Adaptive Query Optimization in a Deductive Database System*

Marcia A. Derr
AT&T Bell Laboratories
600 Mountain Avenue, 2B-430
Murray Hill, NJ 07974-0636 USA
mad@research.att.com

Abstract

Current database research is concerned with the design of new programming languages and systems to support advanced database applications such as software engineering information systems and computer-aided design. Glue-Nail is a deductive database system that offers two complementary languages, one declarative and the other procedural, for writing such applications. This paper is concerned with the problem of query optimization in the Glue-Nail system.

Glue-Nail programs exhibit characteristics, such as a predominance of temporary and dynamic relations, that present problems for conventional query optimizers. This work explores new approaches to optimization that accommodate these characteristics. In particular, it explores the combined use of reoptimization and automatic indexing to adapt query plans to run-time changes in the database. A performance study was conducted to compare and evaluate alternative strategies. The empirical results of this study support the use of adaptive techniques over static approaches.

Keywords: query optimization, deductive database systems, adaptive techniques, performance evaluation, memory-resident databases.

1 Introduction

A current focus of database research is the design of better programming languages and systems to support advanced applications such as computer-aided design and manufacturing, office systems, and software engineering information systems. Glue-Nail [8, 16] is a deductive database system that offers two complementary languages, one declarative, the other procedural, for writing database applications. Nail [12, 13] is a logic-based query language that uses function symbols to represent complex objects and is powerful enough to express recursive queries. Glue [15] is a procedural language that augments query capability with update operations, aggregation, input-output libraries, and procedural control.

Glue-Nail programs are compiled into a lower level language called IGlue. The resulting IGlue program is executed by the back end of the system, the IGlue interpreter. This paper is concerned with how to optimize the evaluation of IGlue code. In particular, we are concerned with query optimization—converting a declarative specification of a query into an efficient plan for executing the query. IGlue programs exhibit characteristics, such as a predominance of temporary and dynamic relations, that present problems for conventional query optimizers and query processors. The goal of this research was to explore adaptive optimization techniques that are able accommodate these characteristics. We have designed and implemented a run-time query optimizer which improves system performance by automatically reoptimizing query evaluation plans. The optimizer also creates and drops indexes automatically. This paper describes the IGlue adaptive optimizer and presents the results of performance experiments that evaluate the effectiveness of alternative reoptimization strategies.

The remainder of the paper is organized as follows. Section 2 presents an overview of the Glue-Nail system architecture and describes the IGlue target language and its interpreter. Section 3 reviews query optimization and argues for the use of adaptive techniques in the Glue-Nail system. Section 4 presents the optimization framework used in the IGlue interpreter. Section 5 describes alternative reoptimization strategies and presents results that compare the performance of each strategy. Section 6 compares the optimization techniques found in IGlue to other approaches. Section 7 presents a summary and conclusions.

*This work was done while the author was as Stanford University, Stanford, CA.

CIKM '93 - 11/93/D.C., USA

© 1993 ACM 0-89791-626-3/93/0011 ....$1.50

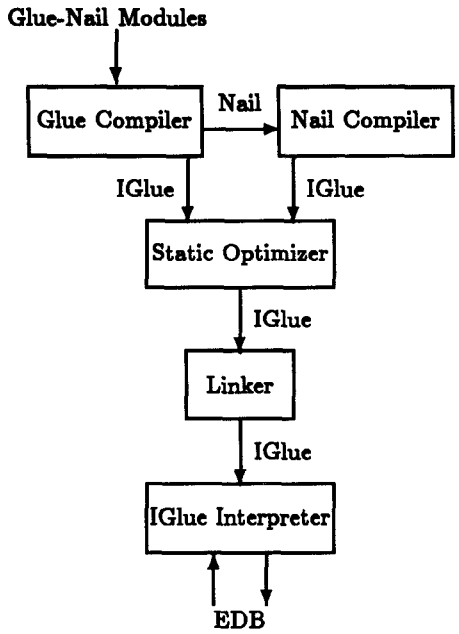

Figure 1: The Glue-Nail architecture.

2 System Overview

Figure 1 illustrates the architecture of the Glue-Nail system. The front end of a system is a language compiler which compiles both Glue code and Nail rules into the target language IGlue. The Glue compiler maps each Glue statement into a sequence of one or more IGlue instructions. The Nail compiler transforms each Nail query and an associated set of Nail rules into an IGlue procedure, using variants of the magic set transformation and evaluation strategies that support well-founded models. The front end also includes an IGlue-to-IGlue static optimizer that performs various peep-hole optimizations and data-flow analyses. The linker combines separately compiled Glue-Nail sources into a single IGlue program. The back end of the system is the IGlue interpreter which is responsible for optimizing and executing IGlue code and for managing access to temporary and EDB[1] relations. In the remainder of this section we will focus on the IGlue language and interpreter.

2.1 IGlue: the Language

IGlue, a relational language with control statements, was designed as the target language for both Glue procedures and Nail rule. Below is a brief overview of the language, with a focus on IGlue's relational instructions. A complete description is found in [7].

IGlue provides the following categories of statements:

Declarations. IGlue declarations tell the interpreter about which EDB relations an IGlue program will access and which temporary relations each IGlue procedure will create.

Branches. IGlue branches are used to implement the loop and if-then-else constructs found in Glue.

Procedure call and return. IGlue provides explicit procedure call and return instructions to implement the implicit procedure calls of Glue.

Aggregation. Aggregation operators are implemented by an explicit "call" instruction similar to procedure calls.

Relational instructions. Relational instructions are used to to express relational queries (e.g., select, project, and join) as well as to update relations.

Relation instructions are the "query expressions" of IGlue. They access and update both persistent and temporary relations. Relational instructions come in four varieties:

```
FORALL(p1, p2, ..., pn)
EXISTS(p1, p2, ..., pn)
MOVE(p1, p2)
TRANSFER(p1, p2)
```

where each p_i refers to a relation, a Glue built-in predicate, or a condition predicate. Predicates in the FORALL and EXISTS forms may be negated.

The FORALL form expresses relational select-project-join queries. It computes the set of tuples that satisfy a query, and, in addition, performs any indicated updates. Relation predicates may be annotated with one of three update operators: ++ for insert, -- for delete, and ~~ for clear. More than one relation can be updated in a single expression. The following example illustrates two FORALL instructions.

```
FORALL(~~EDB(a(_,_))
FORALL(
   EDB(b(g,X,Z)), LOCAL(c(Z,Y)),
   <(X,Y), ++EDB(a(X,Y))
)
```

The first instruction clears the EDB relation a. The second instruction "joins" four predicates. The first predicate accesses an EDB relation, the second accesses a local (temporary) relation, the third is a condition test, and the fourth is an update that inserts tuples into relation a. This instruction is equivalent to the following SQL statement:

[1]EDB stands for *Extensional Database*, a set of persistent relations.

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

The EXISTS instruction implements a special case of the FORALL instruction by computing at most one solution to its arguments. It is is used as a condition in IGlue's branching instructions. The MOVE instruction implements a special case in which the tuples of a source relation, p_1, are moved to a target relation, p_2, first clearing any previous contents of the target. After the move, the source relation is empty. The TRANSFER instruction is similar to the MOVE instruction except that the target relation is not cleared before adding new tuples.

In the example IGlue code above, we see that relation and procedure predicates are annotated with predicate class descriptors, EDB and LOCAL. There are additional descriptors, not shown, for other types of predicates. The Glue compiler analyzes the type of each predicate and determines the set of potential referents for each predicate as early as possible. This avoids the expense of resolving predicate references at run-time. However, for readability, the predicate class descriptors will not be shown in the examples found later in the paper.

2.2 IGlue: The Interpreter

The IGlue interpreter is the back end of the Glue-Nail system database system. It is responsible for optimizing and executing IGlue programs, and for managing access to relations. The interpreter is designed to support a memory-resident database. That is, it preloads all EDB relations required by a program into main memory. All temporary relations created by the interpreter reside only in main memory. The IGlue interpreter is designed for single-user applications and does not provide locking, transactions, or other mechanisms for concurrency control.

A diagram of the interpreter is shown in Figure 2 The IGlue interpreter reads in an IGlue program and converts it into an internal representation. It also reads from disk all the EDB relations that the IGlue program might access. The interpreter's abstract machine then executes the stored IGlue program, invoking the run-time optimizer and the relation manager as needed. When the IGlue program terminates, EDB relations that have changed are written back to disk.

At the heart of the abstract machine is the query processor, which evaluates IGlue FORALL and EXISTS expressions. The query processor uses an index nested-loop join strategy. Whang and Krishnamurthy [21] argue that with the appropriate indexes, the nested-loop join method alone is adequate for memory-resident databases. The query processor depends on a run-time query optimizer to choose a join order and to select appropriate hash indexes on join and selection attributes, creating them if necessary.

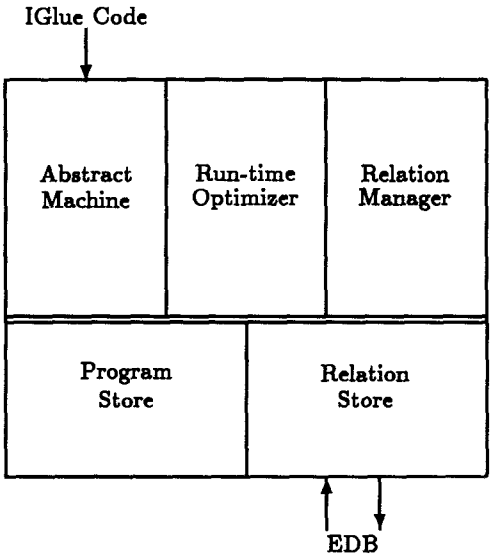

Figure 2: The IGlue interpreter

3 An Argument for Adaptive Optimization

This section discusses the problem of optimizing IGlue programs. We begin by reviewing query optimization for relational database management systems. We then discuss why conventional techniques are inadequate for IGlue. Finally we argue for the use of adaptive techniques.

3.1 Relational Query Optimization

In relational database management systems, queries are expressed declaratively. That is, each query specifies what is to be computed, but does not specify how to compute it. It is the job of the query optimizer to formulate a plan for executing the query. Query optimization can be thought of as a search problem in which the search space is the set of alternative query execution plans. The query optimizer searches for the plan with the lowest cost.

Because exact execution costs usually cannot be determined without actually executing a query, a query optimizer must estimate the cost of each alternative plan. The optimizer calculates cost estimates using parameters that include statistical profiles of the relations involved in the query. Typically, the profiles includes the number of tuples in each relation (cardinality) and the number of distinct values for each attribute (do-

main size). Additional information, such as minimum and maximum values, and histograms of values, may also be included in the profile.

For efficiency, relational queries are usually optimized once to produce compiled query plans that are stored and executed multiple times. In particular, this approach is applied to queries that are embedded in an application program written in a host programming language. The compiled approach assumes that the parameters for which the plan is optimal do not change at run time. Many commercial systems provide mechanism to invalidate a query plan that has become infeasible, e.g., when an index on which the query plan depends is dropped [3]. Commercial systems may also provide utilities for manually updating statistical parameters and manually reoptimizing queries [6]. However, no mechanisms exist to reoptimize a query automatically when changes in statistical parameters indicate that a valid query plan may no longer be optimal.

3.2 Optimizing IGlue Queries

Relational queries primarily access only persistent relations. In many relational database applications, persistent relations are large, relatively static, and shared by multiple users. IGlue programs access persistent relations and temporary relations. Temporary relations include relations defined by a Glue programmer that are local to a procedure, as well as relations introduced by the Glue compiler to hold intermediate results. Temporary relations differ from persistent relations in several ways. Temporary are not shared by other programs or users. Temporary relations range in size from very small (one or two tuples) to very large (e.g., a cartesian product.) In IGlue programs, temporary relation may be frequently updated. In particular, temporary relations that are used in the semi-naive bottom-up evaluation [2] of a recursive query are updated on each loop iteration. In IGlue programs, temporary relations are referenced much more frequently than persistent relations [7].

Conventional compile-time query optimization is not appropriate for IGlue because of the presence of temporary relations. While statistical profiles may be available ahead of time for EDB relations, at best, they can only be estimated for temporaries. Furthermore, queries that access temporary relations may be embedded in loops or procedures that are called repeatedly. Over time, the statistical profiles of temporary relations (and persistent ones, for that matter), may vary, invalidating any optimization decisions made at an earlier stage.

To illustrate this problem, consider the fragment of IGlue code shown in in Figure 3. Each time the loop is executed, tuples are inserted into relation p and deleted from relation r. The size of relation q remains constant. Table 1 shows how the cardinalities of p, q, and r might change over time. It also shows how the optimal join order for the expression in line 4 changes as the relation

```
1)  lbeg1:
      IFNOT EXISTS(cnt(I), I < 15)
        GOTO lend1
2)    FORALL(~~tmp0(_,_))
3)    FORALL(
        p(X,Y,Z), r(Y,V,T),
        q(Z,W,V), ++tmp0(Y,T)
      )
4)    FORALL(
        tmp(W,T), cnt(I),
        ++p(W,I,T), --r(_,_,W)
      )
5)    FORALL(cnt(I), II = I+1, ++tmp1(II))
6)    MOVE(tmp1(I), cnt(I))
7)    GOTO lbeg1
8)  lend1:
```

Figure 3: Example: IGlue code loop with changing relations.

Table 1: Cardinality changes and optimal join order

Iteration	*Cardinalities*			*Optimal Order*
	p	r	q	
1	20	3562	200	p q r
4	256	2473	200	q p r
13	2074	2051	200	q r p

cardinalities change (assuming that domain sizes are approximately equal and constant).

The above example was constructed to demonstrate the effect of changing cardinalities on join order. It is not difficult to find this same phenomenon in "real" Glue programs. The following query was taken from a program that computes the *bill of materials* for a complex object.

```
FORALL(
   unknown(P), !notyet(P),
   assembly(P,S,M), p(S,R,N),
   ++tmp(P,M,S,R,N)
)
```

This expression appears in a loop that is executed until relation unknown(P) becomes empty. During the execution of the loop, relations unknown(P) and notyet(P) shrink, relation p(S,R,N) grows, and relation assembly(P,S,M) remains constant in size. This pattern is also found in IGlue code that is generated by the Nail compiler.

3.3 Alternative Approaches to Optimization

Given that conventional optimization is problematic for Glue queries, how and when should a Glue query be

optimized? One approach is to optimize queries at compile-time using statistical profiles for persistent relations and deriving or guessing parameters for temporary relations. This approach has the advantage that all decisions are made at compile-time, making it possible to compile query plans instead of interpreting them. However, by using poor estimates, this solution could produce bad query plans. Furthermore, even if a plan performs well the first time a query is executed, it could perform badly on subsequent executions if the relation parameters change.

Another approach is to delay optimizing queries that reference temporary relations until run time [3]. In IGlue, since most queries involve temporary relations, this approach defaults to postponing optimization until the first time a query is evaluated. However, this approach still has the problem that query plans have no way to adapt to changing parameters. One solution is to reoptimize a query *each time* it is to be executed to ensure that the query plan would always be optimal for the current parameters. But this approach would be expensive in circumstances where parameters don't change significantly from one evaluation of a query to the next.

A third approach is to adapt query plans to *significant* parameter changes. When a query is first optimized, its parameters are stored with the query plan. When the query is subsequently executed, the current parameters are compared to the stored parameters. If there is significant change, such as a relation cardinality increasing beyond a threshold, the query is reoptimized and the new plan and parameters are saved. Section 5 proposes and compares several alternative criteria for deciding when to reoptimize a query.

4 The Adaptive Query Optimizer

The IGlue interpreter features a query optimizer that constructs query plans at run time. The first time the interpreter encounters a query, it invokes the optimizer on that query. Recall from Section 2 that the IGlue interpreter evaluates queries using an index nested-loop join algorithm. The interpreter also supports two types of access methods: scanning all tuples in a relation and using an index to access selected tuples. To construct a plan to evaluate a query within this framework, the optimizer makes two types of decisions: 1) choose the join order, and 2) determine how to access each relation. Using a cost model [7] that characterizes the cost of each query processing method, the optimizer estimates the cost of alternative join orders and access paths, and chooses the plan with the lowest cost. The optimizer then annotates the query with its choices for join order and access path. To generate alternative query plans, the IGlue optimizer employs a dynamic programming algorithm based on the System R approach [18]. As implemented in IGlue, this algorithm handles queries of up to fifteen predicates.

The run-time optimizer also makes all indexing decisions—when to create, use, and drop each index—automatically. This relieves the programmer or the compiler from having to select indexes without knowing the order in which relations will be joined. For each relation in a particular ordered join, the optimizer decides which access method, scan or index, has the lowest cost. In making this decision, it ignores whether or not an index exists. If the optimizer determines that indexed access is cheaper than scanning, and the index already exists, then the optimizer chooses the index method. However, if the index does not exist, how should the optimizer count the overhead of building it? In [7] we compared several approaches on a benchmark suite of Glue-Nail programs and determined that the most effective strategy is to ignore the overhead. This strategy is based on the heuristic that the cost of creating an index that might be used only once is less than than the penalty of failing to create an index that may be used multiple times in the future.

The optimizer also determines whether to maintain or drop indexes when relations are updated. Indexes on temporary relations are automatically dropped when the temporary relation is deleted. However, the system may want to drop an index earlier to avoid excessive costs of maintaining it. In [7] several strategies were compared experimentally and two of them were found to be effective. The first strategy always maintains indexes. It implements the heuristic that the benefit of using an index is greater than the overhead of maintaining it. The second strategy uses data-flow analysis information, derived at compile-time by a static code analyzer, to determine if an index has any potential uses. The system will drop an index only if it has no potential uses. This second strategy for dropping indexes and the *ignore overhead* strategy for creating indexes were employed by the optimizer in the performance comparison described in the following section.

5 Reoptimizing Queries

As argued in Section 3, one way to achieve good performance in Glue-Nail is to reoptimize queries when relation parameters change. However, because the optimization algorithm itself is expensive, it should avoid reoptimizing queries when unnecessary. An ideal algorithm should reoptimize a query if and only if it will result in a better query plan than the current plan.

5.1 Alternative Strategies

Let us introduce four strategies concerning reoptimization. One strategy is never to reoptimize. The other

Table 2: Changing cardinalities.

Iteration	*Cardinalities*		
	p	r	q
1	20	3562	200
2	38	3023	200
3	178	2734	200
4	256	2473	200

three strategies monitor changes in relation cardinalities to trigger reoptimization.

The Never Strategy. The *Never* strategy never reoptimizes a query. The IGlue optimizer, using the *Never* strategy, chooses a query plan at run time, when the optimizer has access to parameters that describe temporary relations. Once a query plan is selected, it does not change.

The Change Strategy. The alternative to optimizing each query just once is to reoptimize when parameters change. The *Change* strategy implements an approach that monitors the relations involved in a query for changes. Whenever the optimizer chooses a query plan, it remembers the cardinalities of the relations that are involved in the query. Then, each time the query is to be executed, the optimizer compares the current relation cardinalities with the recorded cardinalities. If there are *any* changes the query is reoptimized. Of course, the fact that relation cardinalities have changed does not mean that the optimizer should choose a new query plan.

Again, consider the IGlue code in Figure 3. Table 2 lists some hypothetical cardinalities for relations p, q, and r for the first four iterations of the loop. Because the cardinalities of relations p and r change on each iteration, the *Change* strategy would reoptimize the query on line 4 of Figure 3 on each iteration. However, the only time the optimizer would produce a query plan that is different from the previous one is on the fourth iteration.

The Percent Strategy. The *Percent* strategy tries to detect significant changes in relation cardinalities. Like the *Change* strategy, the *Percent* strategy remembers relation cardinalities when it chooses an initial or a new query plan. A query is reoptimized only if the optimizer detects that a relation cardinality has increased by a factor of k or decreased by a factor of $1/k$, for some parameter k. The current implementation of this strategy uses a value of $k = 2.0$, which was selected earlier based on some preliminary tests.

Consider the cardinality changes in Table 2. The *Percent* strategy would ignore the changes between the first and second iteration because $38 < 20k$ and $3562 < 3023k$. However, the *Percent* strategy would reoptimize the query on the third iteration because $178 > 20k$. This strategy would also ignore the cardinality changes between the third and fourth iterations.

The Rank Strategy. The *Rank* strategy considers how relation cardinalities change relative to one another when deciding to reoptimize a query. The first time a query is optimized, the relations involved in the query are rank ordered by cardinality and the ranks are recorded. Each time the query is to be executed, new ranks are computed and compared to the recorded ranks. If the rank of any relation changes, the query is reoptimized.

Again, consider the cardinalities shown in Table 2. On the first iteration, the rank order of the relations, from largest to smallest, is r, q, p. The rank order remains the same until the fourth iteration when it changes to r, p, q. Thus, on the fourth iteration, the *Rank* strategy would reoptimize the query.

5.2 Comparing Strategies

How well do each of the above four strategies perform on typical Glue programs? For programs in which queries are executed multiple times and in which relation parameters vary at run time, we would expect the three strategies that monitor cardinality changes to give better performance than the *Never* strategy. Of these three strategies, which is best? For programs with queries that are executed only once or whose relation parameters do not significantly change, we would expect the *Never* strategy to perform better than the *Change* strategy, which reoptimizes a query if any cardinalities change. But we would hope that the *Percent* and *Rank* strategies, would avoid reoptimizing queries unnecessarily, and incur only slight penalties from the overhead of monitoring relation cardinalities.

To answer these performance questions, we tested each reoptimization strategy on ten Glue applications compiled into IGlue. These ten applications are summarized in Figure 4.

The first program, *adapt*, which contains code similar to the example shown in Figure 3, was designed to favor the three strategies that reoptimize queries. The other nine programs are Glue-Nail applications written for a variety of purposes by several different programmers. Each Glue-Nail program was compiled into IGlue and executed on a DEC 5000/200 workstation with 32 megabytes of memory. The programs were executed on a lightly loaded system. For each program, all relations were able to fit entirely in physical main memory. The

adapt Contains queries with changing relations cardinalities. EDB: three relations totaling 3,782 tuples.

bill Constructs a *bill-of-materials* for a part-subpart hierarchy. EDB: two relations totaling 15,100 tuples.

cife Schedules tasks and allocates resources for constructing an 8-floor, 16-room building. EDB: twelve relations totaling 725 tuples.

sg Solves the *same generation* problem on a family tree representing eight generations. EDB: two relations totaling 9,595 tuples.

load Converts a circuit description into a set of relations. EDB: eight relations totaling 1,472 tuples.

run Simulates a logic circuit. EDB: ten relations totaling 1,686 tuples.

prim Prim's minimum spanning tree algorithm. EDB: one relation of 100 tuples.

span Minimum spanning tree algorithm, adapted from an LDL example [4]. EDB: one relation of 50 tuples.

oag Searches for flights in an airline database. EDB: seven relations totaling 764 tuples.

car Simulates traffic movement around a circular track for 100 clock ticks. EDB: one relation of 14 tuples.

Figure 4: Glue-Nail application programs

execution time for each program/strategy combination was measured and recorded.

In [7] we combined the four alternative reoptimization strategies with three alternative strategies for creating indexes and five alternative strategies for dropping indexes. In all, 44 different different combinations were compared (not all 60 combinations were plausible.) Here, we report the results for four of those combinations: each of the four reoptimization strategies combined with the *ignore overhead* index creation strategy and the *potential use* index dropping strategy, described in Section 4. Figure 5 compares the the execution times for all four strategies on five of the programs. To compare performance across programs, the execution times are normalized with respect to the *Never* strategy. For the *adapt* program, the *Change*, *Percent*, and *Rank* strategies are 3.9 to 4.2 times faster than the

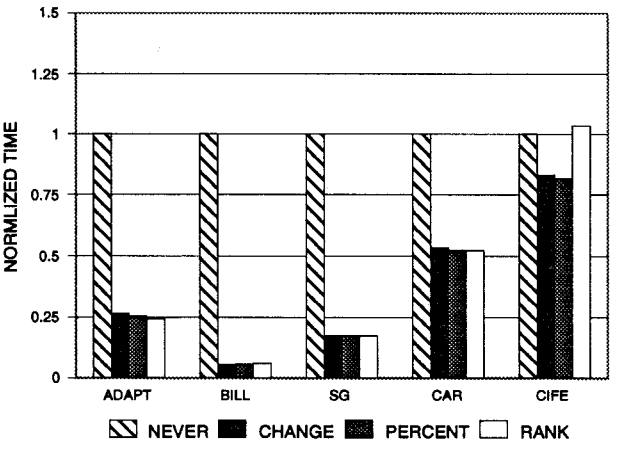

Figure 5: Comparing reoptimization strategies I. The horizontal axis is labeled with the names of the programs. The vertical axis is labeled with execution time normalized with respect to the *Never* strategy.

Never strategy. A trace of the optimizer's actions for the query involving relations p, q, and r shows that the *Change* strategy reoptimized the query 59 times, during which it changed query plans (join order and access method) nine times. The *Percent* strategy reoptimized the same query nine times, and changed query plans five times. The *Rank* strategy reoptimized the query three times, and changed query plans each time. This example demonstrates the importance of being able to reoptimize queries. The cost to reoptimize a query, even when the query plan does not change, is small compared to the penalty of using a single query plan throughout the execution.

For the other four programs, the reoptimizing strategies provide varying degrees of improvement over no reoptimization. The only exception is for the *cife* program where the *Rank* strategy degrades performance by about 4 percent. This strategy failed to to detect several reoptimization opportunities found by the *Change* and *Percent* strategies. In particular, both *Change* and *Percent* are able to detect absolute changes in a single relation that result in the use of an index, but not necessarily a change in join order. The *Rank* strategy detects relative changes among a set of relations, which signal a change in join order. Furthermore, the overhead of the *Rank* strategy resulted in slightly slower performance than the the *Never* reoptimize strategy.

The most dramatic improvement occurs in the *bill* program. An optimization trace for the *Percent* strategy shows that this improvement resulted from reoptimizing two different queries that appear in loops. The first query, which involves three predicates, was reoptimized eleven times and changed query plans three times. The second query, which involves six predicates, was

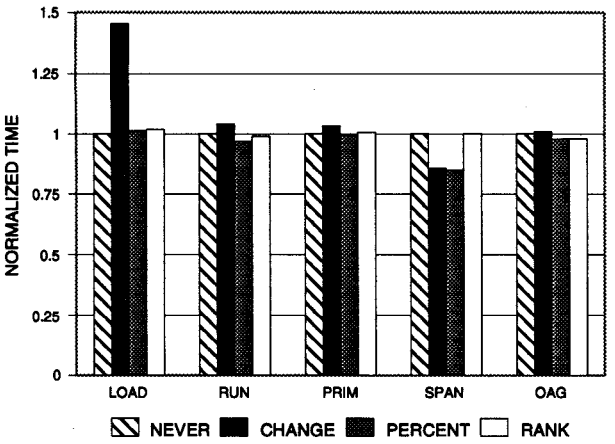

Figure 6: Comparing reoptimization strategies II. The horizontal axis is labeled with the names of the programs. The vertical axis is labeled with execution time normalized with respect to the *Never* strategy.

reoptimized four times and changed query plans three times.

The results in Figure 5 highlight the performance advantage of being able to reoptimize queries. All three alternatives are feasible, with the *Change*, and *Percent* strategies outperforming the *Rank* strategy on one program.

Figure 6 compares the the execution times for all four strategies on the five remaining programs. Again, execution time is normalized with respect to the *Never* strategy. Here, we see that performance improvements are less dramatic for the reoptimizing strategies. The biggest improvement is seen in the *span* program, using the *Change* and *Percent* strategies. An optimization trace for the *Percent* strategy shows that the performance gain resulted from one query of four predicates which was reoptimized six times and changed query plans twice.

While we do not see huge performance gains resulting from reoptimization in Figure 6, neither do we see significant degradation in performance, with one exception. In the *load* program, the *Change* strategy degrades performance by 46 percent. This slowdown results from the overhead of repeatedly reoptimizing queries that do not actually need new query plans.

The combined results in Figure 5 and Figure 6 suggest that the *Percent* strategy is the best alternative. Of the three reoptimizing strategies it best meets the requirements of an ideal algorithm. It improves performance by generating new query plans when it detects significant parameter changes. It does not degrade performance by unnecessarily reoptimizing queries.

6 Related Work

This section reviews related research in two areas: optimization for deductive database systems and alternatives to reoptimization.

6.1 Query Optimization in Deductive Database Systems

Optimization research in deductive database systems has focused more on how to compile recursive queries into database operations than on how to execute those operations efficiently. In the NAIL! system, the subgoal ordering algorithm considered only binding patterns, in choosing the order in which to evaluate subgoals. Because Nail rules were ultimately translated into SQL, the task of ordering joins on the basis of relation cardinality and domain size was left to the SQL optimizer.

CORAL [17], like Glue-Nail, employs a two-language paradigm. The declarative language is based on Horn clauses. The imperative language is C++, extended with a relation and tuple class library. CORAL allows the programmer declare indexes on base or derived relations. The Magic Template transformation, one of the rule rewrite strategies, also declares indexes. CORAL lets the user specify join order information for each rule. Otherwise, when relation cardinalities are unknown, CORAL selects a left to right join order. For semi-naive evaluation, CORAL uses a heuristic that moves any "delta" predicates to the left.

Aditi [20] is a multi-user, disk-based, deductive database system. Aditi programs are written in a variant of Prolog and are compiled, via two intermediate languages, into a low level procedural relational language (RL) that is similar to IGlue. 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. The RL join operator also distinguishes between two types of join conditions: those that are useful for indexing and those that are not. Information about available indexes for each relation is contained in a data dictionary.

LDL (a Logical Data Language) [5, 14] is a main memory, single-user deductive database system. 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 [11] chooses join orders and annotates the graph with access method and execution strategy choices. 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.

6.2 Alternatives to Reoptimization

Instead of reoptimizing queries when parameters change, several researchers have proposed generating multiple query plans to accommodate different parameter values. One example, proposed for the XPRS project [19], is an optimization strategy that generates multiple query plans, each of which is (nearly) optimal over a range of buffer sizes. At query execution time, the actual buffer size determines which of these query plans to evaluate.

Graefe and Ward [9] propose a technique that uses *dynamic query evaluation plans*. In their paradigm, a dynamic query plan consists of a set of alternatives and a decision procedure that is evaluated at run-time to choose the optimal plan. They acknowledge that the number of alternative plans could be prohibitively large but suggest that the number could be kept small by considering only plans that are close to optimal over a wide range of parameter values.

Ioannidis et al. propose a related approach, called *parametric query optimization* [10], that provides a strategy for identifying alternative query plans. In principle, this approach optimizes a query by identifying optimal execution plans for all possible value combinations of run-time parameters. In practice, however, the technique is applied to selected subset of run-time parameters. While experiments have shown this approach to be effective for a single run-time parameter (either buffer size or index type), it is not known how well it would scale to a a large number of run-time parameters.

Antoshenkov [1] describes a dynamic method, called competition, for optimizing single table access in the Rdb/VMS[2] system. This method, which assumes an L-shaped selectivity distribution, executes one or more access strategies simultaneously, may switch to a better strategy at some optimal point during query evaluation.

7 Conclusion

Glue-Nail programs exhibit characteristics that present problems for conventional query optimizers, such as a predominance of temporary and dynamic relations. This work proposes new approaches to optimization that accommodate these characteristics. In particular, it explores the combined use of reoptimization and automatic index selection to adapt query plans to run-time changes in the database. One advantage of this approach is that size parameters of both temporary relations and persistent relations are known at run time. A second advantage is the opportunity to reoptimize a query when these relation parameters change.

Four alternative strategies for deciding if and when to reoptimize a query were implemented in the IGlue run-time optimizer. These strategies were combined with techniques for automatically deciding when to create and drop indexes. All four strategies were compared on a benchmark of Glue-Nail programs. The results clearly demonstrate that reoptimizing queries can result in large performance gains. Of the three reoptimizing strategies considered, the *Percent* strategy, which detects when cardinality changes by a factor of k, provided the best overall performance. This strategy improves performance by generating new query plans when it detects significant parameter changes. It does not degrade performance by unnecessarily reoptimizing queries.

Although the adaptive optimization techniques described here were developed in the context of Glue-Nail, a deductive database system, they generalize to other programming languages and systems that manage and manipulate collections of temporary and persistent objects with dynamic properties. Further research is needed to extend the techniques to other environments, such as concurrent, disk-based systems. Another interesting direction would be to combine adaptive optimization with a framework such as dynamic query evaluation plans [9]. Once selected, a query plan is cached so that it may later be reselected by a choice operator.

[2] Rdb/VMS is a trademark of Digital Equipment Corporation.

Acknowledgements

I would like to extend my thanks to those who contributed Glue-Nail programs: Kathleen Fisher wrote the *car* program; Ashish Gupta and Sanjai Tiwari wrote the *cife* application; and Geoff Phipps wrote the *bill*, *load*, and *run* applications. I am grateful to Arthur Keller, Jeff Ullman, and Gio Wiederhold for their comments and suggestions regarding this work.

References

[1] G. Antoshenkov. Dynamic query optimization in Rdb/VMS. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 538–547, Vienna, Austria, Apr 1993. IEEE.

[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] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. G. Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost. Support for repetitive transactions and ad hoc queries in System R. *ACM Transactions on Database Systems*, 6(1):70–94, 1981.

[4] D. Chimenti and R. Gamboa. *The Salad Cookbook: A User/Programmer's Guide*. Microelectronics and Computer Technology Corporation, Austin, Texas, 1989.

[5] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Abstract machine for LDL. In *International Conference on Extending Database Technology*, pages 153–168, 1990.

[6] C. J. Date and C. J. White. *A Guide to DB2, Third Edition.* Addison-Wesley, Reading, Massachusetts, 1989.

[7] 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.

[8] M. A. Derr, S. Morishita, and G. Phipps. Design and implementation of the Glue-Nail database system. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 147–156, Washington, DC, May 1993.

[9] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 358–366, 1989.

[10] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 103–114, 1992.

[11] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 128–137, 1986.

[12] K. Morris, J. F. Naughton, Y. Saraiya, J. D. Ullman, and A. V. Gelder. YAWN! (yet another window on NAIL!). *Data Engineering*, 10(4):28–43, 1987.

[13] K. Morris, J. Ullman, and A. van Gelder. Design overview of the NAIL! system. In *Proceedings Third International Conference on Logic Programming*, pages 554–568, 1986.

[14] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, Rockville, Maryland, 1989.

[15] 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.

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

[17] 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.

[18] P. G. 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, 1979.

[19] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The design of XPRS. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases*, pages 318–330, 1988.

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

[21] 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.