Execution times (seconds) for acyclic linear lists**

| *N* | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|
| new compiler | 0.13 | 0.48 | 1.58 | 6.05 | 25.32 | 95.62 |
| previous compiler | 0.14 | 0.22 | 0.52 | 1.49 | 4.94 | 17.34 |

**Table 4. Execution times (seconds) for complete binary trees**

| *H* | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|
| new compiler | 0.54 | 1.23 | 2.32 | 5.87 | 11.96 | 27.84 |
| previous compiler | 1.49 | 3.80 | 10.89 | 34.41 | 121.05 | 453.75 |

**Table 5. Execution times (seconds) for cyclic linear lists**

| *N* | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|
| new compiler | 0.03 | 0.06 | 0.11 | 0.20 | 0.39 | 0.80 |

program is modularly stratified, while the new method frees the programmer from this task. Furthermore, the new method can deal with non-modularly stratified cases.

In Table 5 we also include performance results of non-modularly stratified cases of the win program in which we use cyclic linear lists, i.e., (1,2),. . .,(*N*-1,*N*),(*N*,1). Because the new compiler can handle three-valued well-founded models, it needs to tell the calling Glue procedure that some queries are undefined. However, the Glue language is based on two-valued logic. This discrepancy is resolved by introducing a new Glue operator "?" for undefinedness. That is, for any Nail predicate *Q*, ?*Q* succeeds if *Q* is undefined in the well-founded model of the Nail program, !*Q* succeeds if *Q* is false, and *Q* succeeds if *Q* is true.

## 6. The IGlue Language and Interpreter

In this section we describe the IGlue target language, its interpreter, and the adaptive optimizer. In particular, we focus on join processing in IGlue.

### 6.1 The IGlue Target Language

IGlue is the target language for both Glue and Nail. IGlue code is executed by the back end of the Glue-Nail system, the IGlue interpreter. Using an example, we will describe the most important features of the language. A more complete description is found in Derr (1992).

**Figure 5. Example IGlue code for transitive closure**

```
 0)    _MODULE nail_bill_partstc_bf_true
 1)        _EDBDECL  assembly/3
 2)
 3)      _PROCEDURE nail_bill_partstc_bf_true/1:1
 4)        _LOCALDECL context_magic/2, delta_context_magic/2,
 5)            old_delta_context_magic/2, answer/2, changed/0
 6)
 7)        _FORALL(
 8)          _IN(in( nail_bill_partstc_bf_true(VV1))),
 9)          ++_LOCAL(delta_context_magic(VV1, VV1)))
10)    %
11)    %   Perform semi-naive-evaluation until no inferences
12)    %
13)    repeat:
14)
15)        _FORALL(
16)          _LOCAL(delta_context_magic(VV2, VV1)),
17)          ++_LOCAL(context_magic(VV2, VV1)))
18)
19)        _MOVE(
20)          _LOCAL(delta_context_magic(_, _)),
21)          _LOCAL(old_delta_context_magic(_, _)))
22)
23)        _FORALL(~~_LOCAL(changed))
24)
25)        _FORALL(
26)          _LOCAL(old_delta_context_magic(VV1, X)),
27)          _EDB(assembly(X, Y, _)),
28)          !_LOCAL(context_magic(VV1, Y)),
29)          ++_LOCAL(delta_context_magic(VV1, Y)),
30)          ++_LOCAL(changed))
31)
32)        _IF _EXISTS(_LOCAL(changed)) _GOTO repeat
33)
34)        _FORALL(
35)          _LOCAL(context_magic(VV1, X)),
36)          _EDB(assembly(X, Y, _)),
37)          ++_LOCAL(answer(VV1, Y)))
38)
39)        _FORALL(
40)          _IN(in(nail_bill_partstc_bf_true(VV2))),
41)          _LOCAL(answer(VV2, VV1)),
42)          ++_OUT(out( nail_bill_partstc_bf_true(VV2), out(VV1))))
43)
44)        _RETURN
```

Several kinds of IGlue instructions are illustrated in the code in Figure 5. This code, generated by the Nail compiler, is the IGlue translation of the following Nail rules:

```
partstc(X,Y) :- assembly(X,Y,_).
partstc(X,Z) :- assembly(X,Y,_) & partstc(Y,Z).
```

These rules compute the transitive closure of the `assembly(X,Y,Q)` relation for the bill of materials program in Figure 1. The Nail compiler first transformed the rules using the context right-linear transformation. It then generated the IGlue code that performs semi-naive evaluation of the rules. The semi-naive loop can be seen in lines 10–32 of Figure 5.

The `_FORALL` instruction expresses a query in IGlue. Like Glue assignment bodies, this instruction supports negation and provides operators for updating one or more relations: ++ for insert, - - for delete, and ~~ for clear. However, unlike Glue assignment statements, IGlue `_FORALL` instructions have no "head" subgoal; all updates are performed in the `_FORALL` body. In Figure 5, lines 25–30 illustrate a `_FORALL` instruction. The first three subgoals form a query on three relations. Using variable bindings that result from that query, the fourth and fifth subgoals insert tuples into relations.

The `_EXISTS` instruction implements a special case of the `_FORALL` instruction by computing at most one solution to its arguments. It is intended to be used as a condition in one of the IGlue branching instructions, as illustrated in the `_IF` instruction in line 32. The `_EXISTS` instructions must be free of side-effects (i.e., no update operators).

The `_MOVE` instruction, illustrated in lines 19-21, performs a data movement operation. Tuples of one relation (the source) are moved to a second relation (the target), first deleting any previous contents of the target. After the move, the source relation is empty. A similar operation, `_TRANSFER`, adds new tuples to the target without clearing it first. These operators are implemented efficiently by copying pointers to sets of tuples, instead of copying each tuple.

The Glue language offers a uniform syntax for referring to relations, Nail rules, and procedures. For example, the Glue assignment statement in Figure 1, line 25 refers to input relation `in(_)` and Nail rule `partstc(_,_)`. IGlue, on the other hand, treats procedure calls separately. Calls to IGlue procedures, which represent compiled Glue procedures or Nail queries, are explicit. The syntax of a procedure call is as follows:

```
_CALL(initial-bindings,_PROC(call,[possible-procedures]),
   final-bindings)
```

The first argument is a relation that holds the bindings for variables before the procedure call. The third argument holds the bindings after the call. The second argument gives the call with its arguments, and a list of all the possible procedure references. Each entry in the list includes the module name, the procedure name,

and the pattern of input and output arguments. For example, the IGlue call to Nail rule `partstc(_,_)`, with the first argument bound and the second argument free, is the following:

```
_CALL(
  _IN(in(bom(Root))),
  _PROC(nail_bill_partstc_bf_true(Root, P),
  [nail_bill_partstc_bf_true:nail_bill_partstc_bf_true/1:1]),
  _TEMP( proc_sup4(P)))
```

There can be more than one possible procedure referent because the procedure call might have a variable for its name. Thus we cannot know which procedure to call until the variable is bound at run time. Procedure calls also require that relations be materialized to hold the set of variable bindings immediately before and after the call. So procedure calls are expensive to set up, execute, and recover from.

In the example IGlue code in Figure 5, and in the `_CALL` example above, we see that relation and procedure predicates are annotated with predicate class descriptors: `_EDB`, `_LOCAL`, `_TEMP`, `_IN`, `_OUT`, and `_PROC`. There are additional descriptors, not shown, for Nailog predicate names that contain variables, and therefore refer to sets of relations or procedures. As described in Section 4, the Glue compiler analyzes the type of each predicate and determines the set of potential referents for each predicate as early as possible.

### 6.2 The IGlue Interpreter

The IGlue interpreter is divided into three functional components: The relation manager, the abstract machine, and the run-time optimizer. The relation manager is responsible for loading EDB relations into main memory, for creating and maintaining indexes, and for accessing and updating relations. The abstract machine executes IGlue instructions. Below we describe how it processes the `_FORALL` instruction. The run-time optimizer (Section 6.3) adapts query execution plans to changing database parameters.

*Join Processing.* The IGlue `_FORALL` statement can be viewed as a join expression. In the IGlue abstract machine, `_FORALL` statements are evaluated using the nested-loop join algorithm with hash indexes on join and selection arguments. Thus, a join of $n$ subgoals is evaluated using $n$ nested loops. Variations of this join method have been used in other systems for processing joins in main memory (e.g., Whang and Krishnamurthy, 1990). The query processor assumes that the run-time optimizer has selected access methods (index or scan) for each relation and an order for computing a sequence of joins. During the nested-loop computation, the query processor interacts with the relation manager to resolve incomplete relation bindings, obtain access paths, fetch tuples, and update relations. For example, the query in lines 34–37 of Figure 5, is computed by the following steps (assuming that the query optimizer does not reorder subgoals):

```
Open an access path p1 to relation context_magic(VV1,X);
For each tuple obtained from p1 begin
  Record the bindings for variables X and VV1;
  Open an access path p2 to relation assembly(X,Y,_);
  For each tuple obtained from p2 begin
    Record the binding for variable Y;
    Insert tuple (VV1,Y) into relation answer(VV1,Y);
  end;
end;
```

If the optimizer has chosen an index on the first argument of relation `assembly(X,Y,_)` as the access path, then the join processor will direct the relation manager to create the index if it does not already exist. Although the example above is shown as a nested loop, the join processing algorithm is actually implemented as a recursive procedure that can handle joins of an arbitrary number of subgoals. The `_EXISTS` join is handled in a similar manner except that when the first result tuple is generated the process terminates.

### 6.3 The Run-Time Optimizer

The IGlue interpreter employs a run-time optimizer that accommodates dynamic characteristics of IGlue programs in two ways: (1) It reoptimizes query plans adaptively; (2) It selects indexes dynamically. We describe each of these optimizer tasks below.

*Reoptimizing Queries.* Query optimization is the problem of formulating an efficient plan to evaluate a declarative query. This consists of ordering multiple join operations and selecting among available access paths for each relation. Optimizers estimate the cost of alternative query plans and select the plan with the lowest cost. Cost estimates are derived using a cost model of join processing and statistical parameters that characterize the relations involved in the query. In relational database systems, query optimization is typically done at compile time. Some characteristics of IGlue programs are problematic for conventional query optimization techniques. In particular, IGlue programs refer to temporary relations more often than persistent relations. Because the parameters (e.g., cardinality and domain size) of temporary relations are not known until run time, it is difficult for a static, compile-time optimizer to predict which query plan is most suitable. Moreover, because relations are updated frequently in IGlue, their parameters change at run time and a query plan that was optimal early in the computation may perform poorly later. To handle this problem, the IGlue interpreter provides a run-time-optimizer that reoptimizes queries whenever there is a significant change in relation cardinality. Optimizers for some relational database systems will automatically invalidate queries based on schema or index changes (Date, 1986). However, these systems do not reoptimize queries when statistics change to avoid the overhead costs of optimization. Our results in Section 7 show that for Glue-Nail programs, however, the benefit of

reoptimization is worth the overhead cost.

The IGlue run-time optimizer is based on the dynamic programming approach used in System R (Selinger et al., 1979). The first time the optimizer encounters a particular query, it selects and records a join order and access paths. Thereafter, each time that query is to be executed, the optimizer must decide whether to reoptimize it. An ideal criterion would trigger reoptimization exactly when the current query plan is no longer optimal. We investigated several different criteria for deciding when to reoptimize a query based on changes in relation cardinality (Derr, 1993). The criterion that yielded the best performance on a suite of Glue programs is the following:

> Reoptimize a query when the cardinality of any relation in the query increases by a factor of $k$ or decreases by a factor of $1/k$.

For our experiments we chose the value $k = 2$. This criterion was compared against a no reoptimization strategy, a strategy that reoptimizes for every cardinality change, and a strategy that reoptimizes when the rank order of relations by their cardinalities changes. The results comparing the winning strategy with no reoptimization are reported in Section 7.2.

*Automatic Index Selection.* Index selection is the problem of determining which relation indexes to create and maintain. A common approach to index selection is for a database administrator to decide which indexes should exist, based on an expected pattern of access and update. While Glue and Nail require a programmer to declare EDB and local relations, the languages do not let the programmer define indexes on relations. Consequently, all indexing decisions are made automatically by the system. This feature is consistent with the Glue-Nail philosophy that the programmer should have to give only a declarative specification of a query. Furthermore, automatic indexing provides a way to select indexes on temporary relations that are introduced by the Glue and Nail compilers.

One approach to automatic index selection would be to try to anticipate which indexes would be the most useful. However, without knowing relation parameters and join orders, this approach could select a large superset of the indexes that are actually needed. We have chosen an alternative approach that defers index selection until run time. The optimizer treats index selection as two subproblems: (1) deciding when to create an index; and (2) deciding when to drop an index. Some relational systems are able to dynamically create temporary indexes. However, these indexes exist only for the duration of a query. The IGlue optimizer tries to be smarter by determining if each index it creates can potentially be used again later in the program.

The first subproblem, index creation, is handled when the query optimizer must choose between scanning a relation and accessing a relation via an index on all bound arguments. Suppose the optimizer determines that the best method is to access a relation using an index. If the index already exists, the optimizer chooses

the index. However, if the index does not exist, how should the optimizer count the overhead of creating, maintaining, and eventually deleting the index? Here the optimizer considers two cases. If the indexing overhead is less than the benefit of using the index the optimizer chooses to build the index. What if the indexing overhead is greater than the benefit of using the index (as will happen when n relation is scanned only once or twice)? If the query under consideration is executed only once, then the best choice is to scan the relation. However, if the index can be reused later in the program (perhaps in subsequent executions of of the same query), then the overhead costs can be amortized over all uses of the index.

We compared several approaches (Derr, 1992) and found that the most effective strategy for Glue-Nail programs was to ignore the overhead of indexing. This strategy outperformed both a strategy that counted the overhead and an online algorithm that monitored cost and benefit to determine when to switch from scanning a relation to using an index. When the ignore strategy makes the right choice, the cost benefits—avoiding multiple scans—are typically large. When it makes the wrong choice, the penalty—an extra scan—is relatively small. Furthermore, the ignore strategy has the advantage (over the online approach) of simplicity.

The second subproblem, deciding whether to drop or maintain an index, is handled whenever a relation on which an index is defined is updated. Indexes on temporary relations are dropped automatically when the temporary is deleted. Indexes on EDB relations are dropped when the program halts. However, the system may want to drop infrequently used indexes earlier to avoid the overhead of maintaining them. We compared five different strategies (Derr, 1992) that evaluate the cost of maintaining indexes and found two to be effective in the Glue-Nail benchmark programs. One strategy always maintains an index instead of dropping it. The other determines, based on definition-use information, whether an index has any potential uses. If so, the index is maintained. Both strategies assume that the future benefit of using an index will outweigh the cost of maintaining it. As it turned out, the programs in the benchmark could not differentiate between these two strategies because there were no cases in which the second strategy decided to drop an index.

By combining automatic indexing with reoptimization, the run-time optimizer is able to adapt to changes in the database. The combination enables the optimizer to choose new join orders and create new indexes as needed. Performance results presented in the next section demonstrate the advantages of adaptive optimization.

## 7. Applications, Performance, and Evaluation

A number of test applications have been written in Glue-Nail. They were written for two reasons: first, to test the practical expressiveness of the Glue-Nail system; and second, to provide a suite of programs for developing and testing the system. These 11 applications are summarized in the next section.

### 7.1 The Glue-Nail Benchmark Suite

The Glue-Nail benchmark contains 11 applications written by several different programmers. When compiled into IGlue, the applications range in size from 13 instructions (TC) to 713 instructions THOR$_p$). The number of tuples contained in EDB relations varies from 15 (CAR) to 15,100 (BILL). Some of the applications were developed before there was a working Nail compiler and thus are written entirely in Glue. Overall the benchmark suite contains 2,623 lines of Glue-Nail.

THOR The THOR application simulates a logic circuit. The application consists of two programs: THOR$_p$ parses a circuit description and converts it into a set of relations; THOR$_s$ simulates the circuit. The application creates and manipulates 10 EDB relations containing about 1,600 tuples. Together, the two programs contain 1550 lines of Glue and Nail.

BILL The BILL program (Figure 1) constructs a *bill of materials*: the quantity of each basic part required to build a complex object. The BILL program accesses an EDB containing two relations that hold a total of 15,100 tuples. It is written in 54 lines of Glue and Nail.

SG The SG program solves the *same generation* problem: find all pairs of persons at the same level in a family tree. The family tree consists of 4,798 nodes and represents eight generations, where each parent has at most five children. It is written in 19 lines of Glue and Nail.

TC The TC program finds paths in a graph using recursive Nail rules for transitive closure. The TC program was run on a variety of databases described in Section 7.2. The TC program, written in Glue and Nail, is 24 lines long.

WIN The WIN program contains a Nail rule with a negated recursive predicate and is used to demonstrate how the Nail compiler handles both modularly stratified and non-modularly stratified programs. The WIN program was run on a variety of database instances as described in Section 5. The program, written in Glue and Nail, is 25 lines long.

CIFE The CIFE program schedules tasks and allocates resources required for constructing a building. The application uses an EDB that describes an 8-floor, 16-room building. The EDB initially holds a total of 1066 tuples and grows to 2161 tuples. The CIFE program, written in Glue and Nail, is 186 lines long.

SPAN The SPAN program finds the minimum spanning tree for a set of 50 points. The SPAN program consists of 106 lines of Glue.

OAG The OAG program searches for direct and connecting flights in an airline database. The EDB defines seven relations, which hold a total of 764 tuples. The OAG program is written in 223 lines of Glue.

CAR The CAR program simulates the movement of 14 cars around a circular track. The simulation runs for 100 clock ticks. The database initially holds 14 tuples and grows to 1516 tuples. The CAR application is written in 104 lines of Glue and Nail.

CAD The CAD application is a simple 2D drafting system. This application was written for a prototype version of Glue and uses features that are not available in the current Glue-Nail system. The CAD program is written in 252 lines of Glue and Nail.

PATH The PATH program is hand-written Glue version of the transitive closure query. This program was used to experiment with different styles of Nail compilers. It has 80 lines of Glue.

These applications were chosen with an eye to being representative of the code we expect would be run in the Glue-Nail system. The THOR programs are intended to represent a complete application. BILL, SG, OAG, and TC are typical deductive database programs that depend on recursion. The remaining applications exercise a variety of features of the Glue and language.

### 7.2 Performance Results

To demonstrate the advantage of the adaptive optimization techniques described in Section 6.3, let us look at some performance results for the TC application. Consider the following Nail rules for transitive closure:

```
tc(X,Y) :- arc(X,Y).
tc(X,Y) :- arc(X,Z) & tc(Z,Y).
```

and the query $\texttt{tc(X,Y)}^{bf}$. The IGlue code that implements this Nail program is almost identical to that shown in Figure 5 for the transitive closure rules in the bill-of-materials example.

To measure program execution time for databases of different sizes and shapes, we prepared two different `arc(X,Z)` relations. The first relation represents a cyclic linear list using tuples of the form: (1,2),. . . ,(*N*-1,*N*), (*N*,1). The second relation represents a complete binary tree of height *H*. We compared the performance of two versions of the run-time optimizer. The first version (adaptive) reoptimized queries using the strategy described in Section 6.3. The second version (nonadaptive) optimized each query only once. In both cases the optimizer performed automatic index selection. The programs were executed on a SPARC 10/41 with 128 megabytes of memory and running SunOS 4.1.3, as were the benchmarks described below.

**Table 6. Execution time (seconds) for transitive closure on cyclic linear lists.**

| $N$ | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 2048 | 4096 |
|---|---|---|---|---|---|---|---|---|---|
| Adaptive | 0.03 | 0.02 | 0.04 | 0.06 | 0.15 | 0.26 | 1.12 | 2.05 | 4.39 |
| Nonadaptive | 0.01 | 0.01 | 0.03 | 0.08 | 0.21 | 0.62 | 6.99 | 26.21 | 102.49 |

**Table 7. Execution time (seconds) for transitive closure on complete binary trees**

| $H$ | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|
| Adaptive | 0.02 | 0.03 | 0.04 | 0.10 | 0.18 | 0.37 | 0.69 | 1.37 | 2.89 |
| Nonadaptive | 0.02 | 0.03 | 0.05 | 0.14 | 0.40 | 1.28 | 4.52 | 17.02 | 65.77 |

**Table 8. Execution time (seconds) for the Glue benchmark applications**

| Program | BILL | CIFE | SG | SPAN | CAR | OAG | $\text{THOR}_p$ | $\text{THOR}_s$ |
|---|---|---|---|---|---|---|---|---|
| Adaptive | 6.04 | 14.43 | 11.13 | 17.99 | 61.86 | 26.10 | 30.33 | 16.31 |
| Nonadaptive | 18.54 | 21.68 | 104.86 | 22.31 | 130.71 | 26.76 | 30.71 | 19.22 |

Tables 6 and 7 show the execution times, which include the cost of optimization. These results clearly demonstrate the advantage of the adaptive query optimizer. In the adaptive case, the growth of the evaluation times for linear and binary data is almost linear in the size of the `arc(X,Z)` relation. Note that the number of tuples representing a complete binary tree with height $H$ is $2^{H+1}$-1. When we examine optimization traces, we find that the difference in performance occurred because the adaptive optimizer, when it was reoptimizing a query, was able to create a new index that wasn't needed in previous executions of the query.

We also compared the performance of adaptive and nonadaptive optimization on other programs from the Glue-Nail benchmark suite. The executions times, which include optimization costs, are presented in Table 8. Here we see that for six of the programs, execution time was significantly faster using adaptive optimization. Although the adaptive techniques did not improve performance for all programs, neither did they degrade performance for any program.

### 7.3 Comparing Glue with C

It is all very well to think up small example programs, and to code them up; but such programs are artificial examples. Unless we take a real application and code it up, we do not know how the language will perform in practice. Therefore it was decided

to take an existing application, code a section of it in Glue, and compare it with a version coded in C. The THOR circuit simulator (Alverson et al., 1988) was chosen as an example application. We chose THOR because it is an application for which deductive databases are well suited, and because much of THOR's development took place at the Computer Systems Laboratory at Stanford, which gave us access to the original C source code, and to many examples.

A problem with this experiment is that THOR, like any real application, is very large. Hence a suitable subsection of THOR's functionality was chosen for implementation. The choice of what to implement is important. To be a fair test, the subsection chosen must include code representative of all of the problems found in THOR. The Glue version of THOR (Glue-THOR) implements the Component Simulation Language (CSL) of THOR. CSL is a gate and net descriptor language. The Glue code includes a CSL parser ($THOR_p$) and a CSL circuit simulator ($THOR_s$). They are run as separate passes. The parser is slow, because it works with a single tuple at a time. Deductive databases systems, such as Glue-Nail, are designed to work with large sets of data.

Glue-Nail is intended to be a prototyping or niche market language. For these applications there is a strictly limited amount of time available for coding. Glue-Nail emphasizes speed of coding over speed of execution. In this respect it is similar to Prolog, Lisp, Scheme, etc. There are situations when one doesn't have the resources (time and people) to write a full C[5]-SQL implementation. These are the situations for which Glue-Nail is expected to be especially well suited.

Designing a proper experiment to test whether we have achieved the goal of increased programmer productivity would be difficult. One approach would be to pick a suitable problem and then assign it to two groups of programmers. One group would work in Glue-Nail and the other group in a standard language. Such an experiment was outside the scope of this work. The THOR application is merely a preliminary exploration of the field, and by no means a real experiment. However, very few experiments have been conducted on the practical utility of deductive database systems.[6] Programming languages need to be used for their strengths and weaknesses to appear, and deductive databases have primarily been used for research examples.

The Glue implementation of the THOR subset has 1,550 lines (including comments), and took approximately one month to write. Much of the code is occupied by the recursive descent parser. While the C version of THOR uses lex and yacc, the Glue version had to write its own lexical analyzer and parser. The net-list code section of C-THOR code is 9,702 lines long (including comments). As a ratio of lines of code, this is 9,702/1,5650 or roughly 6:1. Given that author of the Glue

5. Or pick your favorite general purpose host language.

6. For a good selection, see Ramakrishnan (1993).

version of THOR can write 1,000 lines of *debugged* C code per month, the ratio of coding time is (9.7 months)/(1 month) or roughly 10:1.

Execution times also were compared. A standard counter-adder example in the THOR distribution was used. The Glue implementation was found to run approximately 100 times more slowly than the C implementation, a disappointing result, but not surprising given that C is compiled and Glue is interpreted.

### 7.4 Evaluating the Glue Language Design

Programming languages are not works of art to be admired, they are tools to be used. Their utility can be judged only by using them for their designed purpose. Accordingly, approximately 3,000[7] lines of Glue-Nail application code were written by several different authors. No formal usability experiments have been conducted, but some general observations have been drawn. In a sense these observations are the results of an experiment in language design. These observations are presented below.

*The Good.* The major benefit that Glue coders experienced was the removal of the barrier between the program and the database. In a conventional two-language system, the computation is performed in some system language (like C), but the data are stored in the database. Attaching to the database and pulling the data over is tedious and error prone.

The implicit looping inherent in the set-oriented semantics of Glue was perceived to be an advantage. Being able to deal with many elements simultaneously without using multiply-nested loops was a useful simplification.

The high level nature of Glue was also advantageous, but no more so than in Prolog. The major advantage over C is that programmers no longer have to worry about pointers, which are definitely the most bug-inducing part of C. Unification of compound terms is a very simple way of handling complex data structures. Glue only has matching, not full unification, but this restriction was not found to be a problem.

Nailog was especially useful in dealing with user defined complex graphical objects in the CAD example.

*The Bad.* The major problem with Glue is the lack of a type system. This "feature" was inherited from the logic programming paradigm. Type systems provide a scaffolding for programmers, which is crucial if large systems are to be maintained. Type systems are most useful when the program structure is too large to carry inside one person's head, or when a programmer is maintaining code that was written by someone else. Small programs can always be written without type errors because they can be understood as a whole.

---

7. The benchmark suite, plus another 1,000 lines.

A type system could easily be added to Glue. A fairly simple scheme that allowed strings, numbers, and compound terms would be enough.

A related problem is the declaration of EDB relations. In the present design of Glue, each module defines its own set of EDB relations. It would probably be better to have a special kind of module that defines the EDB, and let the other (code-only) modules refer to this EDB module.

The other problems were much more minor. Some programs needed counters (e.g., loop counters). The syntax and semantics of Glue are directed towards large relations, so such counters are cumbersome. A way of defining constants (like $\pi$) is needed. A scheme as simple as the `#define`'s of C would solve most of these problems.

Iteration in Glue procedures can be implemented either as recursion or as looping. Programmers found that recursion in Glue behaved a little strangely, and felt more comfortable with `repeat` loops. In addition, procedure calls are expensive in Glue, so loop code is more efficient than recursive code. Perhaps tail recursion optimization could remove this efficiency difference.

Occasionally, it was necessary to operate on the tuples in a relation in some particular order (e.g., printing the tuples of a relation in alphabetical order. It is cumbersome to do this in the current version of Glue; either a `repeat` loop or recursion is needed). By pushing Glue towards set-oriented semantics, it has become difficult to work in a tuple-at-a-time fashion. It might be better to add `sort` as another aggregation operator. This operator would sort a set of tuples and ensure that all side effecting operations use that order.

The scope system worked moderately well, but there were problems with dynamic binding of predicate names. Consider the general purpose transitive closure predicate:

```
path(N,E,X,X):- N(X).
path(N,E,X,Z):- E(X,Y) & path(N,E,Y,Z).
```

The first two arguments of `path(_,_,_,_)` are predicate names and will be understood in the scope environment $S_1$ of `path`. There are no problems if `path` is called from within scope $S_1$, but what if it is called from some other scope $S_2$? The passed arguments from scope $S_2$ for $N$ and `E` may make no sense in scope $S_1$. For this reason the "hat" operator `^` had to be added to Glue-Nail. The hat operator is a run-time operator that means "dereference this term in the current scope, and pass the referent." So a call to `path` from outside of `path`'s module would look like `path(^node(_),^edge(_,_), X, Y)`. The hat operator was rarely used by programmers, because it was rarely needed and difficult to understand.

Another way to handle the imported predicate problem would have been to mark the signature of `path`, indicating the arguments that could be predicate references. In this scheme the caller would not have to provide a marking on each call, the compiler would be able to insert it automatically. This design was not used because it is the beginnings of a type system, and we did not want to worry about type systems. As was mentioned above, it was a mistake not to have a type system.

## 8. Related Work

Glue-Nail can be compared to several other experimental database languages and systems. While these systems vary along several dimensions—language philosophy, multi- versus single-user, disk- versus memory-residency, and hardware platforms—they are all based on a deductive approach. There are no benchmarks available with which to compare the performance of these systems, although various groups are beginning to work on such benchmarks. Many of these systems have their own private benchmarks.

LDL (Naqvi and Tsur, 1989; Chimenti et al., 1990) is a main memory, single user deductive database system developed at MCC. Unlike Glue-Nail, LDL handles both declarative queries and procedural operations in the same language. Hence some rules in an LDL program must be read procedurally. Rules are compiled into an AND/OR graph representing joins and unions. The system allows programmers to define indexes on base relations. If no index is declared for a relation, then by default, the system builds an index on the first argument. The LDL optimizer chooses join orders and annotates the graph with access methods. The graph is then translated into a C program which makes calls to an underlying database management system. The decisions made by the optimizer are hardwired into the target code. Thus, unlike Glue-Nail, query execution plans are not able to adapt at run time.

CORAL (Ramakrishnan et al., 1992) is a database system prototype developed at the University of Wisconsin-Madison. Like Glue-Nail, CORAL employs a two-language paradigm. The declarative language, which is similar to LDL, is based on Horn clauses with extensions for handling left-to-right modularly-stratified negation, non-ground facts, set and multi-sets, and aggregation. The imperative language is C++ extended with a relation and tuple class library. Using annotations, the user can control the evaluation of CORAL in various ways, such as indexing relations, choosing only one answer, and prioritizing execution paths for aggregations. By default, CORAL selects a left to right join order. For semi-naive evaluation, CORAL uses a heuristic that moves any delta predicates to the left. The user can also specify join order for each (rewritten) rule. CORAL's optimization philosophy contrasts sharply with that of Glue-Nail. Using CORAL, a programmer must understand how and when to use a variety of optimization strategies. Using Glue-Nail, the programmer is not allowed to control optimization. All strategy selection is automatic.

Aditi (Vaghani et al., 1990) is a multi-user, disk-based, deductive database system developed at the University of Melbourne. Aditi's language philosophy is that applications and queries should be written in a single logic-based language. Aditi queries can be embedded in Nu-Prolog (Thom and Zobel, 1990), which serves as the procedural support language. They did not create a new language like Glue. Aditi programs are written in a variant of Prolog augmented with mode declarations, compiler directives for specifying evaluation strategies, and well-founded negation (Kemp et al., 1992). The mode declarations specify which adornment patterns

are legal for a predicate. Nail does not have this restriction, although built-in Nail predicates (such as arithmetic) do have limited modes. Aditi programs are compiled into a low level procedural relational language called RL. RL programs are assembled into bytecodes and interpreted by the database back end. Because RL supports only binary join operations, the join order for multi-joins must be determined at the time the RL code is generated. This differs from the IGlue interpreter, which can adaptively optimize the join order for joins with up to fifteen relations.

The EKS-V1 system (ECRC Knowledge Base System) (Vieille et al., 1990) is similar to the Aditi system in that a pure logic query language is embedded in a variant of Prolog called MEGALOG. MEGALOG, designed to handle large numbers of facts efficiently, is based on the BANG file system (Freeston, 1987) for storing both facts and code. Unlike IGlue, Megalog can use secondary storage. The execution strategy for EKS rules is Query-Subquery (QSQ) (Vieille, 1986), which is a "top-down" method. Nail uses a "bottom-up" method. The EKS-V1 system has a static (compile-time) optimizer, which chooses the join order, and identifies common subexpressions and tail recursion. Query evaluation is performed by BANG, which uses relation deltas to avoid repeatedly joining the same tuples.

The LOLA system (Freitag et al., 1991) is a deductive database system designed and implemented at the Technische Universität of Munich. The LOLA language offers clear declarative semantics based on minimal model semantics. LOLA programs are evaluated using semi-naive fixpoint iteration. The LOLA system is implemented in CommonLisp and provides an interface to an external relational DBMS as well as to a Lisp-based main memory database. Like Glue-Nail, LOLA provides multiple levels of optimization. The source-level optimizer performs selection propagation, magic sets transformation, and projection optmization. The operator graph optimizer detects common subexpressions and selects indexes. However, unlike Glue-Nail, there is no fully automatic control for the selection and ordering of optimizations.

We also compare the Glue language to other procedural languages that support relations or sets. Pascal/R (Schmidt, 1977) was an early attempt to reconcile databases with procedural languages (in this case Pascal). Relations (of Pascal records) were added as a fundamental data type. A looping construct over relations was provided; it was more powerful than a simple cursor into a relation, but less powerful than an SQL join operation. Hence, it accessed relations at a lower level than either SQL, Glue-Nail, or any of the other systems mentioned above.

The proposed SQL3 standard (Melton, 1993) is expected to include a control language for the implementation of abstract data types. This language includes procedures, assignment statements, case and if-then-else statements, and looping constructs. This extension will provide some of the advantages of object-oriented database programming languages. Programmers may still embed SQL in host programming languages. However, the control language makes it possible to write procedural code that is handled directly by an SQL server, just as Glue code is handled directly by the IGlue interpreter.

## 9. Conclusion

We have presented an overview of the design and implementation of the Glue-Nail database system. We began by describing features of the Glue-Nail language pair. The declarative features of Nail and the procedural features of Glue combine to enable a programmer to write complete applications. We also described the architecture and implementation of the system. In particular, we focused on the optimization techniques. The Nail compiler selects appropriate transformation and evaluation strategies based on syntactic properties of the Nail program. The Glue compiler, after generating target IGlue code, performs static code optimizations using peephole techniques and data flow analysis. The IGlue interpreter optimizes queries at run time to adapt query execution plans to dynamic database parameters. The combination of these optimization techniques results in a system that executes Glue-Nail programs efficiently.

We then described some applications that demonstrate feasibility of Glue-Nail as a programming language for writing complete database applications. With the THOR application, we demonstrated the programmer productivity advantage of the Glue language. The applications also served as a benchmark for testing the effects of our optimization strategies.

The Glue-Nail system could be improved in several ways. Glue and IGlue both support Nailog and function symbols. However, the current Nail compiler supports only Datalog with negation, and needs to be extended to handle function symbols and Nailog. The performance of the system could be enhanced by providing the Nail compiler with additional strategies for evaluating special cases of Nail programs. Performance could also be improved by executing compiled code instead of interpreting IGlue. A compiled system would have to be able to dynamically recompile query plans selected by the adaptive optimizer.

## Acknowledgments

Geoffrey Phipps was supported by grants AFOSR-88-0266, IST-87-12791, AFOSR 90-0066, and Army DAAL03-91-G-0177. We would like to acknowledge the contributions of Ashish Gupta, Kate Morris, and Ken Ross, who developed code used in the previous and current versions of the Nail compiler. Kathleen Fisher wrote the CAR application. David Chang wrote a statistical package in Glue. Ashish Gupta and Sanjai Tiwari wrote the CIFE application. We are grateful to Jeff Ullman for his comments on earlier versions of this article.

## References

Alverson, R., Blank, T., Choi, K., Hwang, S.Y., Salz, A., Soule, L., and Rokicki, T. THOR user's manual: Tutorial and commands. Technical Report CSL-TR-88-348, Computer Systems Laboratory, Stanford University, 1988.

Bancilhon, F. Naive evaluation of recursively defined relations. In: Brodie, M.L. and Mylopoulos, J., eds. *On Knowledge Base Management Systems*, New York: Springer-Verlag, 1986, pp. 165-178.

Beeri, C. and Ramakrishnan, R. On the power of magic. *Proceedings of the 1987 ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, San Diego, California, 1987.

Chen, W., Kifer, M., and Warren, D.S. HiLog: A first-order semantics of higher-order logic programming constructs. *Logic Programming: Proceedings of North American Conference*, Cleveland, 1989.

Chimenti, D., Gamboa, R., and Krishnamurthy, R. Abstract machine for LDL. *Proceedings of the International Conference on Extending Database Technology*, Venice, Italy, 1990.

Date, C.J. *An Introduction to Database Systems*, vol. 1. Reading, MA: Addison Wesley, 1986.

Derr, M.A. Adaptive optimization in a database programming language. PhD thesis, Department of Computer Science Report No. STAN-CS-92-1460, Stanford University, Stanford, California, December 1992.

Derr, M.A. Adaptive optimization in a deductive database system. *Proceedings of the Second International Conference on Information and Knowledge Management*, Washington, DC, 1993.

Derr, M.A., Morishita, S., and Phipps, G. Design and implementation of the Glue-Nail database system. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, 1993.

Freeston, M. The BANG file: A new kind of grid file. *Proceedings of the ACM SIGMOD Annual Conference*, San Francisco, CA, 1987.

Freitag, B., Schütz, H., and Specht, G. LOLA—a logic language for deductive databases and its implementation. *Proceedings of the Second International Symposium on Database Systems for Advanced Applications*, Tokyo, Japan, 1991.

Kemp, D.B., Ramamohanarao, K., and Somogyi, Z. Right-, left-, and multi-linear rule transformations that maintain context information. *Proceedings of the Sixteenth International Conference on Very Large Databases*, Brisbane, Australia, 1990.

Kemp, D.B., Stuckey, P.J., and Srivastava, D. Query restricted bottom-up evaluation of well-founded models. *Proceedings of the 1992 Joint Conference and Symposium on Logic Programming*, Washington DC, 1992.

Kerisit, J.-M. and Pugin, J.-M. Efficient query answering on stratified databases. *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, 1988.

Lloyd, J.W. *Foundations of Logic Programming*. New York: Springer-Verlag, 1984.

Makinouchi, A. A consideration on normal form of not-necessarily-normalized relations in the relational data model. *Proceedings of the International Conference on Very Large Data Bases*, Tokyo, Japan, 1977.

Melton, J., ed. *ISO-ANSI Working Draft Database Language SQL (SQL3)*, 1993. ANSI X3H2-93-091 and ISO DBL-YOK 003.

Morishita, S. An alternating fixpoint tailored to magic programs. *Proceedings of the 1993 ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, Washington DC, 1993.

Morris, K., Naughton, J.F., Saraiya, Y., Ullman, J.D., and Van Gelder, A. YAWN! (Yet Another Window on NAIL!). *Data Engineering*, 10(4):28–43, 1987.

Morris, K., Ullman, J.D., and Van Gelder, A. Design overview of the NAIL! system. *Proceedings of the International Conference on Logic Programming*, Location? 1986.

Mumick, I.S. and Pirahesh, H. Extending the right-linear transformation. Research Report RJ 7938, IBM Research Division, Computer Science, Almaden Research Center, January 1991.

Naqvi, S. and Tsur, S. *A Logical Language for Data and Knowledge Bases*. New York: Computer Science Press, 1989.

Phipps, G. Glue: A deductive database programming language. *Proceedings of the NACLP Workshop on Deductive Databases*. Kansas State University Technical Report TR-CS-90-14, 1990.

Phipps, G. Glue: A deductive database programming language. PhD thesis, Department of Computer Science Report No. STAN-CS-92-1437, Stanford University, Stanford, California, July 1992.

Phipps, G., Derr, M.A., and Ross, K.A. Glue-Nail: A deductive database system. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, Colorado, 1991.

Ramakrishnan, R. Magic templates: A spellbinding approach to logic programs. *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, Seattle, WA, 1988.

Ramakrishnan, ed., *Proceedings of the Workshop on Programming with Logic Databases*, Vancouver, BC, Canada, October 1993. Computer Sciences Department Technical Report #1183, University of Wisconsin-Madison.

Ramakrishnan, R., Srivastava, D., and Sudarshan, S. CORAL: Control relations and logic. *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, Vancouver, Canada, 1992.

Ross, K. Modularly stratification and magic sets for datalog programs with negation. *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, Nashville, TN, 1990.

Ross, K. Modular acyclicity and tail recursion in logic programs. *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, Denver, CO, 1991.

Schmidt, J.W. Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261, 1977.

Selinger, P.G., Atrahan, M.M., Chamberlin, D.D., Lorie, R.A., and Price, T.G. Access path selection in a relational database management system. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Washington, DC, 1979.

Thom, J.A. and Zobel, J. Nu-Prolog Reference Manual, version 1.5.24. Technical Report 86/10, Department of Computer Science, University of Melbourne, Australia 1990.

Ullman, J.D. *Principles of Database and Knowledge-Base Systems*, vol. 2. Rockville, MD: Computer Science Press, 1989.

Vaghani, J., Ramamohanarao, K., Kemp, D.B., Somogyi, Z., and Stuckey, P.J. The Aditi deductive database system. *Proceedings of the NACLP Workshop on Deductive Databases*. Kansas State University Technical Report TR-CS-90-14, 1990.

Van Gelder, A. The alternating fixpoint of logic programs with negation. *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, Philadelphia, PA, 1989.

Vieille, L. Recursive axioms in deductive databases: The query/sub-query approach. *Proceedings of the First International Conference on Expert Database Systems*, Charleston, SC, 1986.

Vieille, L., Bayer, P., Küchenhoff, V., and Lefebvre, A. EKS-V1, a short overview. *AAAI Workshop on Knowledge Base Management Systems*, Boston, MA, 1990.

Whang, K.-Y. and Krishnamurthy, R. Query optimization in a memory-resident domain relational calculus database system. *ACM Transactions on Database Systems*, 15(1):67–95, 1990.

Wiederhold, G. Views, Objects, and Databases. *IEEE Computer*, 19(12):37–44, 1986.