Design and Implementation of the Glue-Nail Database System*

Marcia A. Derr
AT&T Bell Laboratories

Shinichi Morishita
IBM Tokyo Research Laboratory

Geoffrey Phipps†
Sun Microsystems Laboratories

Abstract

We describe the design and implementation of the Glue-Nail database system. The Nail language 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 both compile into the target language IGlue. The Nail compiler uses variants of the magic sets algorithm, and supports well-founded models. Static optimization is performed by the Glue compiler using techniques that include peephole methods and data flow analysis. The IGlue code is executed by the IGlue interpreter, which features a run-time adaptive optimizer. The three optimizers each deal with separate optimization domains, and experiments indicate that an effective synergism is achieved. The Glue-Nail system is largely complete and has been tested using a suite of representative applications.

1 Introduction

The Glue-Nail database system [15] provides two complementary languages for programming deductive database applications. The Glue procedural language [14] augments relational-style queries with control structures, update operations, and I/O. The Nail declarative language [11, 12] provides rules for expressing complex recursive queries or views.

The purpose of this paper 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 for each major component of the system. We present empirical results that demonstrate the synergetic effects of these optimizations.

We begin by reviewing the background and philosophy underlying the design of the Glue-Nail system. Glue-Nail evolved from the first Nail system. Logic-based query languages such as Nail have proven to be powerful query languages. However, being based on logic is both a strength and weakness. It is a strength, because logic is declarative and side-effect free. By declarative we mean that although the query semantics are well defined, the actual evaluation method is unspecified. Queries can therefore be expressed clearly and optimized easily. The logical basis is 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 needed procedural operations, yet these very same operations were 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 (such as C, C++ , and Prolog). Glue [14] 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++ is. 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 [3]). 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. In almost all other logic-based languages, the predicate name must be known at compile time. The Nailog system gives the programmer additional power and flexibility that has been found to be useful in actual practise.

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. The back end interpreter generated SQL calls to an underlying commercial relational database. Typical Nail programs were compiled into code that created many temporary relations. These temporary relations were managed by the the same facilities that managed persistent shared relations on disk. Consequently, temporary relations, which are often small, short-lived, and did not need to be shared, incurred the same overhead as persistent relations. Another problem was that

*This work was done at the Department of Computer Science, Stanford University.

†Supported by IST-87-12791, AFOSR-88-0266, AFOSR-90-0066, and Army DAAL03-91-G-0177.

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0147...$1.50

the configuration was too slow. It involved multiple levels of interpretation and retrieved answers one tuple at a time.

A solution to the architecture problems 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 (pronounced "igloo"). IGlue code is directly 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.

2 Overview

In this section we present an overview of the Glue-Nail system architecture and provide a simple example of how an application is compiled and run under the system. We also introduce the optimization problems we encountered during the design of the system and the approaches we took to solve them.

2.1 The Glue-Nail System

The Glue-Nail system consists of the Glue compiler, the Nail compiler, the static optimizer, the linker, and the IGlue interpreter. The Glue-Nail source is split into its Glue and Nail text fragments, which are compiled by the appropriate compiler. In both cases IGlue code is produced. This IGlue code is statically optimized and linked. The final code is executed by the IGlue interpreter.

Let us illustrate the process of compiling and executing a Glue-Nail program with a simple example. Figure 1 shows a Glue-Nail code module that defines a program to compute the transitive closure of a binary relation. The module called `tc` begins by exporting the `main(:)` procedure outside the module, and by importing procedure `read(:X)` from a predefined `io` module. In procedure declarations and definitions, input and output arguments are separated by a colon (`:`). Thus, procedure `main(:)` has no input or output arguments, and procedure `read(:X)` has no input arguments and one output argument. The module also declares that it will access extensional database (EDB) relations `arc(X,Y)`, and `answer(X)`.

Next is the definition of Glue procedure `main(:)`. The body of the procedure contains one assignment statement. The statement calls the input procedure `read(S)`, which provides a binding for variable `S`. This binding is passed to the predicate `tc(S,X)`, which is defined by Nail rules. A reference to a Nail rule is a Nail query. This Nail query invokes the rule with the first argument bound and the second argument free, indicated with the annotation *bf*. The result of the join provides a set of bindings for variable `X`, which is used to construct a set of tuples for assigning to the `answer(X)` relation.

The rules that define the `tc(X,Y)` relation are shown at the end of the module. The first rule says that if `(X,Y)` is an arc then it is in the transitive closure. The second rule says that if `(X,Z)` is in an arc, and we can prove that `(Z,Y)` is in the transitive closure, then `(X,Y)` is in the transitive closure.

```
module tc;
export  main(:);
from    io import read(:X);
edb     arc(X,Y), answer(X);

prog main(:)
  answer(X):= read(S) & tc(S,X).
end
tc(X,Y):- arc(X,Y).
tc(X,Y):- arc(X,Z) & tc(Z,Y).
end
```

Figure 1: Glue-Nail program for transitive closure.

The Glue compiler separates the Glue code from the Nail rules in module `tc`. It passes the Nail rules and Nail query $\texttt{tc(S,X)}^{bf}$ to the Nail compiler, and compiles the Glue code into the IGlue target language. The Nail compiler transforms the Nail rules and query into an IGlue procedure that computes the answer to the query, shown in Figure 2.

The IGlue code that is generated by either compiler is optionally analyzed and transformed by the static optimizer. The linker collects all relevant IGlue code into a single file which can be executed by the IGlue interpreter.

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

2.2 Optimization Issues

To build an efficient Glue-Nail system we included optimization at multiple points in the architecture: the Glue compiler, the Nail compiler, and the IGlue interpreter.

2.2.1 Compiling Nail Rules

As a way to optimize recursive query evaluation, the Nail compiler applies one of two variants of the magic-sets transformation to the Nail program and query. The compiler then chooses a strategy to evaluate the transformed program. The Nail compiler does not choose join orders or select indexes for each IGlue query that it generates; this is left to the IGlue interpreter.

Another important feature of the Nail compiler is its ability to handle fully general Datalog programs with negation. An earlier version of the Nail compiler, developed for the Glue-Nail system, could handle only a subclass called modularly stratified programs. In the general setting, however, the magic-sets transformation may not compute the correct answer. To solve this problem we created a novel method obtained by tailoring Van Gelder's alternating fixpoint technique [21] to magic programs so that

```
_MODULE nail_tc_bf
  _EDBDECL arc/2

  _PROCEDURE nail_tc_bf/1:1
  _LOCALDECL answer/2, changed/0,
    context_magic/2, delta_context_magic/2,
    old_delta_context_magic/2

  _FORALL(
    _IN(in(nail_tc_bf(VV1))),
    ++_LOCAL(delta_context_magic(VV1,VV1)))
repeat:
  _FORALL(
    _LOCAL(delta_context_magic(VV2,VV1)),
    ++_LOCAL(context_magic(VV2,VV1)))
  _MOVE(
    _LOCAL(delta_context_magic(VV2,VV1)),
    _LOCAL(old_delta_context_magic(VV2,VV1)))
  _FORALL(~~_LOCAL(changed))
  _FORALL(
    _LOCAL(old_delta_context_magic(VV1,X)),
    _EDB(arc(X,Z)),
    !_LOCAL(context_magic(VV1,Z)),
    ++_LOCAL(delta_context_magic(VV1,Z)),
    ++_LOCAL(changed))
  _IF _EXISTS(_LOCAL(changed)) _GOTO repeat
  _FORALL(
    _LOCAL(context_magic(VV1,X)),
    _EDB(arc(X,Y)),
    ++_LOCAL(answer(VV1,Y)))
  _FORALL(
    _IN(in(nail_tc_bf(VV2))),
    _LOCAL(answer(VV2,VV1)),
    ++_OUT(out(nail_tc_bf(VV2),out(VV1))))
  _RETURN
```

Figure 2: IGlue code generated for Nail rules.

it effectively restricts the search space and computes the correct answer to the query. We will further describe the Nail query optimization strategies in Section 4.

2.2.2 Improving Code

The Glue compiler is responsible for static optimization of Glue programs, while the IGlue interpreter is responsible for dynamic optimization. Accordingly the Glue compiler tries to reduce the number of IGlue operations and relations. 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. We will elaborate on these compiler optimization strategies in Section 5.

2.2.3 Optimizing Queries

Query optimization is the problem of formulating an efficient plan to execute a declarative query. Query optimizers estimate the cost of alternative plans, based on parameters of the relations involved in the query, and choose the plan with the lowest cost. In IGlue programs, temporary relations are referenced more frequently than persistent relations. Because temporary relations are not instantiated until run time, they cause problems for conventional query optimizers. One solution is to defer query optimization until run-time, when the parameters of temporary relations are known. However, optimizing queries just once at run-time may still not be sufficient. Queries that access temporary relations may be embedded in loops or procedures that are called repeatedly. Because temporary relations are frequently updated, their parameters may change and invalidate a query plan that was chosen earlier. To handle this problem, the IGlue interpreter provides a run-time optimizer that reoptimizes queries whenever there is a significant change in relation cardinality.

The run-time optimizer also selects indexes automatically. It dynamically determines which indexes to build and maintain for both temporary and persistent relations. The IGlue interpreter and its adaptive optimizer will is described in more detail in Section 6.

3 The IGlue Target Language

IGlue is the target language for both Glue and Nail. A complete description of the language is found in [4]. Here we explain particular features of the language that are relevant to discussions later in the paper.

Several kinds of IGlue instructions are illustrated in the code in Figure 2. The _FORALL statement expresses a query in IGlue. It also provides a negation operator, !, and operators for updating one or more relations: ++ for insert, -- for delete, and ~~ for clear. The following example illustrates a _FORALL instruction.

```
_FORALL( _EDB(b(a,X,Z)), _EDB(c(Z,Y)),
         <(X,Y), ++_EDB(a(X,Y)))
```

The first two predicates access relations, the third predicate tests a condition, and the fourth predicate inserts tuples into a relation. This instruction is equivalent to the following SQL statement:

```
INSERT INTO A(X,Y)
SELECT DISTINCT B.X, C.Y
FROM  B,C
WHERE B.W = 'a' AND B.Z = C.Z AND B.X < C.Y;
```

The _EXISTS instruction implements special cases 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. The _MOVE instruction implements a special case update in which the 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.

The Glue language offers a uniform subgoal syntax for referring to relations, Nail rules, and procedures. For example, the second Glue assignment statement in Figure 1 refers to relation answer(X), procedure read(S) and Nail

rule `tc(S,X)`. IGlue handles relation references and procedure calls separately. Calls to IGlue procedures, which represent compiled Glue procedures or Nail queries, are explicit. The syntax of a procedure call is:

`_CALL(`*before*`, _PROC(`*call*`, `*possibles*`), `*after*`)`

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 middle argument gives the call with its arguments, and a list of all the possible procedures references. Each entry in the list includes the module name, the procedure name, and the pattern of input and output arguments. For example:

```
_CALL(r1(X), _PROC(f(X,Y), [m:f/1:1]), r2(X,Y))
```

There can be more than one possible procedure referent, because the procedure call might have a variable for its procedure name. 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 bindings immediately before and after the call. So procedure calls can be expensive to set up, execute, and recover from.

In the IGlue code in Figure 2 we see that relation and procedure predicates are annotated with predicate class descriptors: `_EDB`, `_LOCAL`, `_IN`, and `_OUT`. Additional descriptors are described in [4]. As will be described in Section 5, the Glue compiler analyzes the type of each predicate and determines the set of potential referents as early as possible.

4 The Nail Compiler

In order to optimize Nail query evaluation the Nail compiler, which is implemented in Prolog, applies one of two variants of the magic-sets transformation to the Nail program and query. The transformation strategy is selected as follows:

```
if Nail program is negation-free and right-linear then
    use context right-linear transformation [7];
else
    use supplementary magic-sets transformation [19];
```

The compiler then chooses one of three strategies to evaluate the transformed program, according to the following decision procedure:

```
if the transformed program is negation-free then
    use semi-naive bottom-up evaluation [2];
else
    if the transformed program is stratified then
        use Kerisit-Pugin's method [9];
    else
        use alternating fixpoint for magic programs [10];
```

The Nail compiler makes these selections at compiler time, because all conditions in the above algorithms can be tested by checking only the program syntax. Kerisit-Pugin's method 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 linker and static optimizer.

The previous Nail compiler applied Ross's method [17] 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 and bill-of-materials, the context right-linear transformation is much more efficient than the magic-sets.
- For stratified programs Kerisit-Pugin's method is better.
- The previous compiler handle 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 not possible to decide syntactically whether a set of rules with negation is modularly stratified or not. The programmer must guarantee that the given program is modularly stratified in order to make the system run correctly.

The last limitation motivated us to look for a robust algorithm that works for fully general Datalog with negation and that supports three-valued well-founded models. In this general setting, however, the well-founded model of the magic program may not agree with the well-founded model of the original program. To fix this disagreement Kemp et al. [8] developed a method, which, however, tends to make too many magic facts true, and therefore may not restrict the search space well. We created a novel method obtained by slightly tailoring Van Gelder's alternating fixpoint technique [21], a standard method to compute well-founded models, to magic programs so that it computes the correct answer to the query and always generates fewer (and in some cases significantly fewer) magic facts than Kemp et al.'s method. Its formal presentation, correctness and theoretical properties can be found in [10]. We implemented this method for fully general Datalog with negation.

Since our new method is intended for the general case, the reader might anticipate that it does not work well for the subclass of modularly stratified programs compared with the previous compiler. While one can write a modularly stratified program for which the new method is significantly inferior to the previous method, 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. See details in [10].

Since 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 solved by introducing a new Glue operator "?" for undefinedness.

5 The Glue Compiler

Compiling Glue into IGlue takes place in four phases: parsing, code generation, static code optimization and linking. Parsing and linking are straightforward and will not be discussed any further. However, we will describe

some interesting aspects of the code generator and the static optimizer. The Glue compiler is implemented using a combination of C, C++ , and Prolog.

Nailog semantics were particularly difficult to handle efficiently, because in general the predicate to which a Glue subgoal refers can only be determined at run time. In other words, Nailog is a late binding system. Another major technical problem that needed to be solved was the careful control of compiler-generated temporary relations. Temporary relations are frequently needed to ensure the correctness of emitted code, but they are expensive in time and space.

5.1 Code-Generator Optimizations

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. They are described below.

5.1.1 Predicate Class Analysis

As was mentioned earlier, the Nailog term system allows subgoals to have variables as their predicate names. Predicate selection for these subgoals can in general only be resolved at run time. The simplest solution would be to perform predicate selection at run time, when the functor will be fully bound. However, that would mean treating every subgoal as a potential IGlue 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.

5.1.2 Temporary Relation Compression

Temporary relation compression involves analysis of the values that are stored in compiler-generated temporary relations. The compiler uses these relations to store variable bindings between joins. 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. On average, this optimization reduced code size by 15% and decreased execution times by 15%.

5.2 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. The optimizer repeatedly performs peephole and data-flow directed optimizations until the program is stable. Peephole analysis uses only local information. Constant and copy propagation require data flow analysis. The aim is to reduce the number of operations on relations, or to remove a relation entirely.

A cardinality analysis algorithm has been developed. It identifies relations that contain at most one tuple. These relations could be replaced by simpler data structures. The required changes to the IGlue language and interpreter have not yet been made, so performance numbers are not yet available. However, the analysis algorithm is very effective at locating single-tuple relations. In one particular case, the analyzer algorithm was able to prove that 65 out of 66 local and temporary relations were single-tuple relations.

5.2.1 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 reduce the number of relations and variables within the join. The Glue optimizer analyzes the use of variables within the join. It removes existential 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 benchmark suite varied between 0% and 3%.

5.2.2 Data Flow Analysis

Data Flow Analysis (DFA) in IGlue is similar to DFA in traditional single-valued variable imperative languages, but there are some important differences. Firstly, DFA in IGlue is concerned with relations, not variables. Secondly, 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 practise 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).

For programs in the benchmark suite, the DFA optimizations reduced code size from between 20% and 25%, but only reduced run times from between 1% and 20%. 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 were not created equally. The static optimizer is very good at identifying redundant operations (such as unnecessary data copying and movement). Unfortunately, these operations are usually quick to execute precisely because they *are* redundant. Hence their removal does little to reduce program run times. Most of the IGlue interpreter's execution time is spent computing nonredundant joins. The Glue optimizer can do very little to improve the internals of these joins, the peephole optimizations are all that apply. The Glue optimizer can do more with small joins and copying operations.

The optimizer performed noticeably better on the THOR code (and especially the parser) than on the other programs. The THOR programs have more control flow operations and smaller relations than the other benchmarks. Hence the IGlue interpreter spends proportionally less of its time inside large joins. Glue control statements typically produce small joins and many temporary relations, which is the type of code that the Glue optimizer handles best. Hence the speed improvements are best for these programs.

The PATH and BOM programs have very few control branches, and so the IGlue interpreter spends most of its time executing large joins. Therefore the Glue optimizer can do little to improve these programs.

For the reasons given above, the IGlue optimizer has proven to be effective exactly when the Glue optimizer is ineffective, and vice versa. The two optimizers achieve a useful synergy.

6 The IGlue Interpreter

The IGlue interpreter is a single-user, memory-resident database system. 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 IGlue interpreter, implemented in C++, runs on Sun3 and DEC5000 workstations with 16 to 32 megabytes of main memory. 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.

The IGlue interpreter is divided into three functional components. The relation manager controls access to relations. The dynamic optimizer selects execution plans for IGlue queries. The abstract machine executes IGlue instructions. Here we focus on describing how the interpreter optimizes and executes query plans.

6.1 Query Processing in IGlue

IGlue's query expression, the `_FORALL` instruction, is implement using the nested-loop join algorithm with hash indexes on join and selection arguments. Variations of this join method have been used in other systems for processing joins in main memory (e.g., [24]). The query processor assumes that the run-time optimizer, which will be described below, has selected access paths 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, access tuples, and update relations. For example, the following query plan

```
_FORALL(_EDB(p(X,Y,W)), _EDB(q(Y,W,Z)),
        ++_LOCAL(r(X,Z)))
```

is computed by the following steps.

```
Open an access path to relation p(X,Y,W);
For each tuple obtained from access path :
    Record bindings for variables X, Y and, W;
    Open an access path to relation q(Y,W,Z);
    For each tuple obtained from the access path :
        Record the binding for variable Z;
        Insert tuple (X,Z) into relation r(X,Z);
```

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 predicates. Duplicate tuples are eliminated on insertion. The `_EXISTS` instruction is handled in a similar manner except that when the first result tuple is generated the process terminates.

6.2 The Dynamic Optimizer

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

6.2.1 Reoptimizing Queries

Query optimization is the problem of constructing an efficient execution plan to answer a declarative query. This consists of 1) choosing an order for multiple join operations, and 2) 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 the join process and statistical parameters that characterize the relations involved in the query. In relational database systems, query optimization is typically done at compile time. As described earlier, characteristics of IGlue programs generated from Glue-Nail code are problematic for conventional query optimization techniques. In particular, IGlue programs access 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. Thus it makes sense in IGlue to optimize these queries at run time when the statistical parameters of the temporary

relations become known, and to reoptimize queries when their statistical parameters change.

The IGlue run-time optimizer uses the dynamic programming approach of System R [6]. 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 or not to reoptimize it. An ideal criterion would trigger reoptimization exactly when the current query plan is no longer optimal. In [4] we investigated several different criteria for deciding when to reoptimize a query based on changes in relation cardinality. The criterion that worked best 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 $1/k$.

In the performance studies described later, we used the value $k = 2$.

6.2.2 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 which 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 design decision 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 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 would likely select a superset of 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.

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 via 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? In [4], we compared several approaches and found that the most effective strategy was to ignore the overhead of indexing.

The second subproblem deciding whether to 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 maintaining them. In [4], we compared several strategies and found two of them to be effective. The simplest strategy assumes that the cost of maintaining an index will be less then the benefit of using it index. This strategy never drops an index early. The other strategy uses data-flow analysis information derived by the static optimizer to determine if an index has any potential uses. If not, the index is dropped.

By combining of 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.

6.3 Performance Results

To see the advantage of adaptive optimization, let us look at some results. Consider the IGlue code in Figure 2, which is the context right linear transformation of the transitive closure Nail rules in Figure 1 against the query `tc(1,X)`. We prepared two different `arc(X,Z)` relations. The first represents a linear list with a loop using tuples of the form: $(1,2),\ldots,(N-1,N),(N,1)$. 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 above. The second version (nonadaptive) optimized each query only once. In both case the optimizer performed automatic index selection.

Tables 1 and 2 show the evaluation times (in seconds). These results clearly demonstrate the advantage of the adaptive query optimizer over the case when we do not use it. 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 to represent a complete binary tree with height H is $2^{H+1} - 2$.) 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.

Table 1: Adaptive vs. nonadaptive optimization: transitive closure on cyclic linear lists.

N	16	64	256	1024	4096
adaptive	.050	.191	.777	3.40	12.5
nonadaptive	.054	.270	2.180	29.77	475.1

Table 2: Adaptive vs. nonadaptive optimization: transitive closure on complete binary trees.

H	3	5	7	9	11
adaptive	0.035	0.129	0.508	2.04	8.3
nonadaptive	0.035	0.168	1.426	18.19	309.3

We also compared the performance of adaptive and nonadaptive optimization on programs from the application benchmark suite that will be described in Section 7.1. The executions times (in seconds) are presented in Table 3. Here

Table 3: Adaptive vs. nonadaptive optimization: programs from the Glue benchmark suite.

Program	BILL	CIFE	SG	SPAN
adaptive	14.81	32.65	16.95	22.73
nonadaptive	265.47	40.15	96.77	26.69

Program	CAR	OAG	$THOR_p$	$THOR_s$
adaptive	93.43	30.83	37.48	20.28
nonadaptive	216.80	31.58	36.90	21.01

we see that for five of the programs, execution time was significantly faster using adaptive optimization. Although the adaptive techniques did not improve performance for all programs, neither did it degrade performance in for any program.

7 Glue-Nail Applications

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

7.1 The Glue-Nail Benchmark Suite

The Glue-Nail benchmark contains ten applications. When compiled into IGlue, the applications range in size from 15 instructions (TC) to 951 instructions (THOR). The number of tuples contained in EDB relations varies from 15 (CAR) to 15,100 (BILL).

THOR simulates a logic circuit. The application has two parts: $THOR_p$ parses a circuit description and converts it into a set of relations; $THOR_s$ simulates the circuit. The application creates and manipulates ten EDB relations containing about 1,600 tuples.

BILL constructs a *bill of materials*: the quantity of each basic parts required to build a complex object. It accesses an EDB containing two relations that hold a total of 15,100 tuples.

SG solves the *same generation* problem: finding 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.

PATH finds paths in a graph using a transitive closure algorithm coded directly in Glue. This benchmark was written before the Nail compiler was operational.

TC shown in Figure 1, finds paths in a graph using recursive Nail rules for transitive closure. It was run on a variety of databases as described in Section 6.

CIFE schedules tasks and allocates resources required for constructing a building. The application ran on an EDB that describes an 8-floor, 16-room building. The EDB holds a total of 725 tuples.

SPAN finds the minimum spanning tree for a set of 50 points.

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

CAR simulates the movement of cars around a circular track. The simulation runs for 100 clock ticks. The database holds 14 tuples.

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

7.2 THOR Circuit Simulator

It is all very well to think up small example programs, and to code them up; but such programs are still just 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, and to code a section of it in Glue-Nail. The THOR circuit simulator [1] was chosen as an example application. 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 is a high level language system. Speed of coding is emphasized over speed of execution. There are situations when one doesn't have the resources to write a full C-SQL implementation. These are the situations that Glue-Nail is expected to be especially well suited.

Designing a proper experiment to test whether we have achieved this goal of increased programmer productivity is difficult. However, a preliminary comparison can be drawn from the THOR application. The Glue implementation of the THOR subset has 1,563 lines, and took approximately one month to write. The net-list code section of C-THOR code is 9,702 lines long. As a ratio of lines of code this is roughly 6:1. Given that author of the Glue version of THOR can write 1,000 lines of debugged C code per month, the ratio of coding time is roughly 10:1. Hence in this case Glue-Nail achieved its goal of improving programmer productivity, although at a significant cost in execution speed.

7.3 Glue Language Benefits

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 is 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 has only matching, not full unification, but this restriction was not found to be a problem. Nailog was especially useful in dealing with user defined complex objects.

8 Related Work

Before concluding we briefly mention several other experimental database languages and systems. These systems vary on several dimensions—language philosophy, multi- versus single-user, disk- versus memory-residency, and hardware platform.

LDL [13] is a main memory, single user system. LDL handles both declarative queries and procedural operations in the same language. Hence some rules must be read procedurally. Rules are compiled into an AND/OR graph representing joins and unions. Programmers can define indexes on base relations, or use the default of an an index on the first argument. The optimizer chooses join orders and annotates the graph with access method and execution strategy choices. The graph is translated into a C program which uses an underlying database. The decisions made by the optimizer are hard-coded into the target code.

Like Glue-Nail, CORAL [16] employs a two-language paradigm. The declarative language is based on Horn clauses with extensions for handling modularly-stratified negation, non-ground facts, set and multi-sets, aggregation, and I/O. The imperative language is C++ extended with a relation and tuple class library. Using annotations the user can control the evaluation in various ways. The user can specify join order information for each rule, or use the default left to right join order.

Aditi [20] is a multi-user, disk-based, deductive database system. Aditi programs are written in a variant of Prolog augmented with mode declarations, compiler directives for specifying evaluation strategies, and well-founded negation [8]. Aditi queries can be embedded in Nu-Prolog [18], which serves as the procedural support language. Aditi programs are compiled into a low level procedural relational language called RL, which is 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.

The EKS-V1 [23] system embeds a pure logic query language into a variant of Prolog called MEGALOG. MEGALOG is based on the BANG file system [5] for storing large amounts of facts and code. The execution strategy used is Query-Subquery (QSQ) [22]. The system also has a static optimizer, which chooses the join order, and identifies common subexpressions and tail recursion. Query evaluation is performed by BANG, and uses relation deltas.

9 Conclusion

We have presented an overview of the design and implementation of the Glue-Nail Database system. In particular, we described the optimization techniques that are used in the system. 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 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 serve 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. 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

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; Ashish Gupta and Sanjai Tiwari wrote the CIFE application. We are grateful to Jeff Ullman for his suggestions during the preparation of this paper.

References

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

[2] F. Bancilhon. Naive evaluation of recursively defined relations. In M. L. Brodie and J. Mylopoulos, editors, *On Knowledge Base Management Systems*, pages 165–178. Springer-Verlag, New York, New York, 1986.

[3] W. Chen, M. Kifer, and D. S. Warren. HiLog: A first-order semantics of higher-order logic programming constructs. In *Logic Programming: Proceedings of North American Conference*, pages 1090–1114, 1989.

[4] M. A. Derr. *Adaptive Optimization in a Database Programming Language.* PhD thesis, Stanford University, Stanford, California, December 1992. Department of Computer Science Report No. STAN-CS-92-1460.

[5] M. Freeston. The BANG file: a new kind of grid file. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, San Francisco, USA, 1987.

[6] P. Griffiths Selinger, M. M. Atrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 23–34. ACM, 1979.

[7] D. B. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, left- and multi-linear rule transformations that maintain context information. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 13–16, Brisbane, Australia, 1990.

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

[9] J.-M. Kerisit and J.-M. Pugin. Efficient query answering on stratified databases. In *Proceedings of International Conference on Fifth Generation Computer Systems*, pages 719–726, 1988.

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

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

[12] K. Morris, J. D. Ullman, and A. Van Gelder. Design overview of the NAIL! system. In *Proceedings 3rd Int Conference on Logic Programming*, pages 554–568, New York, 1986. Springer-Verlag.

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

[14] G. Phipps. *Glue: A Deductive Database Programming Language*. PhD thesis, Stanford University, Stanford, California, July 1992. Department of Computer Science Report No. STAN-CS-92-1437.

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

[16] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL—Control relations and logic. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 238–250, 1992.

[17] K. Ross. Modularly stratification and magic sets for datalog programs with negation. In *Proceedings of the 1990 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 161–171, 1990.

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

[19] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, Rockville, Maryland, 1989.

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

[21] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proceedings of the 1989 ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–10, 1989.

[22] L. Vielle. Recursive axioms in deductive databases: The query/sub-query approach. In L. Kerschberg, editor, *Expert Database Systems*. The Benjamin/Cummings Publishing Company, 1987.

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

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