Specifying Rule-based Query Optimizers in a Reflective Framework

Leonidas Fegaras, David Maier, Tim Sheard

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
20000 N.W. Walker Road P.O. Box 91000
Portland, OR 97291-1000
{fegaras,maier,sheard}@cse.ogi.edu

Abstract. Numerous structures for database query optimizers have been proposed. Many of those proposals aimed at automating the construction of query optimizers from some kind of specification of optimizer behavior. These specification frameworks do a good job of partitioning and modularizing the kinds of information needed to generate a query optimizer. Most of them represent at least part of this information in a rule-like form. Nevertheless, large portions of these specifications still take a procedural form.

The contributions of this work are threefold. We present a language for specifying optimizers that captures a larger portion of the necessary information in a declarative manner. This language is in turn based on a model of query rewriting where query expressions carry annotations that are propagated during query transformation and planning. This framework is reminiscent of inherited and synthesized attributes for attribute grammars, and we believe it is expressive of a wide range of information: logical and physical properties, both desired and delivered, cost estimates, optimization contexts, and control strategies. Finally, we present a mechanism for processing optimizer specifications that is based on compile-time reflection. This mechanism proves to be succinct and flexible, allowing modifications of the specification syntax, incorporation of new capabilities into generated optimizers, and retargeting the translation to a variety of optimization frameworks.

We report on an implementation of our ideas using the CRML reflective functional language and on optimizer specifications we have written for several query algebras.

1 Introduction

There have been many recent efforts to develop extensible query optimizers. Modern query optimizers must keep pace with the increasing need for new data structures and algorithms, with increased functionality resulting from algebraic operations over new bulk data types, and with new optimization techniques. This need for change and experimentation is dramatic in the case of object-oriented database systems, partly because there are as yet no well-established query algebras and optimization techniques. This need suggests that optimizers should

be specified declaratively and be composable from independent modules, to partition the optimizer specification task into chunks of independent knowledge.

The work reported here is a new language for specifying query optimizers as well as a methodology for constructing their implementations. Most current optimizer specification frameworks are concerned with knowledge engineering, that is, with organizing the optimizer specification into manageable pieces of independent knowledge. They are not so much concerned with succinct ways of representing this knowledge. Frequently large parts of the optimizer specifications are just chunks of procedural code that must be evaluated as is and are not suitable for further translation. (Note that optimizer specifications given as examples in many papers are actually just schematic forms of the real inputs.) Our intention is to develop a specification language in which optimizers can be expressed naturally in a concise and easily understandable syntax and a mechanism in which this syntax can be translated directly into executable code. Thus, in addition to the knowledge engineering aspect of specifying an optimizer (the partitioning of information), we are concerned with the knowledge representation aspect (how that information is expressed and processed). In contrast to most optimizer specification architectures, our specification framework is a complete tool that produces real code from declarative specifications that include a larger range of information. Our language can capture a wide range of optimization frameworks, in a declarative and easily extensible form. It is well-suited for experimentation and fast prototyping.

The query optimization task can be seen as a mapping from logical algebraic terms that represent database queries into physical plans that represent scripts for evaluating the queries, such that the produced plans are optimal with respect to some cost model. This mapping can be captured in the form of a *rule-based term-rewriting system* [5, 6, 1]. The optimization process involves searching the space of equivalent program forms, generated mainly by the syntactic transformations captured as rewrite rules, and by the alternative access paths for accessing the same database objects, such as by the indices for retrieving data from relational tables. When the search space is very large, considering all these alternatives exhaustively may result in an infeasible search. Consequently, search must be guided by a sophisticated control strategy that uses heuristic information and cost-estimation functions.

Rule-based term-rewriting systems are characterized by their declarative structure. They support a partial separation of control from the semantics of the rewrite rules. This separation and the independence between rules are properties that make the task of the optimizer specification more manageable. Rule-based systems are powerful tools for systematic description of query optimizers, supporting easy creation, extension, and maintenance of large systems. They facilitate error-free optimizer specifications as well as experimentation and fast prototyping.

For example, consider the following rule expressed in our language:

```
<< join(‘x,‘y,‘p1 and ‘p2) >>
       = [ << intersect(join(‘x,‘y,‘p1),join(‘x,‘y,‘p2)) >> ]
```

The first line gives the rule head, which is a term pattern with pattern variables `x`, `y`, `p1`, and `p2`, while the second line gives the rule body, which is a term construction. The logical operation `join(x,y,p)` has three parameters: the outer relation `x`, the inner relation `y`, and the join predicate `p`. This rule indicates that a join whose predicate is a conjunction of two predicates `p1` and `p2` is transformed into an intersection of two joins: the first with predicate `p1` and the second with `p2`. When a logical expression matches the head of the rule, all pattern variables are bound to the subterms of this logical expression. These subterms are then optimized by the same rewriting process. The rule body constructs a new term by replacing the variables with the new optimized terms associated with the variables.

A rule describes a pattern of a local syntactic transformation on an expression tree. Most rule-based term-rewriting systems in addition support a partial form of context sensitivity in the form of rule guards: that is, predicates that test semantic properties of the term being transformed. Experience with query optimizers and other term-rewriting systems suggests the need for more sophisticated context control. For example, query optimizers for relational databases that support sort-merge joins need to enforce sort order on some subplans [5, 9]. In addition, there are often groups of rules that should be applied in a strict or partial order. In general, there might exist a transformation script that describes a strategy for the successive application of atomic transformations. Incorporating such information into the search strategy could result in a system whose control is not completely isolated from its rewrite rules, thus lessening the benefits of rule-based systems. The alternative is to incorporate this knowledge about rule ordering as part of the rewrite context, switching context after each rule application. This approach implies that rules should not consist of pure syntactic transformations only, but include semantic transformations to the context as well.

Attribute grammars [8] suggest a method of solving this problem: they provide a declarative way of expressing both syntactic and semantic transformations. Attribute grammars have been introduced as a method to describe context-sensitive semantics on top of a context-free syntactic base. They have been used as a tool for formally specifying programming languages as well as for automatic construction of their compilers. They introduce two types of attributes: *inherited* and *synthesized*. Semantic transformation at each grammar production is indicated by specifying how these attributes are propagated up and down a parse tree. Our optimizer specification language is based on attribute grammars, but there are some important differences. Even though a rewrite rule resembles a grammar production rule, terms in our language are not parse trees and attributes are propagated not only through rewrites but also inside of terms. By using term-rewriting systems with attributes we gain many advantages [14]. The attribute grammar framework is adequate to capture other optimizer specification frameworks, such as Volcano [6], in a more uniform and concise way. In fact, attributes are generalizations of logical and physical properties used in that and other optimizer frameworks. In addition, specifications are clearer because of

the functional evaluation of attributes. Semantic and syntactic definitions are separated, yet they are both integrated into the same patterns of terms manipulated at each rule: each rule now specifies how terms are transformed and how attributes are propagated. The alternative is to maintain the rewrite context globally, performing destructive updates when a different context is needed between rewrites. This approach often leads to obscure specifications with potential for errors, especially when the rewriting system is specified by rules whose interleaving evaluation is unpredictable.

As an example of how attributes are propagated in our specification language consider the following rule taken from the domain of relational query optimization:

```
{order=exp_ord}
   << join( `x <= { order=exp_ord },
            `y <= { order=[] },
            `p ) >>
   = [ << nested_loop(`x,`y,`p) >> ]
```

This rule basically says that `join(x,y,p)` is transformed into the physical algorithm `nested_loop(x,y,p)`. The name `order` is an inherited attribute that represents the required (expected) sort order of the stream of tuples generated by a term. If a term is a join from which we expect any order `exp_ord`, then the expected sort order for the outer relation `x` is also `exp_ord`, while the expected order for the inner relation `y` is empty. That is, this rule propagates the expected order only to the outer relation of the join, because the order of the inner relation does not affect the sort order of the join result.

Our experience with the Volcano optimizer generator [6, 1] revealed that there are some routinely performed coding tasks during an optimizer specification that are highly stereotyped and therefore amenable to automation. If they are not abstracted to system-supported primitives, they consume a lot of energy during coding and they may result in obscure code, vulnerable to errors. These tasks involve effective manipulation of expression trees. Most tree transformations require decomposition of trees into their components, construction of trees from their constituent subtrees, and pattern matching. These tasks can be facilitated in a language that directly supports programming based on such operations. Examples of such languages include most pattern-based functional programming languages, such as SML [10].

For example, the following rewrite rule, in our specification language:

```
<< map(fn `x => `e) (map(fn `y => `u) `z) >>
          = [ << map(fn `y => `(subst(e,x,u))) `z >> ]
```

implements the algebraic transformation:

$$\mathrm{map}(\lambda x.e(x))(\mathrm{map}(\lambda y.u(y))\,z) \;\rightarrow\; \mathrm{map}(\lambda y.e(u(y)))\,z$$

That is, the two map functions in the left side of the rewrite rule are decomposed into function parameters and function bodies, while the function of the derived

map is constructed using the support function `subst(e,x,u)` that replaces all occurrences of `x` in the tree representation of `e` with `u`.

In addition, we need a language that is good for both optimizer specification and specification processing. The specification language should be close to the language we use when we write down the optimizer on paper. Specification processing must yield efficient optimizers and be flexible enough to incorporate changes in both processing and optimization strategy. We want the ability to adjust the behavior of optimizer implementations, by controlling how rewrite rules are compiled into procedures. This ability is very convenient when we need to retarget the optimizer specification as input to another optimizer generator, such as Volcano. Current optimizer specifications are not that easily manipulable themselves. For example, in Volcano only the transformation rules are translated automatically, the rest of the specification is more or less incorporated as is. These systems are called optimizer generators, but they are really optimizer frameworks, as they support a limited form of program generation. Much of what the user provides is bodies of certain optimizer routines. In our specification language, we have a large portion of the specification amenable to manipulation.

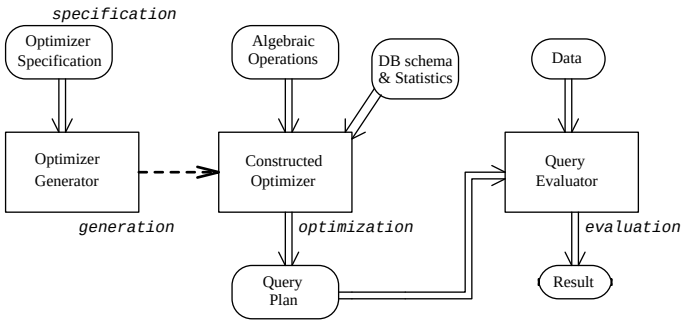

Fig. 1. The Architecture for Query Optimization and Evaluation

The ability to process the specification code during the specification process itself requires the aid of reflection. We are using CRML [13], a compile-time reflective version of SML, as the language for expressing both the specification of an optimizer and the processing of these specifications. CRML provides data structures that represent program fragments in SML. These data structures can be *reflected* into SML programs and compiled into machine code. Figure 1 shows our framework for query optimization and evaluation. The optimizer specification includes of a set of rewrite rules and a set of support functions. In our framework, rewrite rules are translated into recursive function representations that are further compiled into machine code by reflection. That way, reflection removes

a layer of interpretation by generating customized programs, tailored individually to each different rule-based system. The constructed optimizer searches the space of alternative plans by recursively calling itself at some carefully selected prespecified points. No general pattern matching is necessary during rule evaluation as rule patterns are translated into efficient programs by using standard pattern decomposition techniques. Even though we can always use a standard compiler to produce customized programs, we have this "compiler" integrated as part of the specification language, because of our ability to reflect. Therefore, implementation changes in the optimizer specification can be easily incorporated as part of the specification process. In fact, our optimizer specification language consists of a fixed (but extensible) set of macros on top of CRML. Changing the behavior of the optimizer generator is achieved by changing the definition of these macros before optimizer specification. This scheme allows a flexibility and control beyond the capabilities of other optimizer specification systems. In addition, query plans generated by the constructed optimizers are CRML program representations that can be reflected into SML programs. Query evaluation is simply a matter of defining the physical operators as SML functions and evaluating the query plans as programs. This scheme results in rapid development of an evaluation-plan interpreter.

The control strategy currently provided to the constructed optimizers is a combination of bottom-up and top-down. The user is responsible for specifying which terms at each rewrite rule will follow each strategy. Programs generated from rule-base specifications are higher-order, parameterized by parts of the rewriting process itself. This form allows easy manipulation of the control structure generated by these specifications. There may be multiple rule-base modules in our system whose control can be connected in a user-defined way, permitting hierarchical modules. Search can be controlled further by the use of special-purpose inherited attributes at each rule. Finally, the predefined control strategy itself can be changed for some rule-base modules by modifying the CRML macros that implement rule specifications.

The paper is organized as follows: Section 2 introduces CRML. We devote a large part of this paper to describing this language because we believe it is useful for many specification systems that require language extensions to be incorporated in the specification code itself. Section 3 describes the general framework on which our optimizer-specification language is based. Section 4 presents a complete query optimizer specification in the domain of relational databases. Section 5 gives the details of our specification language. Section 6 presents the optimizer specification for an object-oriented language. Section 7 presents the related work on extensible query optimizers. Finally, Section 8 concludes this paper.

2 Introduction to CRML

Compile-time reflection introduces program-generation capability into a language, thereby enabling automatic generation of many of the functions needed to

build a solution for an application. CRML [13] is an implementation of compile-time reflection for a subset of SML, built on top of the Standard ML of New Jersey compiler [10].

Reflection is the "magic" that turns data into programs. Compile-time reflection allows user-written functions to access data calculated during compilation in order to construct program representations. These representations are then transformed, by reflection, into the programs they represent. Essentially, compile-time reflection allows program representations (as data) calculated at compile-time to be type-checked and submitted to the compiler itself. The compiler turns those representations into object code that is integrated with the rest of the compiler's output.

To this end, CRML contains features that allow the language SML to be its own meta-language. Language tools for meta-programming usually consist of an *object language* in which the programs being manipulated are sentences, and a *meta-language*, which is used to describe the manipulation. In CRML, SML serves both these roles. The object language is "encoded" (represented) as SML datatypes. There is a datatype for each syntactic feature of SML. Object language manipulations are described by computations on these "representation" datatypes.

Programs written in CRML are meta-programs because they can manipulate object programs. The key features that allows CRML to be its own meta-language are the features for turning representations into programs (*reflection*) and programs into representations (*reification*). In addition, CRML contains syntactic sugar (object brackets << >>, and escape ‘) for constructing and pattern matching program representations that reflect the structure of the corresponding actual programs. Thus, meta-programs that manipulate object programs may either use the explicit value constructors for the program-representation datatypes or use this "object-language" extension to SML's syntax. This syntax allows users to write in the meta-language programs without knowing the details of the encoding program-representation datatypes.

A CRML source program consists of a series of top-level declarations, some of which are meta-programs (meant to be evaluated at compile-time) and others of which are object programs (meant to be evaluated at run-time). Compile-time functions are meta-functions because they may manipulate the representations of the run-time functions. Object programs can actually be templates with varying degrees of completeness (varying from completely specified to almost completely blank). The meta-functions provide algorithms for "filling in" the incomplete portions of these program templates. This ability to calculate programs provides important abstraction mechanisms not available in traditional languages.

One way to think of CRML is as a preprocessor that the user can program, and out of which the user obtains a source-level program with no reflection, which is compiled in the normal way. In an incrementally compiled language, such as SML, the preprocessing and compilation phases can be intermixed.

Text within object-language brackets (<< >>) represents an SML program fragment. It is parsed but not immediately compiled. Instead, the associated pro-

gram representation is returned as the value (of type `exp_rep`). Meta-language expressions may be included in the object-language text by "escaping" them with a backquote character (‘).

Inside object brackets, wherever a program fragment is expected, an *escape* operator may occur. The operand of the escape operator is expected to be an expression (in the meta-language) that provides the value to "fill in" the hole in the object code at that point. For example, if `x`, `y`, and `z` are variables in the meta-language of type `exp_rep`, then `<<if ‘x then ‘y else ‘z>>` is the program representation of the `if` expression with meta-variable `x` as its premise, `y` as its then-clause, and `z` as its else-clause.

In the meta-language, object-bracketed expressions that appear in patterns are called *object patterns*. They match the representation of program fragments with similar shape. Escape operators applied to variables inside these object-patterns represent variables in the patterns, which "bind" to corresponding sub-expressions upon a successful match. This capability makes it particularly easy to write meta-functions that perform source-to-source transformations on the representations of program fragments. A function that performs the transformations $x+0 \rightarrow x$, $x \times 1 \rightarrow x$, and $x*0 \rightarrow 0$ on an object program can be expressed easily as:

```
fun simplify e = case e of
                   << ‘x+0 >> => x
                 | << ‘x*1 >> => x
                 | << ‘x*0 >> => << 0 >>
                 |    _       => e;
```

By using the escape operator in object programs it is possible to use reflection to construct arbitrarily complex program fragments, reminiscent of macro systems. CRML supports the definition of macros in the following manner. Let `f` be a function with type $\texttt{exp_rep}^n \rightarrow \texttt{exp_rep}$. The syntax of CRML permits the form `<f< x1, ..., xn >>` as a shorthand for `‘(f(<<x1>>,...,<<xn>>))`. This syntax allows the expression-manipulating function `f` to be used as a macro. For example, a macro implementing an *if* expression in terms of the *case* construct could be implemented as:

```
fun If <<‘x,‘y,‘z>> = <<case ‘x of true => ‘y | false => ‘z>>
  | If _ = error;
```

Thus the expression `<If< k=3, 4, 5 >>` would expand to `case k=3 of true => 4 | false => 5`.

3 The Optimizer Specification Framework

Query optimizers in our framework are specified as term-rewriting systems expressed by a set of rewrite rules. Each rewrite rule describes a transformation from a logical algebra expression into either a logical algebra expression or a physical plan. Both logical expressions and physical plans are manipulated as

program representations (of type `exp_rep`). In this way, program representations that represent logical expressions or physical plans take a form that reflects the meaning of these expressions. For example, it is more convenient to write `<< if 'x then 'y else 'z >>` to represent an if-then-else expression than to use explicit value constructors to build the data structure that represents this expression. The advantage of using CRML program representations instead of user-tailored datatypes for representing expression trees is that CRML provides a rich library of functions to manipulate these trees and, more importantly, it provides a flexible and convenient reflective language to code their manipulations. Our optimizer specification language is very close to the language that one might use to specify an optimizer by writing it down on paper. This form facilitates error-free optimizer specifications as well as experimentation and fast prototyping.

Our rule-based term-rewriting system is based on attribute grammars [8]. An *attribute list* is a binding from a fixed set of attribute names to values. Each optimizer expressed in our specification language will have two attribute lists: *inherited* and *synthesized*.

The *inherited attributes* provide context during term-rewriting. Suppose that we are currently rewriting a term t. When a subterm s of t is chosen to be transformed, new inherited attribute values for s need to be computed. These new attributes values typically are derived from the inherited attribute values passed to term t before it was transformed. Therefore, inherited attribute values are propagated from one rewriting to the next. This scheme results in a purely functional term-rewriting system, because the rewriting context is passed only in the form of attributes, just as functions use parameters instead of global variables to pass information. By passing the rewriting context in a form of inherited attributes, rule-based optimizer specifications are easier to understand. Changes to context can be made locally to each rule, by specifying how the attribute values are propagated from their values before the application of the rule (that is, before this current rewriting) into the subterms chosen to be transformed next (that is, to the subsequent rewritings). In addition, memoization of the completed rewrites becomes possible, since each rewrite depends on the term to be transformed and on the inherited attribute values only.

Examples of inherited attributes include logical properties that need to be propagated and physical properties that need to be enforced. This scheme is consistent with other rule-based systems in the literature [5, 9]. One example of logical properties is schema information, such as relation names associated with their column names. Examples of physical properties include ordering of tuples in a relational expression, the site where objects reside in distributed databases, presence or absence of objects in memory, and the set of available access paths. Other use of inherited attributes is to control which rules can be considered during a rewrite. They can also be used for passing heuristic information to the main rewrite engine so that it can select more effectively which rules to apply next.

The *synthesized attributes* are attributes whose values are computed when a

rewrite is complete. They form the physical properties of the produced plans. They are passed bottom-up from the leaves of a completely transformed term (that is, the leaves of a physical plan that evaluates a query) to the root of this term. Examples include the order of a stream of tuples produced by a physical plan, the selectivity estimation of join predicates, the cardinality of a stream of tuples, and the cost of a plan. In contrast to the inherited attributes, synthesized attributes are specified independently of the rules in the rule-base. This decision was made because different rules may yield similar physical plans, and thus they may require identical code to be expressed twice to calculate the properties of those plans. In addition, there may be rules that do not produce physical plans at all (such as rules that produce intermediate logical expressions that are meant to be transformed by other rules into physical plans), and hence will not have physical properties. In our system, synthesized attributes are associated with each physical operator type.

As an example of how inherited attributes are propagated and how synthesized attributes are accumulated consider the following rule taken from the domain of relational query optimization:

```
{ order=expected_order }
     << join( `x <= { order=required_order(tables(x),p) },
              `y <= { order=required_order(tables(y),p) },
              `p ) >>
      = if subsumes(expected_order,#order(x))
           then [ << merge_join(`x,`y,`p) >> ]
           else []
```

This rule translates a `join` into a `merge_join`. The identifier `order` is an inherited attribute that represents the required order of the stream of tuples generated by a term. The pattern variable `expected_order` is bound to the current value of the attribute `order` before the transformation. The expected order propagated to the inner and outer relations is computed by the support function `required_order`, which returns the columns of the outer and inner tables that participate in the join predicate. Function call `tables(x)` returns the names of all relations referred by the logical expression `x`. Note that when a logical expression matches the head of the rule, the pattern variables `x`, `y`, and `p` are bound to the subterms of this logical expression. These subterms are then optimized by the same rewriting process. The rule body constructs a new term by replacing the variables in the rule body with the new optimized terms associated with these variables. The notation `#order(x)` in the rule body returns the value of the synthesized attribute `order` computed after the completion of the transformation of `x` (that is, this is the real order of the physical plan assigned to `x`). If the expected order of join subsumes the real order of the outer relation, then the rule returns a `merge_join`. Otherwise it returns no plan (the rule is rejected).

Each optimizer specification in our language consists of a number of rule-bases, that is, a number of independent modules of knowledge, connected in a user-defined way. Each rule-base is compiled by the CRML compiler into a higher-order function, which is parameterized by parts of the rewriting process

itself. This scheme allows easy manipulation of the control structure generated from a rule-base. The standard control strategy provided by our system is a combination of bottom-up and top-down. The user is responsible for specifying which term transformations will follow which strategy. Search can be further controlled by the use of inherited attributes. This compilation of rule-bases to programs is directed by a fixed number of small CRML macros that can be changed during the optimizer specification. This flexible scheme enables us to have rule-bases with different control strategy in a single optimizer. Note that changing the specification implementation does not require changing the specification itself. Even though the current implementation supports only one such set of macros, we are planning to support a wide range of them. There are other advantages to having the specification implementation expressed as macros: we can write these macros in such a way that the compilation of a rule-base is an input specification to another optimizer generator, such as Volcano, or a program that uses the search engine of another optimizer framework. As before, there is no need for changing the specification itself but only the macros.

4 Example: a Simple Optimizer for SQL

Before describing the complete syntax of our optimizer specification language, we present an example taken from the well-known domain of query optimization for relational databases [12], and inspired by the approach of the Volcano optimizer generator [6].

4.1 Attribute Specification

The following declaration in our specification language defines the inherited and synthesized attribute names used by the SQL optimizer:

```
<attributes< sql,
    inherited: { order: columns,
                 indices: (string * columns) list },
    synthesized: { order: columns, size: int,
                   cardinality: int, cost: real }
         = {order=[],size=1,cardinality=0,cost=max_real},
    iequal = fn (x,y) => (#order x)=(#order y),
    hash_table = lhash[997]
>>;
```

This macro defines an *attribute module* called `sql`. We may have multiple attribute modules in an optimizer specification and each module may be used by multiple rewrite rule modules. The inherited attribute `order`, which is of type `columns` (equal to the type `exp_rep list`), specifies the required order of the query output. It consists of a list of columns represented as a list of projections of the form `table.attribute`. Attribute value `indices` contains the set of available indices. Each index is represented as a list of columns (we may have

multiple-column indices). We have four synthesized attributes: `order` gives the real order of a plan, `size` the size of tuples in bytes, `cardinality` the number of tuples, and `cost` the real cost. The part after the equal sign specifies the default values for the synthesized attributes. Function `iequal` is the equality function for inherited attributes. It is used during cycle detection and memoization. Variable `lhash` is a hash table of 997 entries which is used to memoize completed rewrites.

4.2 Query Representation

Our intention here is to write a function `optimize` that will accept any SQL query and return the evaluation plan with the lowest estimated cost. For example,

```
optimize( { order=[], indices=[("emp",[<<emp.status>>])] },
          << select( [ emp.name ],
                     [ emp, paper, conf ],
                     [ emp.eno=paper.eno, emp.eno=conf.eno,
                       paper.year=conf.year, emp.status="Professor",
                       paper.subject="DB", conf.name="VLDB" ] ) >> )
```

The first argument to `optimize` gives the initial values for the inherited attributes. Here relation `emp` is indexed by its column `status`. The second parameter to `optimize` represents the following SQL query:

```
select e.name
  from emp e, paper p, conf c
  where e.eno=p.eno and e.eno=c.eno and p.year=c.year
    and emp.status="Professor"
    and p.subject="DB" and c.name="VLDB"
```

This query is translated by a simple CRML function into the following algebraic form (this is the default translation derived directly from the query):

```
<< project( join(join(access(conf,[conf.name="VLDB"]),
                      access(paper,[paper.subject="DB"]),
                      [paper.year=conf.year]),
                 access(emp,[emp.status="Professor"]),
                 [emp.eno=paper.eno,emp.eno=conf.eno]),
             [emp.name] ) >>
```

Each join predicate is a conjunction of simple predicates and it is represented as a list of simple predicates. A simple predicate is a comparison between two table columns or between a table column and a value. These list forms can be processed by the functions `listify` and `unlistify`. Function `listify` transforms an `exp_rep` that represents a list construction into a list of `exp_reps`, while function `unlistify` performs the inverse operation. The `join` operator has three parameters: an outer relation, an inner relation, and a join predicate.

```
val rulefn = <rule< rules[sql],

(* 1 *) <<project(`t,`p)>> = [<<Project(`t,`p)>>],

(* 2 *) <<access(`t,`p)>> = [<<table_scan(`t,`p)>>],

(* 3 *) {order=ord,indices=idx}
   <<access(`tbl,`p)>>
      = (map(fn (s,c) => <<index_scan(`tbl,`(unlistify c),`p)>>)
           (filter(fn (s,c) => (s=tbl) andalso (subsumes(ord,c))))
                  idx),

(* 4 *) {order=exp_ord}
   <<join(`x<={order=exp_ord},
          `y<={order=[]},
          `p)>>
      = [<<nested_loop(`x,`y,`p)>>],

(* 5 *) {order=exp_ord}
   <<join(`x<={order=required_order(tables(x),p)},
          `y<={order=required_order(tables(y),p)},
          `p)>>
      = if subsumes(exp_ord,#order x)
           then [<<merge_join(`x,`y,`p)>>]
           else [],

(* 6 *) {order=(a::r)}
   <<`x<={order=[]}>>
      = if subsumes(a::r,#order x)
           then [<<sort(`x,`(unlistify(a::r)))>>]
           else [],

(* 7 *) <<join(`x,`y,`p)>> = [<<join(`y,`x,`p)<={}>>],

(* 8 *)
<<join(join(`x,`y,`p1),`z,`p2)>>
  = let val preds = append(listify p1,listify p2)
        val p3 = join_preds(tables(y),tables(z),preds)
        val p4 = join_preds(tables(x),union(tables(y),tables(z)),preds)
    in [<<join(`x,join(`y,`z,`(unlistify p3)),
               `(unlistify p4))<={}>>] end

>>;
```

Fig. 2. Rewrite Rules for the SQL Query Optimizer

4.3 Rule Base Specification

Figure 2 specifies the rule set for the SQL query optimizer. This rule module is called `rules` and it uses the attribute module `sql`. Each rule can be seen as a mapping from a term (of type `exp_rep`) to a (possibly empty) list of terms. All these mappings are evaluated during a rewrite and all produced plans are accumulated into a list. If there is no match between the current term and the head of a rewrite rule, then the rule is viewed as returning an empty list of terms. The body of a rule can be any CRML expression that returns a list of terms. If it returns `[]`, then essentially the rule rejects the expression.

Rule 1 translates the `project` algebraic operator into the `Project` physical plan. Both `t` and `p` are pattern variables that are bound to subterms when a term matches the head of the rule `project(‘t,‘p)`. Their bindings, after they are recursively optimized, are used to construct the physical plan `Project(‘t,‘p)` in the body of the rule. Rule 2 transforms an `access` to a relation into a sequential `table_scan`.

Rule 3 generates all possible `index_scan`s for a table `t` whose order is subsumed by the expected order `ord`. The form `{order=ord,indices=idx}` before the head of the rule indicates that the inherited order attribute of the current term can be of any value `ord` and the list of available indices can be of any value `idx`. Variables `ord` and `idx` serve as free pattern variables, to be bound to attribute values before the transformation. In fact we may assign complex SML patterns to inherited attributes instead of simple variables, specifying the form of the expected attributes that are allowed to be passed to this rule. The body of Rule 3 is complex, because, in contrast to the other rules, it returns a list of terms whose length varies, depending on the number of applicable indices. (It may return an empty list). The `filter` function scans the index list `idx` to locate those that refer to the table `tbl` and have the appropriate order. Then `map` generates the `index_scan` terms for each such index. Function `subsumes` is a support function that tests if a list is a prefix of another list.

Rule 4 translates a `join` into a `nested_loop`. The form `‘x<={order=exp_ord}` indicates that the value of the inherited attribute `order` is set to `exp_ord` before term `x` is transformed by a subsequent rewriting. This form can also be expressed as `‘x`, meaning that the values of all inherited attributes are propagated to `x` as is. This abbreviation convention will not work for the second parameter `‘y<={order=[]}` because here order is changed to nil. Therefore, the head of the rule indicates that we can propagate the expected order only to the outer relation of the join, because the order of the inner relation does not affect the order of the join, as the inner relation needs to be scanned multiple times during the nested-loop join.

Rule 5 is the most interesting rule. It translates a `join` into a `merge_join`. The expected order required by the inner or outer relations is computed by the support function `required_order`, defined as follows:

```
fun required_order ( tbs, preds ) =
map(fn <<‘t.‘a=‘s.‘b>>
```

```
    => if member(t,tbs) then <<`t.`a>> else <<`s.`b>>)
  (filter(fn <<`t.`a=`s.`b>> => true | _ => false) (listify preds));
```

The `filter` call selects the equality predicates from the predicate `preds` and the `map` call selects the part of an equality predicate that refers to a table in `tbs`. That is, `required_order` returns the columns of the outer or inner tables that participate in the join predicate. Function call `tables(x)` returns the names of all relations mentioned in the logical expression `x`. The notation `(#order x)` in the rule body returns the value of the synthesized attribute `order` computed after the completion of the transformation of `x` (that is, this is the real order of the physical plan assigned to `x`). If the expected order of join subsumes the real order of the outer relation, then the rule returns a `merge_join`. Otherwise it returns no plan.

Rule 6 is invoked only when the expected order is not empty (i.e., when it matches the pattern `a::r`). The inherited order of `x` is switched to empty before `x` is transformed again. Note that if we do not set the next expected order to nil, this rule will apply indefinitely, inserting a `sort` operator each time. The body of this rule checks if the real order of the transformed plan `x` (that is, the value of the synthesized attribute `order`) is what it was expected. If it is so, it enforces a sorting using the expected order.

Rule 7 expresses the commutativity of join. The produced join is tagged with `<={}`. This syntax indicates that this term needs to be passed to the rewriter again with the same inherited attributes. This form specifies a top-down transformation. Rule 8 gives the associativity of join. It is more expensive because it involves recomputing the predicates of the new joins (details are omitted here).

The synthesized attributes for the `sql` attribute module is computed by the following macro (only the portions for `sort` and `nested_loop` are shown):

```
<synthesized_attributes< sql,
    <<sort(`x,`ord)>>
       = { order = ord,
           size = (#size x),
           cardinality = (#cardinality x),
           cost = (#cost x)+(#cardinality x)*(#size x)
                           *log((#cardinality x)*(#size x)) },
    <<nested_loop(`x,`y,`p)>>
       = let val card = (#cardinality x)*(#cardinality y)
                        *(selectivity(p))
         in { order = (#order x),
              size = (#size x)+(#size y),
              cardinality = card,
              cost = (#cost x)+(#cost y)+((#size x)*card) } end,
    ... >>;
```

This macro call indicates that the cardinality of a stream of tuples produced by a nested loop is the product of the input cardinalities with the selectivity of the predicate.

5 The Syntax of the Optimizer Specification Language

Optimizer specifications consist of calls to three CRML macros: `attributes`, `synthesized_attributes`, and `rule`.

5.1 The `rule` Macro

The `rule` macro specifies the rewrite rules. It has the form:

```
<rule< module[attributes], rule1, rule2, .... >>
```

where `module` is the name of this rule module and `attributes` is the name of a previously defined attribute module that specifies the inherited and synthesized attributes for these rules. Each rule has one of the two following forms:

```
                        head = body
{ attr=pattern, ... } head = body
```

An attribute list `{ attr=pattern, ... }` before the head of a rule indicates that each inherited attribute `attr`, passed along with the term being transformed, must match the `pattern` in the attribute list. The names of these attributes must be inherited attribute names, but they can be in any order and some of them may even be left unspecified. In the latter case the inherited attribute can be of any value. (It is ignored and passed as is to the next rewriting). If these patterns are complex, they serve as guards to the rule. The children of the node matching the head of the rule are the subterms bound to the free variables of the head pattern. After we are provided the inherited attributes for the term being transformed (that matches the `head` of the rule), we may need to pass them to its children (to the subterms), possibly with modified values. For example, if the head is `f(`x,`y)` then `x` and `y` will be bound to the left and right subterm of the matching term and they will be both transformed before the rule is applied. This recursive transformation of the subterms is a bottom-up rewriting, since subterms are transformed before the matching term. New inherited attributes are propagated to the subterm `x` by specifying `` `x<={ attr=expr,... } ``, where `attr` is an inherited attribute name and `expr` is an SML expression that depends on the variables in the `pattern`s before the rule head. Again inherited attributes can be in any order and can be selectively omitted. In the latter case the inherited values passed to the children of a term are exactly the same as the values provided before the rewriting of this term.

The `body` of a rule is an SML expression that returns a list of terms. This list may be empty (no alternative expressions or plans are produced), or it may have one element (the typical case), or it may have more than one element (as when there are many alternative access paths for scanning a table). The rule body may have one of the following forms:

```
1     if expr then body else body
2     case expr of pattern => body | ...
3     let bindings in body end
4     [ << eterm >>, ... ]
```

where `expr` is any SML expression and `bindings` are SML let bindings. Term `<<eterm>>` is the construction of a plan to be returned from this rule. Any synthesized attribute returned by the subsequent rewrites, such as those associated with the pattern variables `` `x `` or `` `x<={...} `` in the head of the rule, can be accessed in any place in the rule body by using the form `(#attr x)`, where `attr` is a synthesized attribute name.

If we want to specify a guard to a rule, such as a predicate `p`, we can write `if p then [<<eterm>>] else []` as the body of the rule. That is, if `p` is false we return no terms.

If there is a form `e<={ attr=val, ... }` in the text of `<<eterm>>` in `body`, then the term `e` is also transformed by the same rewriting process. In that case, the attributes are the new inherited attributes that are propagated to the new rewrite. This transformation is a top-down rewriting, since parts of the term are rewritten after the current rule is applied. These types of rules are used for specifying transformations from logical to logical operators (such as commutativity laws). The other types are used when we have logical to physical transformations (so that the produced plan does not need to be transformed again). The first rule type must be selected with caution, as it may yield an inefficient search. In addition, there is a danger of falling into an infinite loop, even though no rule is applied twice for the same term and the same inherited attributes. One example of such a case is the rule `{a=n} <<`x>> = [<<`x<={a=n+1}>>]`.

There is also support for memoization. When a term x, associated with inherited attributes IA, is transformed into a list of expressions $[x_1, \ldots, x_n]$ (these are physical plans because the rewrite is complete) with their synthesized attributes $[SA_1, \ldots, SA_n]$, then the tuple $(x, IA, [(x_1, SA_1), \ldots, (x_n, SA_n)])$ can be stored in a system-provided memoization table. Therefore, rewriting a term x with IA involves searching this table to test if x and IA are already memoized. If they are, the resulting rewrite is retrieved from the table; if not, the result is calculated by applying the rewrite rules.

5.2 Search Strategy

The `rule` macro produces a function `fn (X,IA,REWRITE,CYCLES,ACC,BOT) => body`, where `X` is the term to be transformed, `IA` its inherited attributes, `REWRITE` is the recursive function that calls this produced function, `CYCLES` is used to prohibit partially completed rules being invoked twice on the same term with the same inherited attributes (such as, commutativity and associativity rules), which may lead to infinite recursion, `ACC` is the function that accumulates plans (usually `union`), and `BOT` is a list of plans in which the newly generated plans from one application are accumulated. Function `REWRITE` takes three parameters: the term to be transformed, its inherited attributes, and the `CYCLES` list. It returns a list of pairs of terms with their synthesized attributes. The `ACC` function can be very sophisticated. For example, instead of forming the union of all plans each time, it can select only a small number of them based on their cost. This scheme will make the search a beam search, which is a hill-climbing search where only the best plans are considered each time.

If we define the `REWRITE` function to be a call to the function `rulefn` which is produced by the `rule` macro (such as the `rule` macro in Figure 2):

```
fun REWRITE (x,IA,cycles) = rulefn(x,IA,REWRITE,cycles,union,[]);
```

then the `REWRITE` function has the following semantics:

```
REWRITE (x,IA,cycles) =
for each rule rule#: HEAD=BODY whose head matches x do:
    { if (x,IA,rule#) in cycles then return [];
      bind all free variables Xi in HEAD;
      calculate the new inherited attributes IAi for each Xi;
      Ri = REWRITE(Xi,IAi,[]);
      for each combination of (Ti,SAi) in Ri:
          { evaluate BODY with Xi=Ti;
            for each term e<={IAe} in BODY:
                replace e with REWRITE(e,IAe,(x,IA,rule#)::cycles);
            calculate the new synthesized attributes } };
accumulate all terms and synthesized-attributes found at each matching rule.
```

Note that no explicit term matching is required in the generated code because pattern matching in SML is achieved by standard pattern decomposition. Even though the control of the rewriting appears as a combination of bottom-up (the calls `REWRITE(Xi,IAi,[])` for the subterms `Xi` of the head), and top-down (the calls `REWRITE(e,IAe,...)` for the subterms `e` of the body), it can be more general by using the inherited attributes to gain more complex search control.

A rewrite function for the SQL optimizer that supports memoization is the following:

```
fun REWRITE (x,IA,cycles) =
case table_find(x,IA) sql of
  just(r) => r        (* already memoized *)
| _ => table_insert(x,IA,rulefn(x,IA,REWRITE,cycles,union,[])) sql;
```

There can be multiple `rule` macros in a rule-base system. One way of combining them is to use the following combinator:

```
fun vertical (f,g) =
fn (X,IA,REWRITE,CYCLES,ACC,BOT)
   => g(X,IA,REWRITE,CYCLES,ACC,f(X,IA,REWRITE,CYCLES,ACC,BOT));
```

where `f` and `g` are functions generated by `rule` macros. This vertical composition accumulates all plans produced either by `f` or `g`: it fuses the rules of `f` with the rules of `g`. There are other ways of combining rules. Another one is the horizontal composition:

```
fun horizontal (f,g) =
fn (X,IA,REWRITE,CYCLES,ACC,BOT)
   => fold(fn ((a,SA),r)
              => ACC(g(a,IA,REWRITE,[],ACC,[]),r))
          (f(X,IA,REWRITE,CYCLES,ACC,[])) BOT;
```

that is, after an intermediate plan `a` is produced by `f`, the rules in `g` are applied.

A `rule` macro can be used in the body of a rule defined in another `rule` macro (because any function that returns a list of terms can be called in a rule body). This ability is useful when we have a number of rule systems controlled by another rule system. In fact we may have a hierarchy of rule systems where a non-leaf rule system controls the evaluation of its children. This framework is similar to the one described by Mitchell, et al. [11].

6 An Optimizer for an Object-Oriented Language

In this section we specify an optimizer for object-oriented queries expressed in the ZQL[C++] language [1]. ZQL[C++] is an SQL-based object query language designed to be well-integrated with C++. Our approach to solving this optimization problem is influenced by the Volcano optimizer generator approach, as described by Blakeley, et al. [1].

One example of a ZQL[C++] query is the following:

```
select e.name, e.dept.name, e.job.name
  from Employee e in Employees
 where e.dept.plant.location="Dallas"
```

In [1] this query is translated directly into the following expression:

```
project(select(Mat(Mat(Mat(get(Employees):e,
                           e.job):j,
                       e.dept):d,
                   d.plant):p,
               p.location="Dallas"),
        (e.name,j.name,d.name))
```

where the form `e:x` assigns the name `x` (a range variable) to the output produced by expression `e`. That way the results of an expression can be accessed from various points. We express this query in our notation as:

```
project(select(Mat(Mat(Mat(get(Employees,e),
                           [e.job],j),
                       [e.dept],d),
                   [d.plant],p),
               [p.location="Dallas"]),
        [e.name,j.name,d.name])
```

`Mat` is the *materialize* operator that brings a set of inter-object references (i.e., object links) into a context. For example, in `Mat(x,[e.dept.plant],v)` the range variable `v` is bound to the result of materializing the path `e.dept.plant`. Variable `e` must be a range variable defined in expression `x`. Operator `Mat`, in contrast to algebraic operators, does not construct or consume values, but it extents the bindings of its input `x`.

```
val rulefn = <rule< rules[oodb],
(* 1 *)
<<Mat(Mat(‘e,‘p,‘y),‘s,‘x)>>
  = if refers(s,y)
    then []
    else [<<Mat(Mat(‘e,‘s,‘x),‘p,‘y)<={}>>],

(* 2 *)
<<Mat(Mat(‘e,‘p,‘y),‘s,‘x)>>
  = if refers(s,y)
    then [<<Mat(‘e,‘(unlistify(append(listify(p),listify(s)))),‘x)<={}>>]
    else [],

(* 3 *)
<<Mat(‘e,‘p,‘x)>>
  = case listify(p) of
      a::b::l
        => let val nv = Id(newname())
           in [<<Mat(Mat(‘e,[‘a],‘nv),‘(unlistify(b::l)),‘x)<={}>>] end
    | _ => [],

(* 4 *) {extents=ext}
<<Mat(‘e,[‘path],‘v)>>
  = case find(fn (_,pidx) => pidx=(last(path))) ext of
      just(index,_)
        => let val nv = Id(newname())
           in [<<join( ‘e, get(‘index,‘nv),
                       [‘path=‘nv.self] )<={}>>] end
    | _ => [],

(* 5 *) {path_indices=pi}
<<Mat(‘e,[‘path],‘v)>>
  = case case find(fn (_,p) => p=path) pi of
      just(index,_)
        => let val nv = Id(newname())
           in [<<project(join(‘e,get(‘index,‘nv),
                              [‘path=‘nv.from]),
                         [to])>>] end
    | _ => [],

(* 6 *) {extents=ext}
<<select(get(‘t,‘x),[‘s=‘value])>>
  = if first(s)=x
    then case find(fn (_,pidx) => pidx=(last(s))) ext of
           just(index,_) => [<<index_scan(‘index,[‘(last(s))=‘value],‘x)>>]
         | _ => []
    else [],
  ... >>;
```

Fig. 3. Rewrite Rules for the Object-Oriented Optimizer

We use two inherited attributes. Attribute `extents` is a list of all available class extents. For example, the extent of the class `department` may be kept as an explicit set named `Departments` that contains all objects from this class. Performing a projection such as `Employees.department` can then be achieved by the join `join(get(Employees,x),get(Departments,y),x.department=y.self)`. In addition, some chains of object links are stored explicitly as path indices (or access support relations [7]). Path indices are implemented as binary relations with columns `from` and `to`. They are indexed on `from` (our optimizer can be easily extended to support inverse path indices, that is, paths indexed on column `to`). This information is stored in the inherited attribute `path_indices`. The translation of `Mat` into `join` allows us to apply the standard relational optimization methods to the domain of object-oriented databases.

Figure 3 presents some of the rules for optimizing the object-oriented algebra expressed in our specification language. Rule 1 switches the positions of two adjacent materialize operations. This transformation is possible only when the path of the outer `Mat` does not depend on the range variable of the inner `Mat`. Rule 2 collapses two materialize operations into one. This operation can be done only if the range variable of the outer `Mat` is equal to the first path component of the inner `Mat`. Rule 3 is the inverse of rule 2: it splits a `Mat` into two `Mat`s. Rule 4 translates a `Mat` into a `join`: if the `path` of `Mat` consists of only one path and there is an extent for this path in `extents` then materialize is translated into a join over this extent. The first and last operations applied to a path return the first and last component of the path, respectively. Rule 5 checks if there is a path index that implements a materialize operation. Rule 6 translates a selection over a table access into an indexed access, provided that there is an index for the selection predicate.

7 Related Work

Our work is influenced by the EXODUS and Volcano optimizer generators [1, 5]. EXODUS allows the database implementer to write a customized optimizer by providing explicit rules. The input to the EXODUS optimizer generator is a model description file, where the database implementer lists the set of operators of the data model, the set of "methods" to be considered when building and comparing access plans, the transformation rules that define legal transformations on the query trees, and the implementation rules that associate methods with operators. The EXODUS output is a customized optimizer that consists of query tree transformations derived from the model description file, support functions written in C code, and a search engine that evaluates the transformations. Volcano extends EXODUS in various ways. It provides more support for physical properties. It introduces a new family of operators, called enforcers, to be inserted in physical plans when these plans do not satisfy the expected properties. Volcano supports a flexible search engine and a flexible cost model. It also supports dynamic programming by memoizing the results of all rewrites. Our experience with the Volcano optimizer generator shows that it is the support functions, not

the rewrite rules, that requires the bulk of the effort during optimizer specification. Even though term transformations are expressed declaratively, support functions, such as propagation of physical properties, are expressed procedurally. In addition, routinely performed operations, such as construction of query trees, are not supported directly in a more convenient language. Even though our work does not add more functionality to the the Volcano optimizer, we believe that our language is far clearer and more concise.

There is a proposal for an improved interface to Volcano, called Prairie [2]. The Prairie specification forms are highly influenced by the ones of Volcano, but they are in a more abstract and declarative form. Like Volcano, Prairie supports expected and provided properties, but the distinction between them is left to the Prairie compiler (by examining the context in which they are used) and it is not an explicit part of the Prairie specification. Expression tree nodes in Prairie rules are labeled by descriptor names. Attribute propagation during a rule evaluation is specified as part of the rule itself. This specification consists of equations that reference the descriptor names in the rule trees. Manipulation of synthesized and derived properties are grouped together, in contrast to what we do. (Synthesized properties are in one place.) In contrast to Volcano, property enforcers are specified as regular implementation rules. If a physical operator, such as sort, was enforced redundantly or without any physical effect at all, it is translated into a NULL algorithm, which is basically a NOOP operation. We believe that Prairie is an improvement of Volcano since it reduces the effort of writing support functions for propagating and matching properties and provides a declarative way of specifying property enforcers.

Another extensible query optimization framework based on attribute grammars is the one for Starburst, as described by Lohman [9]. This framework is not a rule-based term-rewriting system, even though their specifications define optimizers that perform term-rewriting. Each production rule (called a STAR) defines a transformation from a high-level non-terminal symbol into a low-level one. Query trees and plans (called LOLEPOPs) are passed as parameters to these symbols. Each rule may produce multiple plans or multiple non-terminal symbols, that is, a rule is viewed as a generator of plans. The plan generation and the hierarchical evaluation of rules provide a strict order to some parts of the search, yielding a more feasible search. As in our work, they introduce two types of attributes: required (inherited) and available (synthesized). The difference is that Starburst uses these attributes for representing properties only, not as a general mechanism for context control. In order to apply the same hierarchical transformations to all subnodes of a query tree and to enforce properties to plans, they have a Glue operation that glues operators together to achieve the required properties (in the same way Volcano uses enforcers).

There are other systems that use a declarative specification language to describe specific optimizers, but most of them, as far as we know, do not provide a specification implementation. One example is Freytag's work on rule-based query optimization for relational databases [3, 4].

8 Conclusion and Future Work

We believe our specification system for query optimizers is more comprehensive and more declarative than other systems to date. We intend to produce additional optimizer specifications to validate this hypothesis, particularly ones that use attributes to control optimization context, in the manner of Mitchell, et al. [11]. We also intend to experiment with different search engines that support cost-based pruning. Another interesting research area is the development of specialized libraries of support functions to aid the optimizer specification process within a specific domain of applications. Our specification processor also appears quite flexible, and we plan to experiment with input syntax modifying and parametrizing the search strategy and retargeting the output to other search engines. CRML proved a powerful tool in constructing the specification processor. One extension we would like to see is easier extension of "object brackets" to deal with syntaxes other than that of SML. Such a capability would make it easier to experiment with the input syntax and the format for query expressions.

9 Acknowledgments

This work is supported in part by the Advanced Research Projects Agency, ARPA order number 18, monitored by the US Army Research Laboratory under contract DAAB-07-91-C-Q518.

References

1. J. Blakeley, W. McKenna, and G. Graefe. Experiences Building the Open OODB Query Optimizer. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Washington, D.C.*, pp 287–296, May 1993.
2. D. Das. *Prairie: An Algebraic Framework for Rule Specification in Query Optimizers.* Ph.D. Dissertation Proposal, University of Texas at Austin, March 1993.
3. J. C. Freytag and N. Goodman. Translating Aggregate Queries into Iterative Programs. In *Proceedings of the Twelfth International Conference on Very Large Databases, Kyoto, Japan*, pp 138–146, August 1986.
4. J. C. Freytag. A Rule-Based View of Query Optimization. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, pp 173–180, December 1987.
5. G. Graefe and D. J. DeWitt. The EXODUS Optimizer Generator. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, pp 160–171, December 1987.
6. G. Graefe and W. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. *IEEE Conference on Data Engineering, Vienna, Austria*, 1993.
7. A. Kemper and G. Moerskotte. Advanced Query Processing in Object Bases Using Access Support Relations. In *Proceedings of the Sixteenth International Conference on Very Large Databases, Brisbane, Australia*, pp 290–301. Morgan Kaufmann Publishers, Inc., August 1990.
8. D. Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

9. G. M. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, Illinois*, 13(3):18–27, September 1988.
10. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.
11. G. Mitchell, U. Dayal, and S. B. Zdonik. Control of an Extensible Query Optimizer: A Planning-Based Approach. In *Proceedings of the 19th VLDB Conference*, August 1993. To appear.
12. P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Massachusetts*, pp 23–34, May 1979.
13. T. Sheard. Guide to using CRML: Compile-Time Reflective ML. Unpublished manuscript, Oregon Graduate Institute, March 1993.
14. R. Wilhelm. Tree Transformations, Functional Languages, and Attribute Grammars. *Proceedings of the International Workshop on Attribute Grammars and their Applications, Paris, France*, pp 116–129, September 1990. LNCS 461.