Implementing an Interpreter for Functional Rules in a Query Optimizer

Mavis K. Lee[*], Johann Christoph Freytag[†], Guy M. Lohman
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

Abstract

Query optimizers translate a high-level, user-submitted query into an efficient plan for executing that query, usually by estimating the execution cost of many different alternative plans. Existing implementations of these sophisticated but complex components of relational database management systems (DBMSs) typically embed the available strategies in the optimizer code, making them difficult to modify or enhance as improved strategies become available. In the last few years, interest in making DBMSs customizable for new application areas has magnified this need for flexible specification of execution strategies in a query optimizer. Several researchers have recently proposed query optimizers that are generated from rules for transforming plans into alternative plans. However, little progress has been reported on developing an implementation design that satisfies the requirements for high degrees of both flexibility and performance in an extensible query optimizer.

This paper presents a design for implementing a query optimizer that interprets a new kind of *compositional* rules for specifying alternative execution strategies that are input to the optimizer *as data*. Modifications to these function-like rules do not necessitate recompilation of the query optimizer, providing greater flexibility. Yet the interpretation, which resembles a macro expander, is so simple that a large number of rules can be processed efficiently. We describe the interpreter's data structures and algorithm, and relate these to the experience we gained from implementing an experimental prototype of this interpreter for the Starburst extensible database system at the IBM Almaden Research Center.

[*]Current address: MIT, Cambridge, MA 02139
[†]Current address: ECRC, Arabellastr. 17, D-8000 Muenchen 81, West Germany

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

1 Introduction

New advances in relational database management system (DBMS) technology have introduced problems as well as benefits. New access methods, join algorithms, etc. now enable us to process queries more quickly and efficiently than before. But they also have highlighted the difficulty of modifying existing DBMSs to incorporate these new techniques. One component of DBMSs that has proven especially difficult to modify is the query optimizer. The task of translating a high-level, user-submitted query into an efficient query evaluation plan (QEP, or just "plan") has traditionally resulted in sophisticated, but complex, implementations of the optimizer. As a result, database implementers have been slow to incorporate advances in technology into optimizers. Extensible DBMSs make extensible query optimizers even more critical.

Query optimizers generally consist of three major components: feasible plan generation, search strategies, and cost functions. The plan generation component creates alternative QEPs for executing a given query. The order in which these QEPs are evaluated is specified by the search strategy. Cost functions are used to assess the desirability of a QEP and to select the cheapest. The goal of extensible query optimization is to develop a flexible specification of each of these components and avoid embedding specific execution plans, cost functions, and search strategies in the code. Search strategies and techniques for the costing of QEPs are fairly well understood. However, specifying alternative QEPs in a way

that facilitates extensibility has proven difficult. Our efforts have concentrated on the problem of alternative QEP generation because of the limited progress in this area to date.

A number of researchers have realized the importance of extensible DBMSs in general and extensible query optimizers in particular. Projects such as EXODUS [C*86], PROBE [DS85], GENESIS [B*86], DASDBS [P*87], Postgres [SR86], and Starburst [S*86],[H*88] have dealt with extensibility in DBMSs, and Freytag [Fre87] and Graefe/DeWitt [GD87] have addressed some of the difficulties of extensible query optimization. The difficulty of specifying the repertoire of alternative QEPs in a high-level, declarative manner has prompted Freytag and Graefe/DeWitt to propose using transformation rules to alter plans. However, existing optimizers based on plan transformation rules require sophisticated pattern-matching to determine the eligibility of rules, and at any point in the processing, a large number of rules may be eligible for application. Lohman [Loh87] has suggested strategy rules as an alternative to plan transformation rules. His "building-blocks" approach, similar to Batory's molecular approach [Bat87b] and to a functional programming language [Bac85], solves many of the problems of plan transformation rules.

This paper describes a design for implementing plan generation in an optimizer by interpreting Lohman's strategy rules, in a way that both performs well and cleanly separates the plan generation, cost estimation, and search strategy components to facilitate extensibility [Fre88]. Our discussion will concentrate on the generic problem of generating and costing plans, rather than on a specific set of strategies or algorithms for deciding among those strategies. Section 2 summarizes the structure of Lohman's rules, their use to generate plans, and the association of cost and other properties with plans. It then motivates the design we present in Section 3 by discussing the advantages and disadvantages of possible implementation approaches. We decided to develop our own special-purpose rule processor rather than use an existing rule processor, due to the structure of our strategy rules and our desire to maintain detailed control over the order of evaluation. Section 4 describes the state of the existing prototype, along with the problems that arose and the solutions that resulted from this work. In Section 5, we present the conclusions we have drawn from our work thus far.

2 The Strategy Rules

The strategy rules we use to transform an internal representation of user-submitted queries into efficient QEPs are called STrategy Alternative Rules (STARs), [Loh87]. Since a processor for STARs necessarily depends upon the structure of those STARs, we first summarize their important characteristics.

2.1 STARs

Like the rules that Graefe and DeWitt [GD87] and Freytag [Fre87] describe for transforming plans, STARs provide a high-level, declarative, implementation-independent specification of the legal strategies for executing a query. However, STARs differ from these rules in that they describe how to build higher-level constructs from primitive operators, rather than how to transform the primitive operators. They therefore resemble more the molecular rules in GENESIS, which Batory uses to build complex "molecules" out of simple "molecules", which ultimately are combinations of "atoms" [Bat87b]. Batory uses this approach for all of GENESIS, whereas STARs are used only in query optimization in Starburst.

STARs resemble a functional programming language [Bac85] in both function and form. Each STAR defines a named, parameterized object in terms of one or more alternative definitions. Each alternative definition may, in turn, be defined in terms of (invoke) one or more other STARs or primitive operators called LOw-LEvel-Plan-OPerators (LOLEPOPs), specifying arguments for the parameters. The arguments may themselves be invocations of STARs or LOLEPOPs. A query evaluation plan (QEP) is a nesting of invocations of LOLEPOPs, which, when interpreted at run-time, execute the given query. The inputs to a LOLEPOP are specified as parameters of that LOLEPOP. As in the definition of a function, each alternative definition may have a condition which determines the applicability of that alternative. The conditions for all alternative definitions of a STAR need not be exclusive: if the conditions on multiple alternative definitions are true, all alternatives having a true condition will be invoked, and multiple (alternative) plans may be returned. Conditions may reference the parameters of the STAR as well as global information available to the optimizer, such as catalog information and the number of buffers.

STARs also resemble somewhat the production rules of a grammar. The named, higher-level constructs defined by STARs are analogous to "non-terminals" in a grammar, and the LOLEPOPs are similar to "terminals". However, STARs differ from productions in that they allow conditions of applicability for each alternative, invocation parameters, and the generation of multiple plans (grammars seek to find a unique parsing of an input set of tokens).

The following example illustrates a simplified set of STARs to access a given table. The *structure* of STARs, not the specific details of the functionality of these simplified STARs, is the important concept here. In all examples, we will print non-terminals (STAR names) in normal font, terminals (LOLEPOPs and constants) in **bold**, and parameters in *italics*. The parameter T is the stored table, C is the set of columns to be accessed, P is the set of predicates to be applied, i is the index to be used, and StoremType is the type of storage manager for a database object [1].

$$\text{TableAccess}(T, C, P) =$$

$$\left[\begin{array}{l} \text{TableScan}(T, C, P) \\ \forall i \in \mathbf{I}(T) : \mathbf{GET}(\text{TableScan}(i, \{\mathbf{TID}\}, P), T, C, P) \\ \quad \text{IF } \mathbf{CONDITION1} \end{array}\right.$$

The TableAccess STAR accesses a stored table and returns the specified columns of the stream of tuples that satisfy the given predicates. When TableAccess is invoked, the alternative definitions on the right side of the equality that have true or non-existent conditions of applicability will *all* be invoked (denoted by the *square* bracket). Since TableScan has no condition, it will always be invoked to scan the table. If **CONDITION1** is true, the second alternative will be invoked once for each i in the set $\mathbf{I}(T)$ of indices of table T to access index i, and the resulting stream of tuple identifiers (TIDs) will be used to **GET** columns C from table T.

This example is a simplification of the TableAccess STAR actually used in Starburst. To keep the example concise, we have allowed the invocation of TableScan in the second alternative to go unnecessarily to the data pages even if the required columns can be obtained from the index. That alternative will also apply the set of all predicates P twice, once while accessing the index and again while accessing the data pages. This redundancy can be eliminated using slightly more sophisticated STARs.

$$\text{TableScan}(T, C, P) = \left\{\begin{array}{l} \mathbf{ACCESS}(\mathbf{Heap}, T, C, P) \\ \text{IF Storage}(T)=\text{'heap'} \\ \mathbf{ACCESS}(\mathbf{BTree}, T, C, P) \\ \text{IF Storage}(T)=\text{'BTree'} \\ \mathbf{ACCESS}(\mathbf{RTree}, T, C, P) \\ \text{IF Storage}(i)=\text{'RTree'} \end{array}\right.$$

The TableScan STAR will map to *only one* (denoted by the *curly* brace [2]) of three (exclusive) alternative definitions – a **Heap BTree** or **RTree** access – depending on the type of the storage manager of table T. Since **ACCESS** is a LOLEPOP, **Heap BTree** and **RTree** are constants, and T, C, and P are parameters that will be instantiated, the resulting plan is fully specified in terms of known quantities. A valid QEP has thus been generated that will execute the given query at run-time.

As STARs are invoked, non-terminals are replaced with alternative definitions, and arguments are substituted for parameters. This process continues until all non-terminals have been replaced with LOLEPOPs, at which point a set of valid QEPs has been created.

The advantages of STAR-based query optimization are detailed in [Loh87]. While plan transformation rules frequently involve complex pattern-matching to determine the eligibility of a rule, STARs are invoked directly by name, subject only to the condition of applicability for that invocation. Furthermore, the resulting hierarchy of STAR invocations has a fanout that is limited by the number of alternative definitions for each STAR. In this example, TableAccess has only two alternative definitions, while TableScan has only three.

2.2 Properties of Plans

Properties are characteristics of a plan that describe the net result of the initial properties of its tables and the work done by that plan [GD87], [Bat87b], [RH86]. We group properties together as a property vector. Unlike Graefe and DeWitt [GD87], we treat the plan's estimated execution cost as any other plan property. Example properties include relational properties (such as tables and columns accessed, and predicates applied thus far), physical properties (like the order of tuples, and the site of the result), and estimated properties (including the cost to produce a plan and the cardinality of the result) [Loh87]. Properties are initially derived from the database catalogs for each stored table or access method used by a plan, and are subsequently altered by the addition of LOLEPOPs to the plan. The changes that are made by a LOLEPOP to a plan's properties are defined by a property function for that LOLEPOP type, which is defined in the C programming language. Once our STAR processor has reduced a STAR invocation to a nested invocation of LOLEPOPs, the property function for each of its LOLEPOPs is invoked to derive the resulting properties of that plan. The query optimizer will keep the cheapest QEPs having distinct properties, which we will call the set of "good plans".

2.3 "Glue" Mechanism

Some LOLEPOPs, notably the **JOIN** LOLEPOP, require certain properties of their input streams. For ex-

[1] In Starburst, an object such as a table or index can be maintained in the database in different data structures by alternative types of storage managers [LMP87].

[2] The distinction between a square bracket and a curly brace is redundant and for readability only; whether the condition functions for all alternatives are mutually exclusive or not is determined solely by their definitions.

ample, all dyadic LOLEPOPs (**JOIN**, **UNION**, etc.) require both input streams to be co-located at the same site, and the sort-merge flavor of **JOIN** requires both streams to be sorted on their respective join columns. This prompted the need for a "Glue" mechanism that finds the cheapest plan that satisfies a certain set of required properties [Loh87]. "Glue" augments each QEP in the set of "good plans" for a given table T (to be discussed in Section 3.2.1) with additional LOLEPOPs, so that its properties match those required, and the cheapest of these augmented plans is returned. For example, any plan not in the required tuple order would have a **SORT** LOLEPOP added, and any plan not resulting at the required site would have a **SHIP** LOLEPOP added. As with other execution strategies, the Glue mechanism can be specified using STARs; space constraints prevent our giving the details here.

2.4 The Implementation of STARs

Having summarized the form and important characteristics of STARs, we turn now to the implementation of a rule processor for the STARs. The design of the STAR processor was motivated by the desire to produce an extensible, flexible optimizer that gives us control over the order in which STARs were processed. We considered a number of alternative architectures in light of our objectives. This section discusses these objectives and how well each alternative satisfies them. To meet all of these objectives, we decided to develop our own interpreter for STARs, whose design is described in the next section.

Our first objective was to separate the implementation of plan generation, search strategy, and cost functions, to isolate each component from changes to the other [Fre87], [Fre88]. Our second objective was to exercise detailed control over the order of evaluation among alternative definitions and among arguments within a STAR. Control should not be surrendered to the defaults of a compiler or an interpreter. It should be possible for the database customizer (DBC) who defines the STARs to initialize these orders, and for the STAR processor to alter them automatically during processing, For example, we may choose to evaluate the easier arguments before the more difficult ones, or a function before its arguments. If the alternative definitions of the function are all false, a potentially lengthy argument list may not have to be evaluated. In addition, maintaining control over evaluation gives us the opportunity to exploit parallelism in execution.

With these objectives in mind, we considered several architectures for implementing STARs. Given the similarity of STARs to functions, one way to implement each STAR is as a function defined in the C programming language[3]. In the body of each STAR's definition, each alternative definition would be an expression in C invoking other functions (STARs or LOLEPOPs), preceded by its condition of applicability as an IF construct. The C compiler would automatically map the arguments of each function invocation to the parameters of its definition, so no "STAR processor" would need to be constructed! Although this procedural method is relatively easy to understand and implement, and exploits the natural function-like specification of the rules, it relinquishes control of the order of evaluation of the rules to the compiler, which automatically evaluates arguments before functions [KR78]. This inflexibility made the procedural method too restrictive for our purposes.

Another possible way to represent STARs is as PROLOG rules. In such an implementation, the STAR processor would be a PROLOG interpreter. The inherent unification capabilities of a PROLOG interpreter would simplify our work; however, a PROLOG interpreter is more general-purpose than we need, and does not exploit the hierarchic structure inherent to query optimization. For example, the order in which tables are joined must be decided before any access paths for those tables can be picked, since the join sequence determines what predicates are eligible to be applied by an access path. As Graefe and DeWitt discovered in a prototype, general-purpose rule interpreters usually perform poorly in comparison to specialized interpreters and compilers which exploit the structure of the application [GD87].

The similarities of STARs to the production rules of a grammar prompted us to consider a compiler generator like YACC [UNI84] to process STARs. However, STARs are sufficiently different from productions to make this difficult. STARs have conditions of applicability and parameters, which productions do not permit. Furthermore, grammars are typically used by parsers to find one sequence of terminals that match an input stream of tokens, whereas we wish to generate *all* such sequences.

Finally, STARs' resemblance to functional programming suggests a functional programming processor to evaluate STARs. Although some prototype processors for functional programming languages have been reported [Bac85], to our knowledge, there is no existing processor for a major portion of any such language. Additionally, processors for functional languages are no more likely to allow us control over the order of evaluation than the compilers for C, PROLOG, or any other general-purpose programming language. What we have developed is essentially a processor for a fairly large subset of a functional programming language, with the added capability to control the order of evaluation.

[3] C was considered simply for compatibility reasons with the rest of the Starburst project.

To achieve the objectives we described above and to overcome the limitations of the above alternatives, we developed a novel design. We are implementing that design in C, but STARs are not represented as C functions. To provide control over the generation of a QEP, we control the interpretation of the rules. Rather than compile the rules, as Batory and Graefe/DeWitt do, we interpret them to maximize flexibility. In an early prototype, Graefe and DeWitt tried and rejected as too slow an interpreter of their rules [GD87]. However, STARs can be interpreted more efficiently, because they avoid the pattern matching required by unification in plan transformation optimizers. Since STARs are input data to the rule processor, we do not need to recompile the entire optimizer whenever a new rule is added. Thus, the strategies available for optimizing two queries within a single program may differ, simply by changing the STAR input file! As data, STARs may be readily shipped to another site as a compact encapsulation of the strategies that any site can perform. This is likely to prove invaluable for optimization in heterogeneous distributed databases.

A potential limitation of our design is that we must rely on the DBC to define the rules properly, i.e. so that they terminate. We have no way of testing the validity of the rules in the rule base. This requirement is similar to Graefe's *soundness* and *completeness* requirements on his rule set [Gra86], [RH86]. Despite this restriction, the design, which we detail in the next section, provides the flexibility and control we needed for our rule-based optimizer to achieve the objectives we presented above.

3 The STAR Interpreter

This section describes our design for an interpreter of STARs, which is currently being implemented in Starburst. An early prototype of this interpreter has been completed, and is detailed in Section 4.

3.1 Environment

To give some context, we first describe how optimization fits into the query processing sequence in Starburst, and how the rule-based generation of plans fits into the optimization scheme. More details can be found in [H*88].

3.1.1 Query Processing Sequence

The language in which queries are submitted to our system is an extensible variant of SQL [H*88]. The steps to compile a statement into an Executable Query Evaluation Plan (EQEP) are shown in Figure 1. As the query is lexed and parsed, a semantic representation of the query called a Query Graph Model (QGM) is constructed and used to check the semantics of the query. QGM acts throughout query processing as an in-memory database for the query, caching the catalog information on the tables, columns, and predicates referenced in that query, as well as the relationships between them that are created by the query. QGM Optimization then makes semantic transformations to the QGM, using a distinct set of sophisticated rewrite rules that transform the QGM query into a "better" one, i.e., one that is more efficient and/or allows more more leeway during Plan Optimization. If alternative QGM representations are plausible depending upon their estimated cost, then all such alternative QGMs are passed to Plan Optimization to be evaluated, joined by a CHOOSE operator which instructs the optimizer to pick the least-cost alternative.

Using the QGM representation of the query as input, Plan Optimization then generates and models the cost of alternative plans, where each plan is a procedural sequence of LOLEPOPs for executing the query. The least-cost plan, called the Optimized Query Evaluation Plan (OQEP), is passed to Plan Refinement, which transforms this "skeleton" plan into a detailed Executable Query Evaluation Plan (EQEP) that will be stored in the database and interpreted at run-time. For the remainder of this paper, we will focus exclusively upon Plan Optimization, and will refer to that component simply as "the optimizer".

3.1.2 Query Optimization

For a given QGM, the top level of the optimizer performs plan optimization "bottom up", constructing new plans that join together optimal plan fragments. This dynamic programming strategy is basically the same as that used in System R [S*79] and in R* [L*85], except that it invokes our STAR processor to evaluate alternative strategies for each table access and join. It first invokes the access-path rules of our STAR processor, once for each stored table, to construct the possible plans for accessing individual tables, applying only single-table predicates. A least-cost plan is retained for each set of tables accessed thus far having different properties – e.g. having different tuple orderings, result sites, or predicates applied. These "good plans" are the "atoms" with which we construct "molecules". Next, the top level of the optimizer invokes the join rules of the STAR processor, once for each pair of tables that may be joined, to evaluate alternative plans for joining those tables. The plans created by these rules point to other plans in the set of "good plans" that produce the inputs to the join [Loh87]. The plans with various interesting prop-

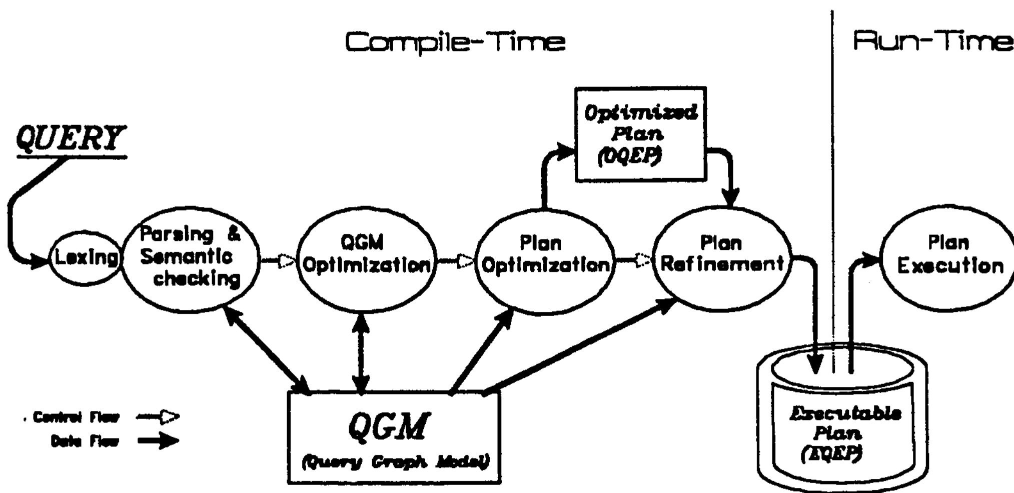

Figure 1: Major components of Starburst query processing.

erties created by the join rules are added to the set of "good plans", and are subsequently used by later invocations of the join rules, building ever larger "molecules" from smaller "molecules" and "atoms". This "building blocks" process repeats, adding more tables to the plans with more joins, until we have a "best plan" having all tables joined. This bottom-up approach ensures that new plans contain only fully optimized plan fragments, which will not be altered subsequently, and thus saves having to find and re-evaluate the cost of every plan that has incorporated that fragment.

Each of the above invocations of the STAR processor can be thought of as an invocation of a particular root STAR, so it must specify the STAR's name and arguments to instantiate that STAR's parameters. The STAR processor then uses the definition of that root STAR to expand its invocation into its alternative invocations, and so on iteratively until the invocations are fully expanded plans, i.e., are nestings of LOLEPOPs. This process will be explained fully in the next section. The STAR processor next evaluates the properties of each such plan, including accumulated estimated cost, and retains the plans having distinct properties, before returning control to the top level of the optimizer.

For example, to generate all the single-table access plans to retrieve columns NAME and ADDRESS from table **EMP** and apply predicates **EMP.SAL** < **30000** and **EMP.AGE** > **45** , the optimizer's top level would invoke the STAR processor with arguments TableAccess, **EMP**, **{NAME, ADDRESS}** , and **{EMP.SAL** < **30000 ,EMP.AGE** > **45 }** . Then the STAR processor might generate from this invocation the following two fully expanded alternative plans:

ACCESS(Heap, EMP, {NAME, ADDRESS} ,
{EMP.SAL < 30000 ,EMP.AGE > 45 })

This plan, when executed at run-time, will simply **ACCESS** table **EMP** using the **Heap** storage manager, retrieving columns NAME and ADDRESS while applying the predicates on columns SAL and AGE. The second fully expanded alternative might be:

GET(ACCESS(BTree, INDEX1, {TID} ,
EMP.AGE > 45),
EMP, {NAME, ADDRESS} ,
EMP.SAL < 30000)

(where **INDEX1** is an index of **EMP** on column AGE). This *nested* plan, when executed, will first **ACCESS** the index **INDEX1** with the **BTree** storage manager while applying the predicate on AGE, resulting in a stream of tuple identifiers (TIDs) that satisfy that predicate. This stream is then the input (first argument) to the **GET** operator, which retrieves columns NAME and ADDRESS from table **EMP** while applying the predicate on SAL. This second plan results in a stream of tuples in AGE order, whereas the first plan has no particular order (since **EMP** is stored using the **Heap** storage manager), so both plans for accessing **EMP** would be retained, and control would be returned to the top level of the optimizer.

3.2 STAR Processor Structure

We now describe how the STAR processor translates a single STAR invocation into a set of alternative, fully expanded plans whose properties have been determined, beginning with the data structures on which that processing depends.

3.2.1 Data Structures

Only four simple data structures are needed by the STAR processor: (1) the STAR array, the internal representation of STARs; (2) the set of "good plans" with distinct properties; (3) the Invocation Tree to keep track

of STAR invocations; and (4) the "To Do" list of invocations in the Invocation Tree yet to be evaluated. The first two are retained throughout optimization, whereas the last two are created in working space that is allocated for each invocation of the STAR processor, in order to minimize the space needed and to eliminate the need for garbage collection. The structure and use of each data structure will be discussed in turn.

The STAR array is an input to the optimizer that stores the internal definition of each STAR. It remains unchanged throughout optimization. The entry for a given STAR points to all of its alternative definitions. Each alternative definition contains an invocation of another STAR or a LOLEPOP, as well as the name of its (optional) condition function, which is a C function to test whether that alternative is applicable or not.

We have already mentioned the set of "good plans", which is a set of least-cost plans having distinct properties, e.g. different predicates applied or columns accessed thus far, site, and tuple ordering. When the STAR processor generates a plan whose properties have been ascertained, the estimated cost of the new plan is compared with the cost of any existing plan having the same properties, and the cheapest is retained. Though this technique was used in System R and in R*, it has been generalized to permit representation of more properties, any number of tables in a query, and to use the space more efficiently.

The Invocation Tree contains a node for each STAR invocation. Each time the STAR processor is invoked, the Invocation Tree is initialized with a single node for that invocation, as shown in Figure 2. An Invocation Tree

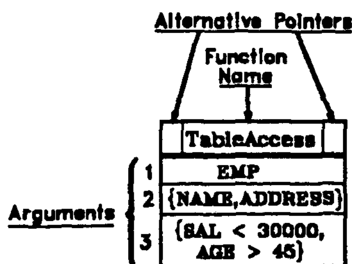

Figure 2: The Invocation Tree *before* evaluation of node TableAccess.

node contains the STAR's name and its arguments. An argument may itself be an invocation, as the function g in the nested expression $f(g(x))$ is both an argument to f and an invocation of g. In such a case, g is treated like a child of f in the Invocation Tree. This relationship of nested invocations is illustrated in the Invocation Tree of Figure 3 by the **GET** node and its first argument, an invocation of TableScan. In addition, any invocation may have alternative invocations, which are like siblings to that invocation because they are generated from alternative definitions of the same STAR, as we will see shortly. All the sibling alternative invocations are also linked together, as illustrated by the TableScan node and the two **GET** nodes in Figure 3.

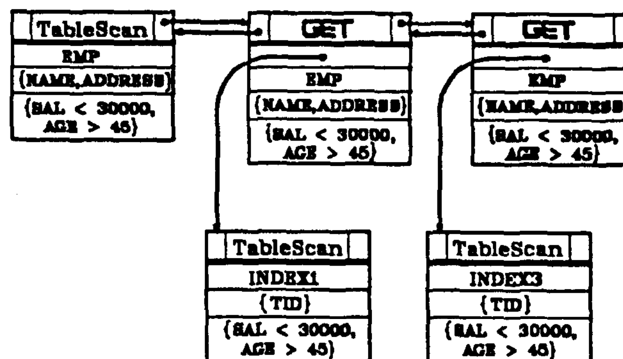

Figure 3: The Invocation Tree *after* evaluation of node TableAccess.

The "To Do" list is a prioritized queue of pointers to invocations in the Invocation Tree. It is used to decide which node (invocation) in the Invocation Tree to evaluate next. Simply by assigning a priority to each alternative, the DBC can determine the order of evaluation of invocations, achieving flexible evaluation order, one of our major objectives. This simple but extremely flexible prioritization scheme includes as a special case the simpler strategies of breadth-first search (i.e., a queue) and depth-first search (i.e., a stack). A similar strategy was used by the Exodus rule-generated optimizer [GD87], in which the priority of each plan transformation that might be applied next is automatically calculated as the expected cost benefit (called the *promise*) of that transformation.

3.2.2 Invocation Evaluation Algorithm

We now describe the algorithm whereby the STAR processor converts one STAR invocation into a set of alternative plans, each with a vector of properties. The algorithm is composed of two phases: (1) the reduction of STARs to nestings of LOLEPOPs (plans) and (2) the determination of each plan's properties. Each phase will be described and illustrated with an example separately, but as implemented, the two phases can be intermixed.

The first phase of this algorithm is essentially a macro processor, which iteratively evaluates a node in the Invocation Tree by replacing that node with its definition (from the STAR array), and substituting the old node's arguments wherever parameters appear in that defini-

tion. Any arguments that are invocations must themselves be evaluated in the same way. This evaluation continues until all invocations have been evaluated [4]. The process therefore differs little from the evaluation of a function invocation in any compiler, *except* that the function that was invoked may have *alternative* definitions (siblings), each with a condition of applicability. In addition, the order of evaluation is controlled by the priority assigned to each invocation in the "To Do" list, not by any built-in mechanism, thereby achieving our objective of flexible control over the order of evaluation.

We now discuss one iteration (evaluation) of this algorithm, and then illustrate it with an example. First the highest-priority pointer is popped off the top of the "To Do" list, and the name of the STAR invoked by that node in the Invocation Tree is looked up in the STAR array. Its entry in the STAR array points to all of its alternative definitions, each of which may have a condition function. Only those alternative definitions whose condition function either returns true or is omitted need be evaluated, which involves creating a new invocation node for it in a sibling set, and a pointer to it in the "To Do" list with an assigned priority. Parameter substitution is accomplished as in any programming language. However, if an argument A in an alternative definition is an invocation of another STAR S, then a new node is created for S – just as above – and is pointed to by A. Finally, the newly-created sibling set replaces the original invocation node in the Invocation Tree.

Using our earlier example, the invocation of the STAR processor with the invocation

TableAccess(**EMP**, {**NAME, ADDRESS**} ,
{**EMP.SAL** $<$ **30000** ,**EMP.AGE** $>$ **45** })

would initialize the Invocation Tree with just this one invocation, as shown in Figure 2 [5]. The STAR name TableAccess is looked up in the STAR array to get a pointer to its alternative definitions. Suppose as before that the definition of TableAccess was:

TableAccess(T, C, P) =

$$\left[\begin{array}{l}\text{TableScan}(T, C, P)\\ \forall i \in \mathbf{I}(T) : \mathbf{GET}(\text{TableScan}(i, \{\mathbf{TID}\}, P), T, C, P)\\ \quad \text{IF } \mathbf{CONDITION1}\end{array}\right.$$

This STAR has two alternative definitions on the right side of the equality: alternative 1 invokes the STAR called TableScan, and alternative 2 invokes the LOLEPOP called **GET**. Suppose that **CONDITION1** is true and that **I(EMP)** = {**INDEX1,INDEX3**}. Alternative 2 is repeated once for each element in $\mathbf{I}(T)$, as will be discussed in Section 3.3.2 below. So 3 new nodes will be created in the sibling set for TableAccess: one for TableScan and two for **GET**.

Parameter substitution for each new node replaces each parameter in the definition with the corresponding argument in the invocation, e.g. the parameter T is replaced by **EMP**. In the second alternative definition of TableAccess, the first argument of **GET** is an invocation of the STAR called TableScan. So a node in the Invocation Tree is created for this nested invocation of TableScan, pointed to by the entry for the first argument of the new node for **GET**, and a pointer to the TableScan node is assigned a priority and put onto the "To Do" list.

The resulting Invocation Tree, after one iteration of this first phase of the invocation evaluation algorithm, is shown in Figure 3. Note that at least three more of these iterations will be necessary, since the TableScan nodes are STARs that have not yet been fully reduced to LOLEPOPs. There will be one entry in the "To Do" list for each of them.

Once this phase has fully expanded any alternative to a nesting of LOLEPOPs, we can in the second phase of the algorithm determine its properties. We start with the most nested LOLEPOP first (usually an **ACCESS** to a stored table) and work "inside out", calling the property function of each LOLEPOP. Each property function uses the arguments of that LOLEPOP's invocation, including input plans and their properties, to set the appropriate properties in the property vector that is returned. Cost equations in the property function add the cost of this LOLEPOP to the cumulative cost of its input(s). The structure of such cost equations are well established [S*79] and validated [ML86], so we will not discuss them further here. The DBC need only adapt them to the particular methods used to implement each flavor of LOLEPOP and the machine environment.

Suppose that the fully expanded form of our example above resulted in two alternative plans:

ACCESS(Heap, EMP, {**NAME, ADDRESS**} ,
{**EMP.SAL** $<$ **30000** ,**EMP.AGE** $>$ **45** })

and

GET(ACCESS(BTree, INDEX1, {**TID**} ,
EMP.AGE $>$ **45**),
EMP, {**NAME, ADDRESS**} ,
EMP.SAL $<$ **30000**)

The first alternative has no nesting, so we need only invoke the property function for the **ACCESS** LOLE-

[4] Recursive STARs raise the possibility that this process may not terminate. We defer discussion of this problem until section 3.3.4.

[5] Actually, constants such as the table, column, and predicate names are pointed to by Invocation Tree nodes, to avoid redundancy.

POP, passing its arguments. The property function would return a property vector that summarized the cost and net effect of that plan: it will retrieve columns NAME, ADDRESS, and TID (which is by default always accessed so we can later re-access the table for more columns) from table **EMP** and apply the predicates on SAL and AGE, and the tuples will have no particular order because they were stored by the **Heap** storage manager.

The second alternative is more complicated, however. The most nested LOLEPOP is the **ACCESS** of **INDEX1**. The property function for **ACCESS** is again called, but this time returns different properties: only the TID of table **EMP** has been accessed, only the predicate on AGE has been applied, and the tuples are now retrieved in the order of the column of **INDEX1**, AGE, by the **BTree** storage manager. The cost so far is only the cost of accessing the portion of **INDEX1** that satisfies the predicate on AGE. Now we may evaluate the properties of the **GET** LOLEPOP, given its input arguments *and their properties* (in the case of its first argument). The property function for **GET** unions the names of the columns that it **GET**s to those of its input stream (defined by argument 1), and does the same with the predicates applied thus far. Thus, after the **GET** has been executed, the [columns] property now equals **{NAME, ADDRESS}** ∪**{TID}** and the [predicates] property is **{EMP.SAL** < **30000 }** ∪ **{EMP.AGE** > **45 }**. Note that, not coincidentally, these properties are identical to those of the previous alternative. However, the [order] property of the tuples in the second alternative is AGE, inherited from the input stream, whereas the order of the tuples in the first alternative is null. Costs are added for accessing the data pages and applying the remaining predicate on SAL to each tuple in the input stream.

Once an alternative's properties have been evaluated in this way, it is retained unless a cheaper plan with the same properties already exists. After this is completed for all alternative plans generated, control is returned to the top level of the optimizer.

Although the above algorithm alternately accesses the rule definitions in the STAR array, the next rule to expand via the "To Do" list, and the properties generated by the property functions, the separation of these components makes it easy to alter one component without affecting the others or the overall algorithm. For example, new STARs can be inserted in the STAR array without affecting the property functions, so long as the same LOLEPOPs are referenced. Similarly, a new search strategy can be implemented by simply changing the priorities assigned to each invocation before it is inserted in the "To Do" list. Thus, our design has achieved our objectives of both extensibility and complete control over the order of evaluation of STARs.

3.3 Special Problems

We now discuss several practical problems that arise in implementing the above algorithm, and anticipated solutions.

3.3.1 Space Management

As more STARs and more alternative definitions to STARs are defined, the Invocation Tree for any given invocation may get quite large, and thus efficient utilization of space becomes a concern. Every time a node in the Invocation Tree is processed by the above algorithm, the space of the old node may be freed, while space for one or more new nodes for each alternative is consumed. Since the nodes are variable in length, the new nodes may or may not be able to re-use the old node's space, leading to fragmentation of the space. For now, we anticipate that there will be sufficient space to generate and fully expand the Invocation Tree for any given invocation of the STAR processor without having to do garbage collection. The working space containing the Invocation Tree and "To Do" list can (and will) be reclaimed upon exiting the STAR processor, since the plans that were generated by that invocation have either been discarded or retained in the set of "good plans".

3.3.2 "For All" Mechanism

Some set-valued arguments may be treated collectively as a set, while others should be treated individually. In the example above, argument 3 of the invocation of TableAccess, the set of predicates **{EMP.SAL** < **30000 ,EMP.AGE** > **45 }** , is passed as a *collection* of predicates to be applied by TableScan (which is then passed on to **ACCESS** unchanged). However, each alternative plan returned by TableScan should have the **GET** operator applied *individually* to it, and similarly we want to apply the TableScan operator to every index in the set $\mathbf{I}(T)$. What is needed is a "FOR ALL" mechanism, which will repeatedly apply an operator to each instance of a set of arguments. This is similar to the MAP operator in LISP. As in [Loh87], we assume that alternative plans are to be treated individually, whereas any other arguments that are sets are to be treated collectively unless otherwise noted by the "FOR ALL" (denoted "∀") operator before an alternative (e.g., the second alternative of TableAccess).

Implementing this "FOR ALL" mechanism becomes more difficult when we permit multiple "FOR ALL"s within a single invocation. Unfortunately, this generalization is unavoidable, since dyadic LOLEPOPs such as **JOIN** and **UNION** may have two input streams, each of which may have multiple alternatives. We want the system to automatically generate all Cartesian products of those alternatives. For example, suppose we have an invocation of a sort-merge **JOIN** such as

JOIN(sort-merge, {**DEPT.DNO** = **EMP.DNO**}, *Outer*, *Inner*)

for which *Outer* has two alternative plans O1 and O2, and *Inner* has three plans: I1, I2, and I3. Then six **JOIN** invocations should be evaluated, one joining the streams produced by O1 and I1, one joining O1 with I2, ..., and one joining O2 with I3.

The other implementation problem is that the "FOR ALL" mechanism must iterate over sets containing different types of objects. The example above iterated over a set of alternative plans, but the TableScan STAR iterates over a set I(T) of indices.

We anticipate having an iterator mechanism for each type of set to be iterated over, which will be OPENed by specifying the set and the type of its elements. Then, each time the NEXT of that iterator is requested, it returns a pointer to the next element in the set.

3.3.3 Re-using Common Subexpressions

The Cartesian products generated by the "FOR ALL" mechanism described in the previous section are likely to generate redundant invocations. Ideally, common subexpressions could be identified and shared by STARs, so that the Invocation Tree would be a directed acyclic graph rather than a tree. However, generalizing STARs to permit any invocation to have multiple parents, i.e., allowing multiple invocations to point to the same argument, added too much complexity to the second, "inside out" phase of our STAR processing algorithm to justify the expected gain in efficiency. Instead, we are able to keep the Invocation Tree a true tree by retaining the set of "good plans" as this shared repository of subexpressions. Our bottom-up, "building blocks" approach to optimization ensures that the best plans for a subexpression will have already been constructed and stored in that set before they are needed, in most cases. Otherwise, Glue will invoke the appropriate STAR to generate and store the needed plans.

3.3.4 Recursive STARs

Thus far, only in the Glue STAR has the use of recursive STARs been required. In our simple example, recursion poses little problem, since the recursion will terminate after adding a finite number of LOLEPOPs to satisfy the finite number of required properties. In general, however, allowing recursive STARs poses the serious risk of infinite recursion, if the STARs are formulated incorrectly. It is conceivable that, as in PROLOG, different orders of evaluation could affect whether a given recursion is safe (i.e., terminates) or not. Since other malfunctions are possible with poorly specified STARs that are similarly very hard to detect, we have decided initially to permit recursion and place the burden on the DBC to ensure that STARs do not contain infinite recursion and other errors. These potential pitfalls are no different than those faced by the definer of a standard grammar, which commonly contains recursive productions. It remains an open problem whether a STAR checker could be built to automate the detection of errors in STAR specifications.

4 Prototype

An initial prototype of the STAR processor design described above has already been built, and its lessons have been incorporated into the design presented earlier. In general, the prototype proved that this design can achieve both of our major goals: clean separation of the plan generation from the search strategy and cost functions, and detailed control over the order of evaluation of rules. Therefore, little in the prototype needed to be changed in our current design. The overall architecture of the major components is unchanged, and the data structures differ only in their implementation details. However, the prototype implemented a limited subset of the algorithm. It nonetheless gave us some ideas for making the algorithm more efficient as well as more flexible. We therefore limit our discussion here to some of the more significant differences between the prototype and the current design, and what we learned from our implementation.

First of all, only a simplified version of the first phase of the algorithm described above – the reduction of STARs to nestings of LOLEPOPs – was implemented in the prototype, as its feasibility seemed the most crucial test of the architecture in general, particularly the separation of the search strategy from the general mechanism that processed each STAR. By changing the "To Do" list in the prototype from a queue to a stack, we readily changed the search strategy from breadth-first to depth-first, thus proving the independence of the search

mechanism and inspiring the more generalized priority queue mechanism discussed in Section 3.2.

Secondly, the first phase of the algorithm was slightly different in the prototype than in the design presented above. As a result, the prototype could process STARs having at most one reference to a STAR in each alternative definition. When evaluating a STAR reference, the prototype did not replace its node in the Invocation Tree with nodes for its alternative definitions, but retained the node for the STAR and linked it to "children" nodes representing its alternative definitions. Then, when all STARs had been fully expanded, the Invocation Tree had to be compressed to remove the non-LOLEPOP nodes. This separation of the plan generation phase into two separate expansion and compression sub-phases made it very easy to trace the sequence of STARs that had been invoked, but had two disadvantages. First, we were concerned that the Invocation Tree might quickly become quite large, even though the prototype did not actually require the entire expansion process to terminate before compression began. Second, we wanted the flexibility of allowing more than one STAR to be referenced in any alternative definition. Specifically, a dyadic operation such as **JOIN** is likely to invoke STARs, not LOLEPOPs, for both of its input streams. Both problems were obviated by the algorithm presented in Section 3, which permits any number of STARs to be referenced in an alternative definition, as functions and/or as arguments, and replaces nodes for STAR invocations by one or more nodes representing their alternative definitions.

Thus, the prototype both proved the operability of our design and contributed substantially to its improvement.

5 Conclusion

We have presented a design for implementing a modular, extensible query optimizer based upon a set of declarative rules, called STARs, for representing alternative query execution strategies. By interpreting these rules, we gain the flexibility of treating them as data that is an input to the optimizer, and so can readily change the strategies without having to re-compile the optimizer. By building our own interpreter of STARs rather than using a general-purpose programming language, we retain complete control over the order of evaluation of STARs. This permits us to experiment with alternative search strategies, and ultimately to "fine tune" those strategies for individual applications. We have implemented a prototype of our STAR processor to prove the feasibility of our design, and have incorporated the lessons of that prototype into our current design. Our modular design cleanly separates the three major components that generate execution plans, estimate the cost and other properties of those plans, and determine the next plan to generate. The design is also quite simple, requiring only four data structures and a straightforward, two-phase algorithm that first expands STARs into plans like a macro processor, and then evaluates the properties of those plans to find the cheapest. This simplicity enhances both extensibility and performance.

In retrospect, our STAR processor actually may have much wider applicability than just query optimization. Since STARs so closely resemble functions, our interpreter should be able to interpret a subset of a functional programming language. STARs may also be viewed as parametrized productions in a generalized grammar, in which a reference to a non-terminal in the grammar triggers its production only if some additional conditions are met. The expanded universe of languages whose grammars might be directly interpretable with our STAR processor has yet to be characterized.

In the immediate future, we plan to expand our prototype to implement the complete optimizer, to integrate it with the rest of Starburst, and to address some of the open issues we bypassed in developing the prototype. Unfortunately, we were unable to compare the performance of our interpreter with that of the compiled, procedural approach. The STAR array is currently initialized manually in the code, but we hope to build a front end that will permit the DBC to specify STARs in a high-level functional programming language, and possibly do some preprocessing of STARs such as checking, optimization, and early binding.

6 Acknowledgements

We would like to thank several colleagues for their contributions to this work. Laura Haas supplied helpful guidance and constructive critique throughout the evolution of our design and prototype. Ed Wimmers provided references, discussion, and feedback on the relationship of our work to that of functional programming. Shel Finkelstein, Laura Haas, John McPherson, Pat Selinger, Irv Traiger, and Ed Wimmers all reviewed at least one earlier draft of this paper, enhancing its readability substantially.

References

[Bac85] J. Backus. *From Functional Level Semantics to Program Transformation and Optimization.* Tech. Rep. RJ4567, IBM Almaden Res. Ctr., San Jose, CA, Jan. 1985.

[B*86] D.S. Batory et al. *GENESIS: An Extensible Database Management System.* Tech. Rep. TR-86-07, Dept. of Comp. Sci., Univ. of Texas at Austin, March 1986. To appear in *IEEE Trans. on Software Engr.*.

[Bat87a] D.S. Batory. Extensible Cost Models and Query Optimization in GENESIS. *IEEE Database Engr.*, 10(4), Nov. 1987.

[Bat87b] D.S. Batory. *A Molecular Database Systems Technology.* Tech. Rep. TR-87-23, Dept. of Comp. Sci., Univ. of Texas at Austin, June 1987.

[C*86] M.J. Carey et al. The Architecture of the EXODUS Extensible DBMS: a Preliminary Report. In *Procs. of the Intl. Workshop on Object-Oriented Database Systems*, IEEE, Asilomar, CA, Sept. 1986.

[DS85] U. Dayal and J. Smith. A Knowledge Oriented Database Management System. In *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, CA, Feb. 1985.

[Fre87] J.C. Freytag. A Rule-Based View of Query Optimization. In *Procs. of ACM-SIGMOD*, pages 173–180, San Francisco, CA, May 1987.

[Fre88] J.C. Freytag. Towards a Uniform Approach to Query Optimization. Working paper, IBM Almaden Res. Ctr., San Jose, CA, Feb. 1988.

[GD87] G. Graefe and D.J. DeWitt. The EXODUS Optimizer Generator. In *Procs. of ACM-SIGMOD*, pages 160–172, San Francisco, CA, May 1987.

[Gra86] G. Graefe. Software Modularization with the EXODUS Optimizer Generator. *IEEE Database Engr.*, 9(4):37–43, Dec. 1986.

[H*88] L.M. Haas et al. *An Extensible Processor for an Extended Relational Query Language.* Tech. Rep. RJ6182, IBM Almaden Res. Ctr., San Jose, CA, Feb. 1988.

[KR78] B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice Hall, Inc., Englewood Cliffs, NJ 07632, 1978.

[LMP87] B.G. Lindsay, J.McPherson, and H. Pirahesh. A Data Management Extension Architecture. In *Procs. of ACM-SIGMOD*, pages 220–226, San Francisco, CA, May 1987.

[L*85] G.M. Lohman et al. Query Processing in R*. In Reiner Kim, Batory, editor, *Query Processing in Database Systems*, page pp.31, Springer-Verlag, 1985.

[Loh87] G.M. Lohman. *Grammar-like Functional Rules for Representing Query Optimization Alternatives.* In *Procs. of ACM-SIGMOD*, Chicago, IL, June 1988 (to appear). Also available as Tech. Rep. RJ5992, IBM Almaden Res. Ctr., San Jose, CA, Dec. 1987.

[ML85] L. Mackert and G.M. Lohman. *Index Scans using a Finite LRU Buffer: A Validated I/O Model.* To appear in *ACM Trans. on Database Systems.* Also available as Tech. Rep. RJ4836, IBM Almaden Res. Ctr., San Jose, CA, 1985.

[ML86] L. Mackert and G.M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Procs. of ACM-SIGMOD*, pages 84–95, Washington, DC, May 1986.

[P*87] H.B. Paul et al. Architecture and Implementation of the Darmstadt Database Kernel System. In *Procs. of ACM-SIGMOD*, pages 196–207, San Francisco, CA, May 1987.

[RH86] A. Rosenthal and P. Helman. Understanding and Extending Transformation-Based Optimizers. *IEEE Database Engr.*, 9(4):44–51, Dec. 1986.

[S*79] P.G. Selinger et al. Access Path Selection in a Relational Database Management System. In *Procs. of ACM-SIGMOD*, pages 23–34, May 1979.

[S*86] P.M. Schwarz et al. Extensibility in the Starburst Database System. In *Procs. of the Intl. Workshop on Object-Oriented Database Systems*, IEEE, Asilomar, CA, Sept. 1986.

[SR86] M. Stonebraker and L. Rowe. The Design of Postgres. In *Procs. of ACM-SIGMOD*, pages 340–355, May 1986.

[UNI84] UNIX. *UNIX Programmer's Manual, Supplementary Documents.* Dept. of EE and CS, Univ. of California, Berkeley, CA, March 1984.