A Modular Query Optimizer Generator

Edward Siciens
Computer Science Department
Boston University

John Sieg, Jr.
Computer Science Department
University of Lowell

ABSTRACT
Rule-based query optimization is an important component of extendible database systems. In this paper we examine existing rule-based optimizers, and show that they have significant limitations. We then present a new approach to optimization based on the idea of modules. An optimizer is divided into several interconnected modules. Each module has its own set of rewrite rules and can be implemented differently according to how the rules are used. This modular approach avoids the limitations of existing systems and supports flexible optimizer prototyping, efficient optimizer-time search, and convenient access to metadata. To illustrate these ideas, we give a decomposition into modules for an example relational query optimizer.

1. Introduction

A query optimizer transforms an internal representation of a query into a query evaluation strategy. Some researchers have suggested that optimizer transformations be specified by term rewrite rules [4, 2, 15]. Rewrite rules declare that expressions can be replaced by other expressions. Code evaluating the transformed expression may be more efficient than code evaluating the original one.

An optimizer generator is a computer program that accepts as input a specification of an optimizer and outputs code for the optimizer. Optimizer generators are a recent breakthrough. One pioneering effort is the EXODUS optimizer generator [4, 5, 6]. Although the EXODUS optimizer generator has revolutionized query optimization, it falls short in several ways:
• it requires each operator to have an implementation;
• it binds cost analysis to implementations;
• it has a fixed search strategy;
• it allows only one primitive kind of modularization of optimizer code, namely into phases; and
• it performs expensive pattern matching even when unnecessary.

These shortcomings are discussed in Section 4.

Related to optimizer generators are optimizers based on rule interpretation. Starburst's optimizer [10], for example, interprets a set of transformation rules specifying the optimization strategies. It then runs these rules on an input query. The Starburst interpreter is another revolutionary effort, but it also has some notable shortcomings:
• it forces the programmer to use a non-deterministic macro-definition language to define optimizations;
• it is deeply entrenched in the relational paradigm;
• it uses a bottom-up query evaluator that is difficult to extend.

In this paper we describe the design of a query optimizer generator that encompasses many of the features of both of these systems and overcomes their limitations. The chief goal of our design is to support optimizer modules, to which conventional software engineering methodologies can be applied. Our design also emphasizes independent specifications of module features such as search strategy, rule representation, and termination conditions. We believe our approach will support the creation of more flexible, maintainable, and efficient query optimizers.

This paper is organized as follows. In Section 2 we describe term rewrite rules, which formalize optimizer transformations. In Section 3 we discuss how an optimizer can be structured into modules. In Section 4 we describe some of the properties of modules. In Section 5 we illustrate our ideas by presenting a modular description of a relational optimizer. Finally in Section 6 we present our conclusions.

2. Term Rewrite Rules

Term rewrite rules have proven to be useful in many diverse research areas [11, 7, 3]. For example, rewrite rules can be used to synthesize programs. Because the job of a query optimizer is to synthesize a program from a specification, it is no surprise that optimizers can be specified using term rewrite rules.

Definition: Let V be a set of variables, and let F be a set of function symbols such that V ∩ F = ∅. Then t is a term over V and F if either:
(a) t is in V;
or
(b) t is a 0-ary function symbol in F;
or
(c) t = f(t₁, . . . , tₙ), where f is a n-ary function symbol in F and t₁, . . . , tₙ are terms.

In this paper we use the convention that variable names begin with capital letters and function symbols begin with small letters.

A term rewrite rule is an expression t₁ r t₂, where all the variables in t₂ also occur in t₁. A substitution maps variables to terms. If s is a substitution, then for any term t the expression s(t) denotes the term obtained by replacing all variables V in t by s(V). Suppose that t₁ r t₂ is a term rewrite rule. Let t
be a term containing a subterm \( t' \) such that there is a substitution \( t \) with \( t(t') = t \). Let \( t' \) be the term obtained by replacing \( t \) in \( t' \) by \( t(t') \). Then we say that \( t \) \textit{rewrites to} \( t' \).

Expressions in relational algebra can be treated as terms by letting relation names be constant symbols and operators be function symbols. Algebraic optimizations then correspond to rewrite rules.

Example: The relational algebra expression

\[ \sigma_{A \neq C}(r \cup \sigma_{A = C}(s)) \]

corresponds to the term

\[ \sigma_{A = C}(r \cup \sigma_{A = C}(s)) \]

The algebraic transformation that moves a selection inside a union can be written as the following rewrite rule:

\[ \sigma(X, \cup(E1, E2)) = \cup(\sigma(X, E1), \sigma(X, E2)) \]

Here \( X \), \( E1 \), and \( E2 \) are variables. This rule rewrites the above term to

\[ \cup(\sigma(A = C, r), \sigma(A = C, \pi("ABC", s))) \]

which is the transformation of

\[ \sigma_{A = C}(r \cup \sigma_{A = C}(s)) \]

Rewrite rules are not powerful enough to express every useful transformation. Thus we allow \textit{conditions} to be attached to rules, as in [5]. A rule’s condition is written in a host language, and is copied into the optimizer code at optimizer generation time. When a rule’s left hand side matches a term during optimization, its condition is checked. If the condition fails, the matching is considered a failure. The condition code does not access the database.

Example: The transformation that moves a selection inside a join can be written as the following rewrite rule:

\[ \sigma(X, \pi((Y, Z))) \Rightarrow \pi(\sigma(X, Y), Z) \]

with the condition that all the attributes in \( X \) appear in \( Y \).

Terms are analogous to trees. Though trees are a reasonable way to represent algebraic expressions for the user, they are insufficient for a query optimizer. The primary reason is that they do not model the sharing of expressions. Expression sharing is an important technique of multiple query optimization [13, 16]. We therefore extend our definitions to allow \textit{labelled terms} [1]. For example, the term \( f(L1: g(Y), Z, L1) \) denotes a strategy similar to the one denoted by \( f(g(Y), Z, g(Y)) \), except that it requires that the common subexpression \( g(Y) \) is shared: its value is computed once. Labels have also been used in the term rewrite system community to represent cyclic structures. For example, the term \( L1: \pi((\cup(L1, L1), s)) \) might represent a query plan whose results become part of the join's first operand. This kind of plan is useful in evaluating queries against deductive databases [9].

Labels also make it possible to distinguish among various occurrences of the same function name in the code (e.g., \( C \)) attached to rules. For example, associativity of the join operator can be declared by the rule

\[ L1: \text{join}(X, L2: \text{join}(Y, Z)) \Rightarrow L1(L2(X, Y), Z). \]

As in EXODUS [4], this enables the C code to attach properties to distinct occurrences of an operator.

Another useful extension of terms is to allow variables to range over function names. An example of such a rule is the following from [3], which pushes any function inside an \( if \)-term:

\[ X \text{ if } (T_1, T_2, T_3) \text{ if } (T_1, \text{X}(T_2), X(T_3)) \]

One final extension that is useful for query optimization is the ability for a rewrite rule to match terms of arbitrary arity. An example of such a rule is:

\[ \text{and}(\ast H, C1, C2, C3) \text{ and}(\ast H, C1, \text{and}(C2, C3)) \]

The name \( \ast H \) is called a \textit{multivariable}. Multivariables are symbols representing varying numbers of variables. They are indicated by prefixing a star to the name of the variable. Thus the above rule transforms a multi-way and term into a right-associative tree. The rule can be thought of as a shorthand for the following infinite set of rules:

\[ \text{and}(C1, C2, C3) \text{ and}(C1, \text{and}(C2, C3)) \]

\[ \text{and}(H1, C1, C2, C3) \text{ and}(H1, \text{C1, and}(C2, C3)) \]

\[ \text{and}(H1, H2, C1, C2, C3) \text{ and}(H1, H2, C1, \text{and}(C2, C3)) \]

Rewrite rules using multivariables allow the specification of strategies involving arbitrarily long lists of arguments, and are useful enough that an optimizer generator language should support them. Many of the rules of [2] can not be expressed without using multivariables. Multivariables are not available in EXODUS [4] or Starburst [10].

3. Modules

A query optimizer is a large piece of software, and so its structure should be guided by standard software engineering principles. Decomposing an optimizer into modules makes it easier to write, modify, and maintain. Decomposition into modules also has several other advantages specific to query optimization. First, by thinking of each module as an "expert" in a particular aspect of optimization,rewrite rules become easier to formulate; the database system implementor need not think about different areas of expertise at the same time. Second, because the rewrite rules in each module can be used only when that module is active, it is much easier to control the order in which the rules are applied. Third, each module can evaluate its rewrite rules differently. That is, rewrite rules for a given module may have a particularly nice structure, which allows them to be implemented more efficiently than in general. Moreover, different search strategies, termination conditions, and cost evaluation functions may be more appropriate in different modules.

A simple example of the importance of modularizing rewrite rules occurs in [15]. Suppose that the optimizer uses the following two rules to determine the join order of a query:

\[ \text{join}(*A, \text{join}(X, Y), *B) \text{ join}(*A, X, Y, *B) \]

\[ \text{join}(*A, X, Y, *B) \text{ join}(3, X, Y, *B) \]

Repeated application of the first rule changes a tree of two-way joins into a single multiple-way join; repeated application of the second rule nondeterministically chooses a particular join order. Each of these rules corresponds to a different activity, and should not be intermixed. That is, the first rule should be
used only to remove the user-specified join order in a query. The second rule then chooses appropriate join order, and thus should be activated only after the first rule can no longer be applied. Placing these rules in different modules ensures that they will be used correctly.

The EXODUS optimizer uses phases to control the applicability of rewrite rules. Each rule is assigned to one or more phases. The rules in one phase are completely applied before the rules in the next one. Thus each phase corresponds roughly to a module. However, the use of phases has two important disadvantages. First, phases do not provide the other advantages that modules have, such as different implementations, search strategies, etc., for each module. Second, phases preclude bidirectional communication. For example, one phase cannot request another phase to perform a subtask and return the results of that subtask. In addition, a phase cannot base cost estimates on implementations generated by a later phase.

We propose two mechanisms by which modules can communicate with each other: communication can be either explicit or implicit. Explicit communication is performed via call terms in rewrite rules. For example, suppose that module m1 contains the following rewrite rule:

\[ f(X, Y) \rightarrow f(Y, \text{call}(m2, X)) \]

If module m1 matches a term against the left-hand side of this rule, it passes the subterm matching X to module m2. Module m2 elaborates this term into a set of terms T, and returns the set back to module m1. Module m1 then replaces the subterm call(m2, X) in the right-hand side of the rule by each term in T.

Implicit communication between modules occurs when the output of one module becomes the input of another. A module elaborates each of its input terms and passes the resulting terms as input to another module. Such a sequence of modules is called an assembly line. The input to the first module in the assembly line is a term that represents the query. The inputs to the other modules are terms output from their predecessors. The output of the final module is a term (or set of terms) denoting the output of the optimizer. This form of communication between modules is implicit, because the result terms are sent to the next module automatically. Note that implicit communication is different from explicit communication: when a term is sent implicitly to a module, its elaboration is not returned.

In a modular query optimizer, each term that is input to an assembly-line module can be thought of as a strategy for executing a query. The function of a module is to transform its input into a set of more detailed strategies, which is then sent to the next module on the assembly line. The output of the final module corresponds to a set of optimized query plans.\(^2\) There are many ways in which this elaboration can be done, some of which are described in Section 4. However, note that there is no real need for a module to send its output set all at once; instead, the output terms for a module may be sent to its successor module as they become known. Thus terms can be pipelined through the assembly line, allowing modules to execute in parallel.

Much of the work in transformation-based query optimizers (e.g., [5, 8]) takes the approach that the input and output of the transformational part of the optimizer are expressions of a single algebra. All of the expressions generated by a module (including the input and output) must be equivalent with respect to that algebra. Consequently, an optimizer in their approach has two distinct components: the part that does algebraic transformations of terms, and the part that finds implementations of terms. We consider this approach to be too limiting. Instead, each module should be able to perform transformations according to its own algebra. For example in the relational optimizer of Figure 9 (see Section 5), the simple J-strategy module uses transformations from the relational algebra, while the full J-strategy module uses transformations from the sql-algebra [18]. Each module (and its algebra) models more implementation details than its predecessor. Consequently, the process of finding implementations is just another aspect of query transformation, and can be done gradually over several modules. Handling implementations in this way makes an optimizer generator language much more flexible and expressive.

4. The Structure of Modules

In the previous section we discussed how modules are interconnected. We now examine the structure of each individual module. Each module processes its rewrite rules in its own way, using strategies of various complexity. It may also take advantage of certain capabilities of its rules and ignore others. We thus would like a module to be able to specify the way in which it uses its rewrite rules. In this way, the optimizer generator can choose the most appropriate implementation of the module.

For reference, Figure 1 shows a template module specification for our optimizer generator language OGL. Each of the features of this template will be discussed in this section.

4.1. Independent Implementation of Modules

Each module in a query optimizer uses its set of rewrite rules to elaborate its input strategies into output strategies. In principle, this job is performed as follows: The rewrite rules

\[ \text{module module-name} \]
\[ \text{access-structure-registration} \]
\[ \text{search-strategy \& heuristic} \]
\[ \quad \mid \text{branch and bound} \]
\[ \quad \mid \text{EXODUS} \]
\[ \quad \mid \text{simulated annealing} \]
\[ \text{cost \& no cost computation} \]
\[ \quad \mid \text{by elaborations save elaborations} \]
\[ \quad \mid \text{by elaborations do not save elaborations} \]
\[ \text{output candidates \& every term} \]
\[ \quad \mid \text{by pattern matching term-\ldots} \]
\[ \text{module finished \& after n outputs} \]
\[ \quad \mid \text{after no improvement in a rewrite} \]
\[ \quad \mid \text{after cost < time spent times n} \]
\[ \quad \mid \text{when predecessor finished and no more matches} \]
\[ \text{properties \& Church-Rosser} \]
\[ \quad \mid \text{variables only at leaves} \]
\[ \quad \mid \text{macron only} \]
\[ \quad \mid \text{no multivariables} \]
\[ \text{rules rule-\ldots} \]
\[ \text{end module module-name} \]

Figure 1: OGL Module Syntax

\(^2\)Typically, an optimizer produces a single query plan. However, it may be useful for the optimizer to produce several query plans, corresponding to different resource availabilities (such as the number of buffers available). Such a strategy is described in [17].
are repeatedly applied to each strategy in the input set of the module until some termination condition is met. The resulting set of strategies are then given cost estimates, and some number of low-cost strategies are placed in the output set. We can thus partition the operation of a module into four areas of concern: the search strategy, the cost analysis technique, the termination condition, and the output policy.

For each of these areas, there are various techniques that can be used to balance optimizer-generation-time and optimize-time performance. Each module should be able to choose which techniques it will use independently of the other modules. In addition, search strategies, cost analysis techniques, termination conditions, and output policy should be individually specifiable within a module. For example, DBI should be able to experiment with different search strategies for a given module without changing the way in which costs are computed; moreover, this change should not impact the specifications of the other modules of the optimizer.

Several search strategies have been described in the literature, and are summarized in Figure 2. We propose that the specification for each module should indicate which search strategy the module will use; the optimizer generator will then generate the proper code. We expect that for many modules the heuristic policy will be favored. Note that the relative merits of the other search strategies are largely unknown. By allowing different specifications of search strategies, we can generate query optimizers whose differences are carefully controlled. Thus it will be possible for researchers to compare search strategies, and test many of their long-held intuitions (or prejudices).

All search strategies except heuristic require costs to be estimated for terms. Again, there are several possible strategies available for cost computation. These are summarized in Figure 3. The declaration cost by elaborations specifies that the DBI wants cost computation to occur in a lower-level module, that is, one later in the assembly line. In this case, a module sends a term to its successor, and requests its cost; the successor then elaborates the term, analyzes costs according to its own cost computation technique, and returns the cost of the lowest-cost elaboration. The database system implementor can declare whether the unpruned elaborations should be saved by the successor module. Such a strategy avoids repeated cost computations by the successor module at the expense of added space. If costs are computed by explicit cost functions, then cost functions attached to rewrite rules in the module are called at optimize-time to determine the cost of the term using the costs of the terms bound to its variables.

As an example of the need for different forms of cost analysis, consider EXODUS’s optimizer generator [4]. Recall that EXODUS optimizers can be organized into phases, where each phase corresponds to a module. Each module in EXODUS evaluates the cost of a strategy by explicit cost functions. One oddity of the EXODUS design is that these cost functions must be coupled with implementations in the target access module language. So even the earliest, “most abstract” phases must have implementations for all strategies. However, it is often useful to avoid extensive cost analysis, especially at the early stages of optimization. For example, rewrite rules that serve to rearrange the query tree into intermediate forms do not need cost evaluations. We saw an example of such a rule in Section 3, namely:

\[
\text{join(*A, join2(X, Y), *B) join(*A, X, Y, *B)}
\]

The multi-input join form is only temporary; we do not expect the final query plan to use such joins. However, the EXODUS optimizer would require that we assign implementations (with associated costs) to multi-input joins.

The fact that EXODUS requires cost computation for each transformation means that it does not allow heuristic search strategies. For example, consider the rule that pushes selections inside of joins. Although such a rule does not always produce a better query, we may want to assume that it is correct sufficiently often; thus we want to apply it whenever we can, without wasting time calculating costs. However, such a strategy is impossible in EXODUS. Our example relational optimizer in Section 5 contains several modules that use heuristic search.

Because a single input term can rewrite to several other terms, OGL modules need a mechanism to declare when a term is a candidate for output. Figure 4 describes policies for deciding candidate output terms. The first option, every term, specifies the output candidate policy of EXODUS: all generated terms are candidates for output. The option when no match is followed sequences that match no rule’s left hand side to be output candidates. The option by rule switches requires the database system implementor to attach to some rules switch to module declarations. Terms generated by these rules are output candidates. The option by pattern matching is followed by a list of terms. A term that matches any of the terms in this list is an output candidate.

<table>
<thead>
<tr>
<th>cost computation</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>no cost computation</td>
<td>Strategy cost is not computed.</td>
</tr>
<tr>
<td>by elaboration save elaborations</td>
<td>cost of cheapest elaboration - Save unpruned elaborations.</td>
</tr>
<tr>
<td>by elaboration do not save elaborations</td>
<td>cost of cheapest elaboration - Do not save unpruned elaborations.</td>
</tr>
<tr>
<td>by explicit cost functions</td>
<td>Strategy cost is by pattern-matching.</td>
</tr>
</tbody>
</table>

![Table: Cost Computation](image)

**Figure 3: Cost Computation**

<table>
<thead>
<tr>
<th>search strategy</th>
<th>sources</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>heuristic</td>
<td>apply the best transformation; keep new term; throw away old; do no cost analysis</td>
<td></td>
</tr>
<tr>
<td>branch and bound</td>
<td>[19]</td>
<td>greedy hill-climbing</td>
</tr>
<tr>
<td>EXODUS</td>
<td>[5, 4]</td>
<td>transformations have ratings</td>
</tr>
<tr>
<td>simulated annealing</td>
<td>[8]</td>
<td>hill-climbing with perturbations of decreasing sizes</td>
</tr>
</tbody>
</table>

![Table: Search Strategies](image)

**Figure 2: Search Strategies**

<table>
<thead>
<tr>
<th>output policy</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>every term</td>
<td>Every module term is an output term.</td>
</tr>
<tr>
<td>when no match</td>
<td>Output a term if it matches no rule’s LHS.</td>
</tr>
<tr>
<td>by rule switches</td>
<td>Output a term if its generating rule says so.</td>
</tr>
<tr>
<td>by pattern matching</td>
<td>Output terms that match output patterns.</td>
</tr>
</tbody>
</table>

![Table: Output Candidates](image)

**Figure 4: Output Candidates**
Module termination policy determines when no more elaborations will be applied to elaborations of an input term. They are summarized in Figure 5. At the moment, the relative usefulness of each option is not clear. However, we expect that future experimentation with these termination policies in a real system will help clarify which ones are the most useful.

### 4.2. Efficient Search Strategy Implementation

Each of the search strategies in Figure 2 requires the system to locate rewrite rules whose left hand sides match a given subexpression. This process requires many-to-one tree pattern matching. Current matching algorithms for general sets of rewrite rules are fairly slow. Consequently, it has been argued [10] that pattern matching can be made more efficient by restricting the form that rewrite rules can take. In particular, the Starburst team proposes that left hand sides of rules be of the form \( f(X_1, \ldots, X_n) \) for some \( f \in \mathcal{A} \), where \( f \) is a \( k \)-ary function name, and \( X_1, \ldots, X_n \) are distinct variables. This constraint allows a rewrite to be implemented as a macro substitution. The left hand side is the macro header, the right hand side is the macro body, and the matching subexpression is the macro call. Because more than one rewrite rule may have the same left hand side up to renaming of variables, a macro call can often be expanded by more than one body. If the rewrite rules for a module are macro-like, then the optimizer generator can take advantage of this property to create a more efficient optimizer.

Besides having the property of being macro-like, modules may have other useful properties; these properties are listed in Figure 6. Each of these properties can be used to improve the efficiency of the optimizer. If a collection of rewrite rules has one (or more) of these properties, then the DBI can handle the collection as a module, and declare the module to have the properties. The optimizer generator then chooses an efficient implementation of the module.

If a set of rewrite rules is Church-Rosser [7], then a term has at most one normal form; that is, the result after applying rewrites until no more are applicable is unique no matter how the rules are applied. If a query optimizer specification is Church-Rosser, then all elaborations of an input term lead to the same output term. Thus once a strategy is elaborated, there is no need to consider alternative elaborations of the strategy; the optimizer can avoid backtracking.

The variables only at leaves property does not allow variables to range over function symbols. This constraint is made by most optimizer generators, such as EXODUS. However, higher-order rewrite rules can be useful for query optimization [2]. Thus we want to allow modules to contain higher-order terms when necessary. Modules that do not contain such general rules can be processed more efficiently [12].

A similar situation occurs with multivariates. Rules containing multivariables have a useful expressive power; however, they are also more expensive to process. By specifying that a module does not use multivariables, it can be implemented more efficiently.

### 4.3. Rule Properties

In addition to properties adhered to by all the rules in a module, individual rules may have properties. These rule properties are summarized in Figure 7.

Rules may have priorities. Of the rules that match some term, the one with the lowest priority number is fired first. This simple strategy is used in Starburst [10].

Rules may be declared to be throw-away. If a throw-away rule successfully matches a term and the condition—if there is one—is met, the resulting term is kept and the old term is thrown away. The throw-away option is useful for rules that almost always result in better terms. For example, the rewrite rule that pushes selections inside joins almost surely yields a more efficient strategy, and so the input term can be pruned from the space of candidate strategies. All rules in a heuristic-search-strategy module are of the throw-away type.

The switch-to-module property was discussed in Section 4.1.

### 4.4. Knobs

The DBI is not likely to get it right the first time. The optimizer generator language should provide knobs by which the DBI can fine tune his optimizer. A knob is a parameter, adjustable at optimize-time, whose value determines some characteristic of the optimizer. EXODUS supports a set of built-in knobs; we propose that this idea be generalized to include DBI-defined knobs as well.

The specification of a module may need values for various parameters. For example, the simulated annealing search strategy requires the specification of an initial temperature, the hill-climbing search strategy requires a hill-climbing factor, and so on. A knob can be substituted for any numerical parameter value in the specification of a module. A knob causes a prompt at optimize-time, asking the DBI to set the value for the parameter. When the DBI is finally satisfied that he has

<table>
<thead>
<tr>
<th>property</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Church-Rosser</td>
<td>(7) All elaborations of a term yield the same result. No backtracking needed.</td>
</tr>
<tr>
<td>variables only at leaves</td>
<td>[5,4] ( f(X_1, \ldots, X_n) ) where ( X ) is a variable, does not occur.</td>
</tr>
<tr>
<td>macro only</td>
<td>(10) LHS's are of form ( f(X_1, \ldots, X_n) ).</td>
</tr>
<tr>
<td>no multivariables</td>
<td>Sec. 2 No var. can stand for a sequence.</td>
</tr>
</tbody>
</table>

**Figure 5: Termination Policies**

**Figure 6: Module Properties**

**Figure 7: Rule Properties**
determined good values for the knobs, he changes the OGL knob declarations to the desired values of the knobs. Then he regenerates the optimizer. The new version of the optimizer does not prompt for knob values.

4.5. Support for High-Level Access Structures

An optimizer needs to be able to incorporate high-level access structures in its query plans. Optimizer generator languages should support rules such as "for each index I of relation R, implement a retrieval of R by a lookup via I." We propose to use access structure class registration for this purpose. An access structure class is registered by the module declaration

\[
\text{access structure class <term>}
\]

where <term> is an ordinary term. The outermost operator of the term is the access structure class name, and the arguments are parameters of the access structure class. For example, a secondary index class might be declared by

\[
\text{access structure class secondary-index(R, A)}
\]

Here R and A are variables representing a relation and an attribute. A secondary index on attribute a1 of relation r could be referenced by the term \(\text{secondary-index(r, a1)}\).

A registration conceptually creates a set of rewrite rules by doing a table lookup at optimize-time. For example, the above registration causes the optimizer to read a dictionary table describing all of the available secondary indices. An example of such a table is given in Figure 8. The following rules are represented by this table:

- \(\text{secondary-index(R, A)} \rightarrow \text{secondary-index(r, a1)}\)
- \(\text{secondary-index(R, A)} \rightarrow \text{secondary-index(r, a2)}\)
- \(\text{secondary-index(R, A)} \rightarrow \text{secondary-index(s, b1)}\)

Access structure class declarations allow rewrite rules to take advantage of optimize-time knowledge as well as optimizer-generation-time knowledge. By postponing rule generation until optimize-time, the DBI can avoid committing himself too early. The DBI is also able to fine-tune the optimizer, by replacing an optimizer-generator-time rule with optimize-time rules, and vice versa. We note that table lookup is just one way to represent optimize-time rules; another is to implement rule creation functions. We intend to explore these techniques for optimize-time rule creation in future research.

5. Sample Specifications for a Relational Optimizer

In this section we illustrate our ideas by outlining the structure of a relational query optimizer. Our optimizer has eleven modules, corresponding to eleven different optimization expertises. The configuration of these experts is shown in Figure 9. The arrows show the flow of strategies. The sequence of experts from the preprocessor to the code generator forms the assembly line. We briefly describe the functions and properties of each expert. The details of each expert, including most of the necessary rewrite rules, appears in [17].

The purpose of this section is to give a large, concrete example which shows the advantages that modules provide. The strategies used by each expert are not new. What is most important is that the rule sets for these experts all have different properties, and thus can benefit from different search strategies and implementations.

5.1. The Boolean Experts

The three Boolean experts are not part of the assembly line, but instead are called explicitly by the preprocessor. They are the CNF expert, the better-disjunct expert, and the better-conjunct expert.

The CNF expert transforms a Boolean expression to an equivalent conjunctive normal form expression. For example, evaluating the term

\[
\text{call(CNF, or(a =5, and(a >2, not(b =6))))}
\]

returns the term

\[
\text{and(or(a =5, a >2),and(or(a =5, not(b =6))))}
\]

The better-disjunct expert tries to transform a disjunct into one that is more easily handled. For example, the disjunct \(r.A =s.B +2-2\) transforms to the equijoin disjunct \(r.A =s.B\).

The better-conjunct expert tries to reduce the size of conjuncts. For example, it should transform the term \((A >3)\land((A <2)\lor(B =5))\) to \((A >3)\lor(B =5)\).
Note that the terms generated by the Boolean experts do not have implementations. For example, there is no code for \((A \geq 3) \land (B = 2)\) that can be plugged into the code for \(\sigma_{2 \geq 3} \land (A = 5)\). An optimizer generator that requires implementations of all operators (e.g. [5]), cannot use the rewrite rules to transform Boolean expressions.

The Boolean experts need no cost analysis, and so can use the heuristic search strategy. They all have the property variables only at leaves. In addition, the CNF expert is Church-Rosser, and thus can be implemented particularly efficiently.

The Boolean experts could be incorporated into the preprocessor. Doing so would make the preprocessor more unwieldy, and would prohibit the Church-Rosser declaration for the CNF expert.

5.2. The Preprocessor Expert

The preprocessor translates a high-level query into a relational algebra expression. Expressions of the form \(\pi_{A_1} \cdots A_n (G_F (r_1 \times \cdots \times r_m))\) are called project-select-join expressions (PSJ). The preprocessor performs the following steps:

1. Transform the input query into the PSJ expression \(\pi_{A_1} \cdots A_n (G_F (r_1 \times \cdots \times r_m))\).
2. Call the CNF, better-disjunct and better-conjunct experts in turn to replace \(F\) by \(F'\).
3. Pass the expression \(\pi_{A_1} \cdots A_n (G_F (r_1 \times \cdots \times r_m))\) to the simple J-strategy expert.

The preprocessor uses the heuristic search strategy. It has the properties Church-Rosser and variables only at leaves.

5.5. The Operator Implementation Expert

The operator implementation expert chooses implementations of joins and other relational algebra operators. Its search strategy is Church-Rosser. Terms with no matches are output candidates. The module has the properties Church-Rosser and variables only at leaves. It uses the heuristic search strategy.

5.5. The Projection Expert

The projection expert decides placement of projections. Its input terms contain a single projection as the outermost operator, while in its output terms projections have been pushed in as far as possible. This expert has the properties Church-Rosser and variables only at leaves. It uses the heuristic search strategy.

5.7. Temporary Relation Expert

The temporary relation expert determines when temporary relations are needed. It is Church-Rosser and has variables only at leaves. It uses the heuristic search strategy. Terms with no matches are output candidates. The output terms correspond to the P-strategies of [14].

5.8. The Process Expert

The process expert transforms a P-strategy into a partially ordered set of assignment statements, which is called a query evaluation plan (QEP). We represent QEPs as partially ordered assignments instead of a sequence of assignments in order to allow for parallel execution and multiple query optimization techniques. The process expert uses the heuristic search strategy. Its output candidates are terms with no matches. It uses both multivariables and variables at non-leaf nodes. It is declared Church-Rosser in order to eliminate backtracking. Even though an input term can generate many different output candidates, in the process expert we want to generate only one.

5.9. The Code Generator

The code generator expert translates a QEP into an access module, as in [3]. Our code generator uses the heuristic search strategy. It is Church-Rosser, but has variables at non-leaf nodes and multivariables.
6. Conclusion

In this paper, we have discussed the features that optimizer generators and rule-interpretive query optimizers should have, and have pointed out some of the failings of existing systems. We have described a new optimizer generator language with constructs that avoid failings of previous systems. Features of this language include

- support for general term rewrite rules, including multivariables and variables ranging over operators;
- modularization of optimizer code, an essential feature for large-optimizer construction;
- independent declarations for module search strategy, cost analysis, termination conditions, and implementations;
- support for efficient rewriting and pattern matching by attaching properties to modules;
- knobs, which support the fine-tuning of optimizers; and
- access structure classes and foreach rules, which support the use of high-level access structures.

Finally, we have demonstrated our capability to express powerful optimization techniques by specifying a set of modules for a relational optimizer.

Our first priority for future work is an implementation of the optimizer generator. Then we can specify and test the relational optimizer described in Section 5, with an eye towards testing the relative merits of various search strategies, termination strategies, and cost evaluation techniques. We also intend to develop specifications and OGL implementations of optimizers for a deductive database system and a materialized view refresher. We hope also to explore with OGL optimization across boundaries between programming language code and embedded queries.

References