The EXODUS Optimizer Generator

by

Goetz Graefe
David J. DeWitt

Computer Sciences Technical Report #687
February 1987

The EXODUS Optimizer Generator

Goetz Graefe
David J. DeWitt

Computer Sciences Department
University of Wisconsin

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the National Science Foundation under grant DCR-8402818, and by a grant from the Microelectronic and Computer Technology Corporation.

ABSTRACT

This paper presents the design and an initial performance evaluation of the query optimizer generator designed for the EXODUS extensible database system. Algebraic transformation rules are translated into an executable query optimizer, which transforms query trees and selects methods for executing operations according to cost functions associated with the methods. The search strategy avoids exhaustive search and it modifies itself to take advantage of past experience. Computational results show that an optimizer generated for a relational system produces access plans almost as good as those produced by exhaustive search, with the search time cut to a small fraction.

1. Introduction

In recent years, a number of new data models have been proposed including Daplex [SHIP81], ABE [KLUG82], GEM [ZANI83], GEMSTONE [COPE84], IRIS [LYNG86], Probe [DAYA85, MANO86], Postgres [STON86], and LDL [TSUR86]. Unfortunately, implementing a database system for a new data model is a difficult and laborious task. The goal of the EXODUS project is to ease the burden of the database implementor (**DBI**). EXODUS is designed to assist the DBI in both creating a system for a new data model and in augmenting an existing system. For example, one might first use EXODUS to construct a database system for a new data model. Later, one might extend this system by adding a new access method or a new algorithm for an existing operator in the query language. To achieve this, the EXODUS design consists of a powerful, highly efficient storage system, the database implementation language E, which provides language constructs specifically designed to assist in database implementation, a type manager, which maintains state and location information about the types and procedures defined in the system, and an optimizer generator. In the future, we plan on investigating generators for user interfaces. An overview of the architecture of EXODUS can be found in [CARE86b]. The design of the storage manager and file system is presented in [CARE86a]. The E programming language is described in [RICH86]. In this paper, we describe the optimizer generator.

Until very recently, query optimizers [SELI79, WONG76, KOOI80] have been designed and implemented with a specific data model and database system in mind. The operators and their algorithms, the access methods, and the cost model were all known when the database system was being implemented. Consequently, the optimization process could also be tailored to the target data model and its implementation. Only the Postgres optimizer [STON86] allows the incorporation of new access methods into the optimization process.

Since EXODUS does not support a single conceptual data model, it would impossible to provide a single optimizer for all target applications. As a solution we hypothesized [CARE85] that if the query optimizer were organized as rule-based system, then as new operators, access methods, etc., were added to the database system, the optimizer could be informed of their properties by adding new rules to its rule base. As we began to investigate the concept of such an optimizer it became clear that the feasibility of such a designed hinged on being able to separate cleanly the data model specific parts of the optimizer from the common components. The common components consist primarily of the search mechanism and its supporting software. The pieces specific to the data model include special types (e.g. BOX), operators, the algorithms for implementing these operators, the cost functions for the algorithms and the catalog management software. Making it easy to specify these pieces is obviously critical in

making the optimizer generator successful. In the following sections we demonstrate that using a rule based approach makes specifying these components straightforward. Furthermore, our preliminary performance results demonstrate that the access plans obtained are competitive with those produced by exhaustive search techniques while taking only a fraction of the time to produce.

One way to find the optimal access plan for a query is to simply generate all possible access plans, estimate their respective processing costs, and output the least expensive one. In the System R optimizer [SELI79] this basic strategy is augmented with a pruning technique that deletes all but the cheapest of a set of equivalent subplans at each step of the optimization process. Without pruning, the optimizer would be unacceptably slow. Following the System R example, a rule-based optimizer should employ certain laws or "musts" (eg. whenever possible use a join operator rather than a Cartesian product followed by a selection) and heuristics (eg. move selections before joins) in its search strategy in order to reduce the number of access plans considered.

The remainder of this paper is organized as follows. In Section 2, we present the design of our rule based optimizer generator. We also describe the operation of an optimizer produced with the generator. The search strategy employed by a generated optimizer and how it improves itself by learning is presented in Section 3. Section 4 gives some computational results obtained with an optimizer generated for a restricted relational model. In Section 5, we compare and contrast our work with related research. Future directions are outlined in Section 6. Our conclusions can be found in Section 7.

2. Design of the Optimizer Generator

2.1. Overview

In order to be sufficiently general, an optimizer generator must be based on an abstraction of optimization suitable for most data models. We decided that queries and access plans should be expressed as trees, because we believe that operator trees are general to all set oriented data models in which complex queries are composed by nesting a finite set of procedures. The nodes of the query trees are labeled with an operator and its arguments, eg. a selection predicate. There are two alternative ways of transferring data between operators: temporary files and pipelines. Without precluding the use of either one, we simply refer to them subsequently as inputs or streams.

Before a query can be optimized, an initial operator tree must be constructed. In EXODUS, this is done by the user interface and parser. The output of the optimizer, the access plan, can either be interpreted by a recursive

procedure or it can be further transformed. Both approaches have been used successfully in existing database systems. In Gamma [DEWI86], for example, the operators in the access plan are interpreted (though the predicates themselves are compiled into machine language). In System R [SELI79], the access plan was compiled into machine language. Freytag [FREY85, FREY86] suggests applying rule-based techniques for this step.

In most database systems, there are frequently several alternative algorithms for the same logical operation. For example,[1] the relational join operator can be implemented using several alternative join methods. Our model distinguishes between *operators*, corresponding to primitives provided by the data model, and *methods*, that are specific implementations of the operators. The access plans produced by the optimizer are also trees, with a method and its argument in each node. In this model of queries and access plans, query optimization consists of query tree reordering and method selection. Since this optimization scheme is centered around the algebra of the data model, we refer to it as *algebraic optimization.*

As example, consider the query tree and a corresponding access plan shown in Figure 1. Notice that in producing the access plan on the right from the query tree on the left, two types of rules are applied to the tree. First, the operators are rearranged by pushing the selection before the join. Second, each operator is replaced by a method that implements it.

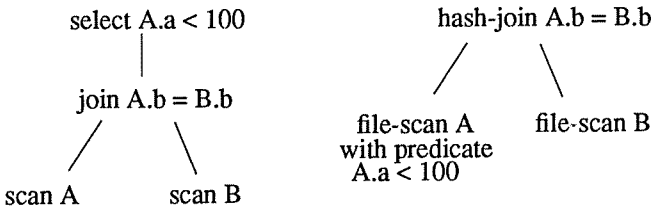

Figure 1

As proposed in [CARE85], we initially intended to implement a rule-based optimizer using an AI language like Prolog [WARR77, CLOC81], OPS5 [FORG81], or LOOPS [BOBR83] as those languages provide pattern matching and a search engine, and since unification can be used elegantly to build new query trees from old ones. In addition, these languages allow augmentation of the rule base at run-time. This capability is desirable for two

[1] A word about the examples in this paper. First, examples based on the relational data model were chosen because they are easily understood. We firmly believe that the ideas presented here apply to most other data models. Second, larger examples are ended with a □.

reasons. First, in a database system that permits the addition of new abstract data types, access methods, etc., it is necessary to inform the optimizer about those changes. Second, when the optimizer finds that certain sequences of transformations occur frequently together, the optimizer could augment the rule set by adding a single rule that combines the sequence of transformations. In successive optimizations, the whole sequence of transformations could then be done in a single step.

We implemented and experimented with a prototype in Prolog, which, unfortunately, had to be abandoned. This prototype had two serious problems. First, Prolog has a fixed search strategy, depth first search. We found that we needed to augment the search strategy dynamically while the optimizer was running; a fairly cumbersome task. Second, our implementation (C-Prolog interpreter) was slower than we were willing to accept.

Having abandoned this prototype we decided to pursue the idea of implementing a rule-based optimizer generator. While building an optimizer generator in C required more work initially, it left us with the freedom to implement exactly the desired functionality and a search strategy tuned to the process of optimizing algebraic queries. Furthermore, we were able to experiment with alternative designs in a straightforward manner. The principal disadvantage of the generator approach is that the optimizer cannot be changed while running, a feature other researchers have found useful [STON86].

The input into the EXODUS optimizer generator consists of a set of operators, a set of methods, algebraic rules for transforming the query trees, and rules describing the correspondence between operators and methods. This information is contained in the **model description file**. Figure 2 gives an overview of the use of the optimizer generator. When the database system is constructed, the generator produces a data model specific optimizer from the description. At run time, each query is transformed into an operator tree by the user interface, optimized by the generated optimizer, and then interpreted or transformed into a program.

The generated optimizer transforms the initial query tree step by step, maintaining information about all the alternatives explored so far in a data structure called MESH. MESH is also used to hold access plans for each query tree that has not been pruned from the data structure. At any time during the optimization process there can be a large set of possible next transformations. These are collected in a data structure called OPEN[2] which is maintained as a priority queue. OPEN is initialized to be the set of transformations that can be applied to the initial query tree.

[2] OPEN is a standard name for the set of possible next moves in AI search algorithms [BARR81].

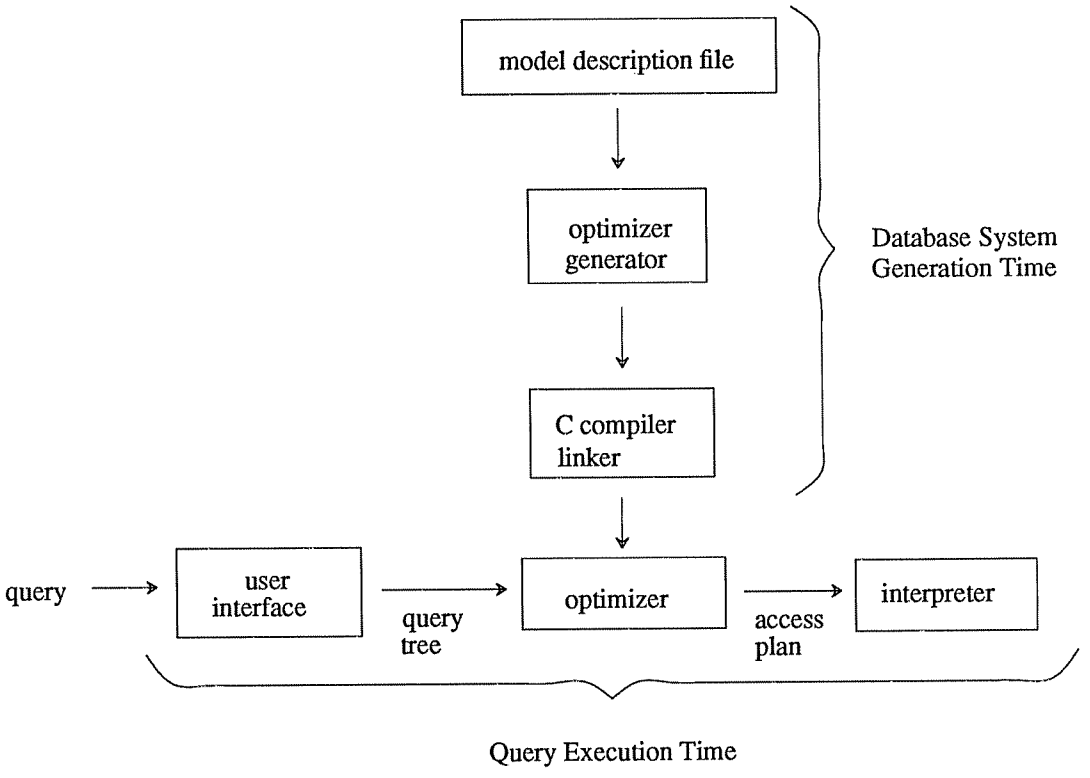

Figure 2

The general optimization algorithm can now be described as follows.

```
while (OPEN is not empty)
        Select a transformation from OPEN
        Apply it to the correct node(s) in MESH
        Do method selection and cost analysis for the new nodes
        Add newly enabled transformations to OPEN
```

The rules governing query tree transformations and method selection are specific for the data model and must be defined in the model description file.

2.2. The Input to the Optimizer Generator

To implement a query optimizer for a new data model, the DBI writes a model description file and a set of C procedures. If the new model resembles one for which an optimizer has already been generated, it might be more convenient to augment an existing model description file. The generator program transforms the description file into a C program. This is compiled and linked with the set of C procedures written by the DBI to form a data model specific optimizer.

In the model description file, the DBI lists the set of operators of the data model, the set of methods to be considered when building and comparing access plans, the rules defining legal transformations of query trees, termed **transformation rules**, and the rules defining the correspondence between operators and methods, termed **implementation rules.**

The model description file has two required parts and one optional part. The first required part is used to declare the operators and the methods of the data model. It can also include C code and C preprocessor declarations to be used in the generated code. The second part consists of transformation rules and implementation rules. The optional third part contains C code that is appended to the generated code. These parts will be discussed in further detail below. In addition, we will illustrate how the pieces fit together through a series of examples.

In the first part of the model description file, called the **declaration part**, the operators and the methods of the data model are declared. The keywords *%operator* and *%method* are followed by a number to indicate the arity and by a list of operators or methods with this arity.

Example:

```
%operator 2 join
%method 2 hash_join loops_join cartesian_product
```

In this example, an operator *join* and three methods *hash_join*, *loops_join*, and *cartesian-product* are declared. The 2's signal the generator that the join operator and the three methods each require two input streams. □

Besides operator and method declarations, the first part of the description file can also include C code that will be written into the output file for the optimizer before any generated code. This capability is used to provide data model specific definitions for four types used by the optimizer generator. These are: OPER_ARGUMENT, METH_ARGUMENT, OPER_PROPERTY, and METH_PROPERTY. These types are used in the structure definition of nodes for query trees, access plans, and MESH to store the arguments of operators and methods, eg. predicates, and "properties" that the DBI can associate with a node. In each MESH node, the proper operator arguments and method arguments are inserted by calling procedures provided by the DBI, and they are stored in memory locations of type OPER_ARGUMENT for the operator and of type METH_ARGUMENT for the method. If the DBI wishes to do so, it is possible to store information about a subtree in its root node, eg. relation cardinality, tuple width, etc. In each node in MESH, there are two fields provided for this information, *oper_property* of type OPER_PROPERTY and *meth_property* of type METH_PROPERTY. The contents of the former field depends only on the operator. while the latter depends on the method chosen for the node. For example, in our relational prototypes we store the schema of the intermediate relation in oper_property and the sort order in meth_property.

The second part of the description file, called the **rule part**, contains the transformation rules and the implementation rules. A rule consists of two expressions and an optional condition. Between the expressions is the keyword *by* for implementation rules and an arrow for transformation rules. The arrow indicates the legal directions of the transformation. The arrow can point to the left, to the right, or can be double-sided. If a one-sided arrow has an exclamation mark with it, the transformation cannot be applied to a query tree generated by this transformation. While useful for an optimizer's performance, it should never be necessary to use this feature for correctness. A typical situation where it can improve the optimizer's performance is a commutativity rule. Using commutativity twice results in the original query tree; if a query tree is generated that is exactly like one generated earlier, the duplication is detected and the new query tree is removed. Thus, not allowing commutativity to be applied twice is only a performance and not a correctness issue.

Each expression in a transformation rule and the expression on the left side of an implementation rule consists of an operator and a parameter list. Each parameter can be another expression or a number. A number indicates an input stream or a subquery. The expression on the right side of an implementation rule consists of a method and a list of inputs.

Example:

```
join (1, 2) ->! join (2, 1);
join (1, 2) by hash_join (1, 2);
```

The first line of this example is the join commutativity rule. Since applying it twice results in the original form, the once-only arrow (with exclamation mark) is used. The second line indicates that *hash_join* is a suitable implementation method for *join*. □

Sometimes the same operator name appears twice in the same expression, for example, in an associativity rule. In this case, it is necessary to identify the operators so that arguments (eg. join predicates) can be transferred correctly when the transformation is applied. For identification, operators in an expression can be followed by a number. If the same number appears with an operator on the other side of the arrow, the arguments are copied between these two operators. If the DBI wishes a default action other than simple copying, a function name *COPY_ARG* can be declared to the C preprocessor, replacing the default action. If something other than simply copying arguments from the initial query into MESH and from MESH into the final access plan is needed, the DBI can define the functions *COPY_IN* and *COPY_OUT*. If this argument passing scheme is not sufficient, a procedure name can be given with a transformation or implementation rule. Instead of using the default mechanism, this procedure is called to transfer (and possibly modify) the arguments.

Example:

project (hash_join (1, 2)) by hash_join_proj (1, 2) combine_hjp;

This rule indicates that there is a special form of hash join, called *hash_join_proj*, that can be used when a hash join is followed by a project operator. When hash-join-proj is chosen, the optimizer will call the the DBI supplied procedure *combine_hjp* to combine the projection list and join predicate to form the argument of *hash_join_proj*. □

Both transformation rules and implementation rules may have a condition associated with them. Conditions are written as C procedures and are executed after the optimizer has determined that a subquery matches the pattern of a rule (ie. that subquery has the same operators in the same positions as the rule). When the condition is not met, the special action *REJECT* is provided. If a *REJECT* action is not executed, the transformation is added to OPEN. The condition code can access the arguments and properties of the operators and the inputs of the expression via pseudo variables defined by the generator. These variables are called OPERATOR_1, OPERATOR_2, etc., and INPUT_1, INPUT_2, etc.. The numbers in these variables are the same as those used to identify operators and inputs. Each variable is actually a structure (record) and includes the fields *oper_property*, *oper_argument*, *meth_property*, and *meth_argument*. In the case of a transformation rule that can be used in both directions, the condition code is inserted twice into the optimizer code. To distinguish these cases at compile time, C preprocessor names *FORWARD* and *BACKWARD* are defined for use in the condition code.

Example:

```
join 7 (join 8 (1, 2), 3) <-> join 8 (1, join 7 (2, 3))
{{
# ifdef FORWARD
if (NOT cover_predicate (OPERATOR_7.oper_argument,
        INPUT_2.oper_property, INPUT_3.oper_property))
  REJECT;
# endif
# ifdef BACKWARD
if (NOT cover_predicate (OPERATOR_8.oper_argument,
        INPUT_1.oper_property, INPUT_2.oper_property))
  REJECT;
# endif
}}
```

This example illustrates the join associativity rule and the use of conditions to control the application of a transformation. Since the join operator appears twice in each expression, the numbers 7 and 8 are appended to distinguish the two instances of the operator. This allows the optimizer to transfer correctly the join predicates between the two operators as the transformation rule is applied. The condition code, the lines between {{ and }}, is copied twice into the optimizer code. Nevertheless, only one if statement from the condition code is executed for each direction (the other one is removed by the C preprocessor). The Boolean function *cover_predicate* is assumed to determine whether all the attributes occurring in the predicate that is the first argument to the function are attributes of the relations described by the second and third arguments. □

The rule set must have two formal properties — it must be *sound* and *complete*. Sound means that it allows only legal transformations. If the condition code is not correct, there is nothing the generator can do about it, and

the generated optimizer will not work properly. Complete means that the rule set must cover all possible cases, such that all equivalent query trees can be derived from the initial query tree using the transformation rules. If the rule set is not complete, the optimizer will not be able to find optimal access plans for all queries. On the other hand, the rule set can be redundant. In fact, if the DBI foresees that a certain combination of rules will be used frequently, it is recommended (but not required) that this combination be specified as a single rule. This will speed up the optimization process, but it will not affect its results, unless the search parameters (described in Section 3) are set too restrictively.

Besides the model description file, the DBI must provide a set of C procedures. These are the property procedures, the cost functions, and some support functions. The name for a property or cost function is the concatenation of the word *property* or *cost* and the operator or method name. The names for the support functions are fixed. For each operator, one property function is required. For each method, a property function and a cost function is required. Support functions include argument comparison, memory allocation/deallocation, and formatting procedures for property and argument fields. The memory functions are used for intermediate data structures and the access plans. The formatting procedures are used by the built-in debugging facilities including an interactive graphics program[3]. Property functions for operators allow the DBI to cache information in individual nodes of the intermediate query trees to speed up condition and argument processing. For example, in our relational prototype, the schema of each intermediate relation is cached. Property functions for methods allow the DBI to derive and cache information that depends on the selected method, eg. physical sort order. Cost functions determine the processing cost for each method, depending on the operator argument and the input streams.

This scheme of using DBI functions to complement the automatically generated optimizer has a very desirable side effect. The DBI is basically forced to write the code in a structured, modular way. The various DBI routines can be written independently, meaning that they can be written at different stages of a development project. The same is true about the transformation and implementation rules. Each rule can be specified independently of other rules. The generator builds the necessary connections and control structures. Again, incremental development

[3] Admittedly, these tools were used when debugging the optimizer generator and the code implementing the search strategy, but they also proved invaluable when debugging the DBI code for our prototype implementation. The graphics capabilities were first implemented for a demonstration, but they are very useful for quick understanding and debugging. Including the debugging tools into the optimizer is a command line switch of the generator program.

and enhancement of a database system and its optimizer component is supported. For example, imagine the DBI wants to explore how useful a newly proposed index structure is. To have the optimizer consider this new index structure for all future optimizations, all the DBI has to do is write a few implementation rules, a property function, and a cost function[4].

The generator produces the source code for the optimizer in a single pass over the description file. While reading the declaration part, it builds a symbol table of operators and methods and copies C source lines into the output file. For the rule part, it maintains three temporary files for the procedures **match**, **apply**, and **analyze**. Match takes a subquery and adds all applicable transformations to OPEN. Apply actually performs a transformation after it has been selected from OPEN. Analyze determines the cheapest possible method for the root of a subquery by matching it against the implementation rules and by calling the cost functions. For bidirectional transformation rules, the code generation procedure is invoked twice for match and apply, once for each direction. Thus, a bidirectional rules appears as two rules in the generated optimizer.

For each transformation rule, three tests are inserted into the procedure match. First, a subquery cannot be transformed by a rule if the rule is a once-only rule and the subquery has been generated by this rule, or if the rule is bidirectional and the subquery has been generated by the opposite direction. Second, a rule cannot be applied to a subquery if the patterns do not match. The patterns match if there are the same operators at the same positions in the rule and in the subquery. Third, a rule cannot be applied if there is a condition and the condition is not met.

To apply a transformation, all necessary new nodes are generated and operators, operator arguments, and inputs are filled in. For each new node, a procedure is called which either finds an existing equivalent node or invokes property caching and method selection for the node. This process is described in more detail below. For each implementation rule, code is added to the procedure analyze. If a subquery and a rule pattern match, this code calls the cost function of the appropriate method and compares the result to the least expensive implementation found so far for the subquery.

When the parser finds the end of the rule part, these procedures and a library of support routines are appended to the output file. The support routines implement the control structure and maintain the OPEN data structure. Finally, the third part of the model description file is appended to the optimizer source code.

---

[4] There remains, of course, the non-trivial problem of coding the operations on the new index structure. EXODUS eases this task with its database implementation language E [RICH86].

2.3. Operation of a Generated Optimizer

The cost model that the optimizer supports is simple but powerful. The cost for a query tree is the sum of the costs of all methods in its access plan. One might criticize this model at first as being too naive since it does not allow the incorporation of buffering effects that potentially reduce the I/O cost of intermediate files. However, if such effects exist, they can and should be incorporated into the cost functions. This is one of the reasons why all available information is passed as arguments to the cost functions that are written by the DBI.

As mentioned earlier, information about the query trees and access plans explored so far is stored in a data structure called MESH. MESH is a network of nodes that represents both alternative query trees and access plans. Since the size of each node is at least 100 bytes,[5] and since there can be many query trees to consider, it was important that MESH be designed to avoid any unnecessary redundancy. Also, since we wish to avoid redundant processing, it seems natural to share as many nodes as possible between query trees. To achieve this, the optimizer allocates nodes only when necessary during a transformation, sharing copies whenever feasible. With this implementation, typically as few as 1 to 3 new nodes are required for each transformation, independent of the size of the query tree.

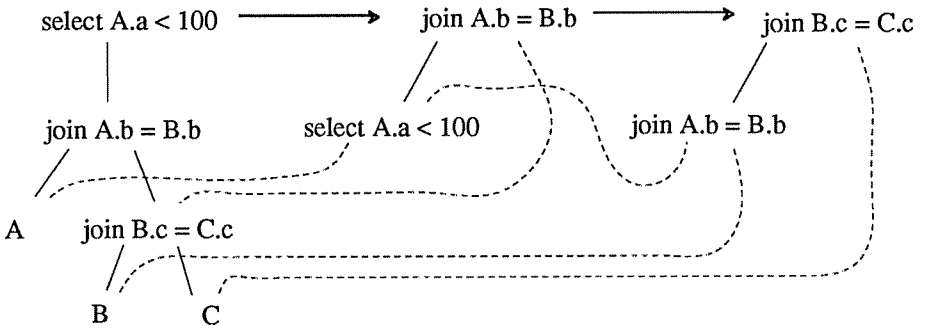

Figure 3

Example: Consider Figure 3. The bold arrows denote transformations, solid lines show the input streams (which flow upward), and dotted lines point to subtrees that are being reused. The first transformation pushes the selection down the query tree. The second transformation applies join commutativity. □

[5] This is the minimal size. The actual size depends on the size of the data structures defined by the DBI, and on the maximal arity of the operators and methods in the data model. In our current implementations, each node is almost 200 bytes long.

More precisely, a node is created for each operator that appears in the transformation rule on the "new" side. The optimizer then traverses the new nodes bottom-up and tries to replace each one by an existing equivalent node. Two nodes are equivalent if they have the same operator, the same operator argument, and the same input(s). A hashing scheme is employed to make the search for equivalent nodes extremely fast. This scheme to detect equivalent nodes is already used when the initial query tree is copied into MESH, so that common subexpressions in the query are recognized as early as possible. If a new node cannot be replaced by an existing duplicate, it is matched against the implementation rules in order to find the optimal access plan for the new subquery rooted at this node. Furthermore, it is matched against the transformation rules, and any applicable transformations are added to OPEN. Then, all parent nodes of the old subquery (those that point to the old subquery or an equivalent subquery as one of their input streams) are matched against the implementation rules to propagate the cost improvement obtained by the transformation performed. We term this **reanalyzing**. Finally, the parent nodes are matched against the transformation rules, as there might now be some (new) possibilities for further transformations. This is called **rematching**.

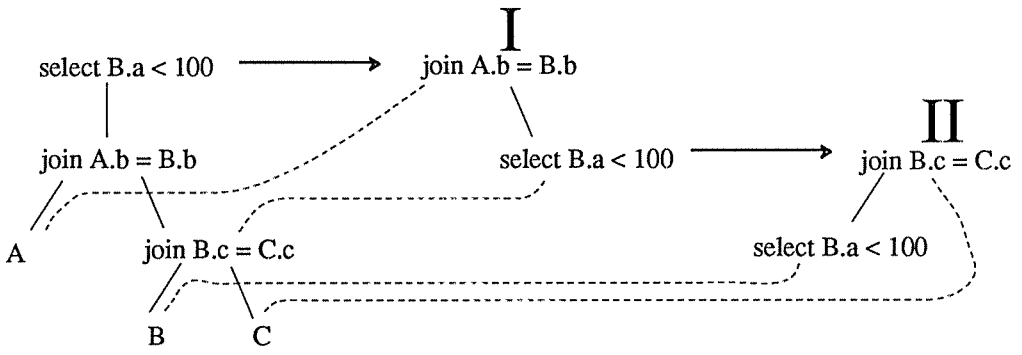

Figure 4

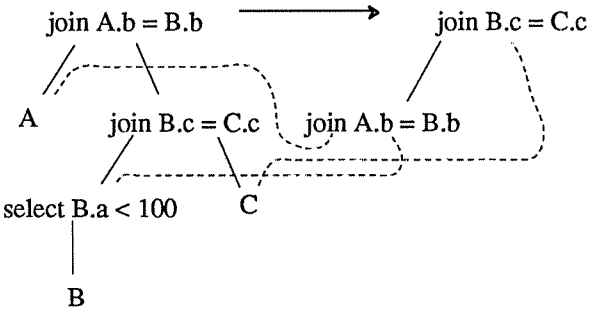

Figure 5

Example: Consider Figure 4. The first two transformations push the selection down the query tree, reusing nodes where possible. To apply join associativity, the node labeled I must be rematched with the node labeled II as its right input, resulting in an entry in OPEN that will eventually lead to the transformation shown in Figure 5. □

3. Search Strategy and Learning

Since the number of possible transformations in OPEN can be very large for a complex query, if such queries are to be optimized in a reasonable amount of time it is critical that the optimizer avoid applying most of these transformations. To find the optimal access plan quickly, the search must be directed [BARR81]. To do this, the "right" transformation must be selected from OPEN at each step of the optimization process. The ideal situation would be to select only those transformations that are necessary to transform the initial query into the query tree corresponding to the optimal access plan. Unfortunately, this is not feasible as the optimal access plan and the shortest sequence of transformations are not known. Instead, the optimizer selects the transformation which promises the largest cost improvement. **Promise** is calculated using the current cost (before the transformation) and information about the transformation rule involved. To measure the promise of a transformation rule, an **expected cost factor** is associated with each transformation rule. Bidirectional transformation rules have two expected cost factors,

one for each direction. The interpretation of this factor is as follows: if the cost before the transformation is **c** and the expected cost factor of the transformation rule is **f**, then the cost after the transformation is **c*f**. If a rule is a good heuristic, like pushing selections down in the tree, the expected cost factor for this rule should be less than 1. If, however, a rule is neutral on the average, (eg. join commutativity), its value should be 1.

The concept of expected cost factors raises two important issues. First, is such a factor valid? That is, is it possible to associate a value with a rule independent of the database and the queries to be optimized? Second, how can these factors be determined? We will address the second question first.

We decided that it is too difficult (and too error prone) to let the DBI set the expected cost factors. On the other hand, since we do not know the data model and the rules a future DBI might implement, we cannot set these cost functions either. Thus, they should be determined automatically by the optimizer by learning from its past experience. An adequate method is to use the average of the observed cost quotients for a particular rule. Recall that the expected cost factor is an estimate for the quotient of the costs before and after applying the transformation rule. Thus, it is suitable to approximate the factor with the observed quotients for the rule.

The simplest averaging method is to take the arithmetic average of all applications of the rule since the optimizer was generated. However, if the query pattern or the database changes, using the average of **all** observed quotients might be too rigid. One alternative would be the average of the last N applications (for some suitable N). This is fairly cumbersome to implement, however, as the last N values must be stored for each rule. A second alternative is to calculate a sliding average for each rule. The sliding average is the weighted average of the current value of the expected cost factor and the newly observed quotient, and is quite easy to implement efficiently. Finally, since we average over quotients, a geometric average may be more appropriate than an arithmetic average. In our tests, we evaluated the following four averaging formulae:

geometric sliding average $f \leftarrow (f^K * q)^{\frac{1}{K+1}}$	geometric mean $f \leftarrow (f^c * q)^{\frac{1}{c+1}}$
arithmetic sliding average $f \leftarrow \frac{f*K+q}{K+1}$	arithmetic mean $f \leftarrow \frac{f*c+q}{c+1}$

In these formulae, f is the expected cost factor for the rule under consideration, q is the current observed quotient of new cost over old cost, c is the count of how many times this rule has been applied so far, and K is the sliding aver-

age constant. As will be discussed below, all of these averaging formulas lead to statistically valid constructs, and the performance differences between them are fairly small.

In many cases, we will find that a beneficial rule is possible only after another (perhaps even negatively beneficial) rule has been applied. To reflect this in the search strategy, the optimizer actually adjusts the expected cost factor of *two* rules after an advantageous transformation. First, it recalculates the factor for the rule just applied using one of the techniques described above. Second, it also adjusts the factor of the preceding rule that was applied, using the same formula but with only half the weight. Thus, a rule that frequently enables subsequent beneficial transformations will have an expected cost factor lower than 1 (the neutral value), and will be preferred over other neutral rules without this indirect benefit. We call this **indirect adjustment**. Finally, if a cost advantage is realized while reanalyzing the parent nodes after a transformation, the rule's expected cost factor is also adjusted with half the normal weight. We call this **propagation adjustment**.

Ordering the transformations in OPEN by the expected cost decrease has a negative effect in some situations. If OPEN contains two equivalent subqueries with different costs each of which can be transformed by the same rule with an expected cost factor less than 1, the transformation of the more expensive query tree will be selected first. This is, of course, counterintuitive, and not a good search strategy. To offset this effect, the optimizer subtracts a constant from the expected cost factor when estimating the cost after a transformation of a part of the currently best access plan. The lowered expected cost factor increases the expected cost improvement, such that the currently best subquery is transformed before the other equivalent subquery.

The expected cost factors are used to direct the search, so the optimizer finds the "optimal" access plan quickly. Once the optimal access plan has been found, the optimizer could ignore all the remaining transformations in OPEN, and output the plan. Unfortunately, it is impossible to know when the currently best plan is indeed the optimal one. Our solution is to let the optimizer keep searching, but to limit the set of new transformations that are applied. To do this, the cost improvement expected by applying a transformation is compared with the cost of the best equivalent subquery found so far. If this improvement is within a certain multiple of the current best cost, the transformation is applied; otherwise, it is ignored and removed from OPEN. Using the analogy of finding the lowest point in a terrain, but sometimes having to go uphill to reach an even lower valley, this technique is termed **hill climbing**. The multiple mentioned above is the **hill climbing factor**. Typical values are 1.01 to 1.5. If it is less than 1, neutral rules will never be applied, even though they might be necessary to explore the complete search space. On the other hand, the experiments described later show that for the relational model hill climbing factors

close to 1 work well.

Finally, there is a **reanalyzing factor**. Recall the importance of reanalyzing from Figures 4 and 5. If the cost of the newly generated subquery is significantly higher than its best equivalent subquery, reanalyzing is probably wasted effort. Only if the cost of a newly generated subquery is within a multiple of its best equivalent subquery are all the parent nodes (ie. those containing the old subquery as one of their inputs) matched against the transformation and implementation rules with the old subquery replaced by the new one.

Unfortunately, the appropriate values for the hill climbing and reanalyzing factors seem likely to depend on the data model. Thus, like the expected cost factors, they too should be learned by the optimizer. We have not, however, implemented this feature yet.

4. Computational Results from a Relational Prototype

In this section, we report some preliminary results obtained with an optimizer generated for a subset of the relational model. This model is restricted to select and join operators. We implemented this model first because producing the optimal join tree is reportedly the major problem in relational query optimization [SELI79, WONG76, KOOI80]. For the leaves of the query trees, we introduced an artificial operator, called *get*. *Get* reads a file from disk and transfers it to the next operator. It was introduced for convenience as it allows us to write the cost functions for the other operators' methods without regard to whether their input streams come from disk or from other operators. It also makes it easy to express the fact that the input for methods based on indices must be a stored relation.

The test queries for our experiments were generated randomly as follows: to generate a query tree, the top operator is selected. A priori probabilities are assigned to join, select, and get; in our test 0.4, 0.4, and 0.2 respectively. If a join or select is chosen, the input query trees are built recursively using the same procedure. If a predefined limit of join operators (here: 6) in a given query is reached, no further join operators are generated in this query. The join argument is an equality constraint between two randomly picked attributes of the inputs. The selection argument is a comparison of an attribute and a constant, with the attribute, comparison operator, and constant picked at random. The database consists of 8 relations with 1000 tuples each. Each relation has 2 to 4 attributes. The schema is cached in main memory during the optimizer test run. The schema of each intermediate relation is cached in the query tree node in MESH as an operator property. The only method property considered in our system is sort order.

Our transformation rules included join commutativity and associativity, commutativity of cascaded selects, and the select-join rule. This last rule allows pushing selects down the query tree, but only on the left branch. If the selection clause must be applied to the right branch, join commutativity must be applied first. We used only the left-branch form of the select-join rule because it forces the optimizer to perform rematching and indirect adjustment. The rule also allows the optimizer to push joins down in the tree, since it is a bidirectional rule. For joins, we considered four methods: *nested loops*, *merge join*, *hash join*, and *index join*. A merge join requires the inputs to be sorted on the respective join attribute. An index join requires that the right input be a permanent relation with an index on the join attribute. Selection is done either with a *filter*, which is a method with one input stream and one output stream, or with a scan. We considered file scans and index scans. A scan can implement any conjunctive clause, ie. a cascade of selects with a get operator at the bottom. The cost calculation estimates elapsed seconds on a 1 MIPS computer with data passed between operators as buffer addresses. When specifying the algebra description, we realized several shortcomings of the generator. Some of them have since been corrected, and others are described in the section on future work.

The first tests were used to ensure that the generated optimizer transforms the query correctly and produces the optimal or a near-optimal query plan. One way to test this is to duplicate an existing optimizer and to compare the query plans produced. However, this would have required imitating all of its cost functions, which is not easily accessible information. More importantly, it would have restricted us to its particular set of operators and methods, leaving little room for modification and experimentation. Thus, we decided to compare our optimization results with those of an exhaustive search of all possible access plans. We modified the optimizer to do undirected exhaustive search. To avoid thrashing on the time-shared computer used for these experiments, however, we aborted optimization of a query when MESH contained 5,000 nodes. That implied that OPEN contained about 5,000 to 10,000 elements, and that the heap area had grown to about 3 megabytes.

The following tables summarize typical results for a sequence of 500 randomly generated queries. The queries in this sequence contain 805 join operators and 962 select operators. The reanalyzing factor is set equal to the hill climbing factor. We report the results for three values for the hill climbing and reanalyzing factors to demonstrate the effects of search effort on the quality of the resulting access plans. A hill climbing factor of ∞ indicates undirected exhaustive search. This allows the comparison of the restricted search strategies with unrestricted search. All remaining runs used directed limited search. The second column, labeled 'total nodes generated', indicates the amount of main memory used for MESH. The average size of MESH is 1/500 of the given numbers. The

third column is the sum of the MESH sizes at the times when the best access plans were found.[6] The fourth column shows the sum of the estimated execution costs of the 500 generated access plans. The last column states the CPU time (in seconds) spent optimizing the entire sequence of 500 queries[7].

Hill Climbing	Total Nodes Generated	Nodes before Best Plan	Sum of Estimated Execution Costs	CPU Time
1.01	64022	21776	46434	131.0
1.03	115903	27564	46257	305.9
1.05	144658	38913	46009	346.5
∞	890433	166679	55571	5546.1

Table 1. Summary of 500 queries.

With increasing search effort (ie. larger hill climbing and reanalyzing factors) the CPU time increases as the cost of the access plans decreases. Notice that the sum of costs for "exhaustive" search is actually higher than for restricted search. This is due to the fact that optimizations had to be aborted because the memory requirement for exhaustive search turned out to be excessively high, ie. the exhaustive search could sometimes not be completed so only a suboptimal plan was produced. It is interesting to restrict attention to those queries that were not aborted in the undirected exhaustive search. When restricted to the 338 queries for which the exhaustive search succeeded, Table 1 becomes the following.

Hill Climbing	Total Nodes Generated	Nodes before Best Plan	Sum of Estimated Execution Costs	CPU Time
1.01	4309	1813	9837	5.0
1.03	4771	1958	9837	5.8
1.05	5277	2002	9833	6.2
∞	80380	7754	9637	87.0

Table 2. Summary of 338 queries not aborted in exhaustive search.

When comparing table 1 and table 2, the reader will immediately notice the substantial differences in resource consumption, both for CPU time and memory. Nevertheless, for more than 310 of the 338 queries the different search strategies produce access plans with exactly the same cost as the optimal plan. The following table gives a more

[6] This is done by associating with the currently best plan (of which there is only one) the number of nodes in MESH at the time the plan was generated.

[7] The times are given in seconds in user mode on a Gould 9080 running UTX/32, version 1.3. The times were measured using the *getrusage* system call. This machine has two CPU's rated at about 5 MIPS each. The optimizer usually ran uninterruptedly on the second CPU.

detailed picture of the cost differences.

Cost Difference Relative to Exhaustive Search	Number of Queries Hill Climbing Factor		
	1.01	1.03	1.05
no difference	314	315	315
more than 0%	24	23	23
more than 5%	20	20	19
more than 10%	20	20	19
more than 25%	9	9	9
more than 50%	1	1	1

Table 3. Frequencies of differences in 338 queries.

For only 20 out of the 338 queries does the cost of the access plans differ by more than 5%. The worst case is a query with exactly double the cost. These results indicate that undirected exhaustive search is inferior to the search strategy presented in this paper, and that the search strategy employed by our rule based optimizer generally does quite well.

As described earlier, we associate an expected cost factor with each rule to direct the search into the most promising direction. We considered it necessary to test whether the expected cost factor is a valid construct. If there really is such a factor for each rule, it should be the same independent of the queries being optimized. To test this hypothesis, 50 sequences of 100 queries each were optimized in independent runs of the optimizer, and the expected cost factors for each rule at the end of the run were compared. For each of these sequences, we selected a different combination for the select, join, and get probabilities used to generate the random queries, and a different limit was set on the number of joins allowed in a single query. While the expected cost factors show some variance, they fall around the mean for each rule in a normal distribution. Our statistical testing indicated that, for our sets of test queries, the equality hypothesis is true with a 99% confidence.

Next we attempted to determine which of the four averaging methods is best suited for use in the optimizer. The results, however, were not conclusive. All four averaging techniques worked equally well with the query sequences tested. This is not discouraging, however. It only means that the differences among the adjustment formulae are insignificant. The differences between directed search and undirected search remain.

Since reordering join trees is considered the major problem in relational query optimization, we designed an experiment which specifically addresses this issue. We created several batches of 100 queries each. The queries in the first batch have one join operator each, two in the second, etc., up to 6 joins per query. The optimization

results are given in the table below. The hill climbing and reanalyzing factor was set to 1.005. Optimization was aborted when the number of nodes in MESH reached 10,000, or when MESH and OPEN together contained 20,000 entries.

Joins per Query	Total Nodes Generated	Nodes before Best Plan	Queries Aborted	CPU Time
1	500	100	0	3.28
2	1411	634	0	4.47
3	5489	1880	0	9.53
4	14182	4313	0	24.37
5	44434	9741	1	100.08
6	183077	44917	11	629.27

Table 4. Optimization of series of 100 queries each.

When N relations are joined in a query, the number of possible join trees is of the order of 8^N. The fact that neither the number of nodes nor the CPU time grow as rapidly demonstrates the effectiveness of sharing nodes between queries and plans. The most important result of this experiment is that the optimizer is able to handle fairly complex queries. It becomes obvious, however, that the search strategy could be enhanced significantly if semantic information were incorporated when directing the search. Such information can be build into the condition code, ie. those transformations which are technically correct are prevented if it is likely that they will not lead to the optimal query tree and access plan.

The above optimizations considered all possible trees. Many optimizers, eg. those of System R [SELI79] and Gamma [DEWI86], consider only left-deep join trees. In a left-deep join tree, the right inputs of all join nodes are scans on base relations. A tree which is not left-deep is called it a bushy tree. If only left-deep trees are considered, it is possible that the optimal access plan for some queries will be missed [ROSE86]. On the other hand, in many systems the restriction to left-deep trees is justified because scheduling operators becomes easier, spooling temporary files to disk can be avoided, and it is possible to guarantee that operators of one query do not compete for scarce resources, eg. buffer space. Optimization becomes easier, too, because there are significantly fewer join trees for a given query when only left-deep trees are considered as the number of possible left-deep join trees grows with the order of 2^N [SELI79]. In Table 5, we summarize how the optimizer performed on the queries used for Table 4 when only left-deep join trees are considered.

Joins per Query	Total Nodes Generated	Nodes before Best Plan	Queries Aborted	CPU Time
1	500	100	0	3.68
2	956	553	0	4.43
3	1569	1148	0	5.85
4	2382	1912	0	8.42
5	3699	3220	0	13.30
6	5228	4631	0	21.93

Table 5. Left-deep optimization of series of 100 queries each.

When small queries (1 or 2 joins) are optimized, approximately the same number of nodes in MESH and the same CPU time is used for bushy and left-deep trees. For larger queries, the differences are up to several orders of magnitude, reflecting the different growth rates for the number of possible join trees. The anticipated cost of the generated access plans, however, is larger if only left-deep trees are considered. The main reason is that the cost model used is based on the assumption that all intermediate results can be pipelined between operators without being written to disk.

These differences have inspired two directions for further research. One is to incorporate spooling costs into the cost model for bushy trees, and determine whether database systems like System R and Gamma should incorporate bushy trees. This issue is interesting in its own right, independent from the issues concerning the optimizer generator. The other idea we intent to examine is to break the optimization into several phases, ie. to use the result of the fast left-deep-only optimization as a starting point for optimization including bushy join trees.

5. Related Work

Many of the techniques employeed by the optimizer generator are based on a variety of earlier efforts in the query optimization area. Pioneering work was done in the System R project [ASTR76, SELI79], in the Ingres project [STON76, WONG76, YOUS79] and by Smith and Chang [SMIT75]. Optimization using algebraic identities was first used in compilers for programming languages, but seems to have only been used once for database optimization, in the MICROBE relational distributed database system [NGUY82]. Freytag assumes in his work on code generation [FREY85, FREY86a] for access plans that query plans for set-oriented data models can be expressed as trees. Recently, Freytag has begun work on designing a rule-based optimization scheme for the relational model [FREY86b]. Search strategies have been used in the areas of deduction and theorem proving, and learning has been used to improve a programs performance, eg. in game playing programs [BARR81].

Most of the query optimization research done to date, as surveyed by Jarke and Koch [JARK84], deals with

relational systems and their extensions. For the designers of previous query optimization programs, the data model has been a given fact. For example, when reordering join trees, [SELI79] and [KOOI80] assume that the order in which joins are executed makes no semantic difference. In the EXODUS optimizer generator, on the other hand, the operators and their semantics are left open, thus allowing the DBI to design and experiment with new data models.

Algebraic transformation laws have also been used in the design and implementation of the optimizer for the distributed relational database system MICROBE [NGUY82]. The goal of the MICROBE rule based optimization step was to minimize the number of operators and the amount of data to be shipped between operators. A set of transformation rules was formulated and proven to guarantee a deterministic result, independent of the actual sequence of transformations. The MICROBE optimizer takes at most o(N log N) steps, where N is the number of operators in the query. Their transformation rules were hand-coded in Pascal, the implementation language of the project.

Our approach differs from the MICROBE approach in three important ways. First, we do not assume a certain fixed data model. Second, we only assume soundness and completeness of the rule set, requiring no further properties. Proving deterministic results for a set of rules is significantly harder, perhaps not be possible for all data models and algebras, and would be asking too much from the DBI. Third, the procedures that transform the query are *generated* in our approach, allowing the DBI to concentrate on their correctness. The approaches are similar in that they both try to use formal properties of the algebra and to do query optimization "along" the theory of the data model.

From an AI standpoint, our search program is a dedicated search algorithm with some adaptive learning capabilities. We would have liked to use a promise function and a search strategy with stronger theoretical properties. Since the optimizer generator is not aware of the target data model, we were unable to use search algorithms like A^* [HART68] which would have guaranteed the optimal access plan for all queries. Even for the special case of the relational model, we were not able to find a way to calculate the promise of a transformation such that we can guarantee the properties needed for A^* and still direct the search in a reasonably effective manner.

6. Future Work

One interesting design issue that remains is to provide general support for predicates as some form of predicates are likely to be appear in all data models. Writing the DBI code for predicates, and operator arguments in general, was the hardest part of developing our optimizer prototypes. The current design is that the DBI must design his or her own data structures, and provide all the operations on them for both rule conditions and argument transfer functions. It may be difficult to invent an all-around satisfying definition and support for predicates, but it would be a significant improvement to the optimizer generator. The fact that predicates are a special case of arguments poses an additional challenge, since the over all design of the argument data structure must still remain with the DBI.

The hill climbing and the reanalyzing factors have a significant effect on the amount of CPU time spent optimizing a query. These values are almost surely model and algebra dependent. Thus, they must either be set by the DBI or must be determined automatically. We feel that the former alternative requires a level of sophistication or time for experimentation that cannot be expected from the DBI. In order to provide the DBI (or DBA) with some control over the optimization process, we intend to leave some control over the tradeoff between the quality of resulting access plan and the cost of optimization.

Our experiments indicate that, independent from the hill climbing factor, the reanalyzing factor, and the averaging method, more than half of the nodes are typically generated after the best plan has been found. An additional stopping criterion might help to avoid a large part of this wasted effort after the best plan has been found. In commercial INGRES, a comparison between the optimization time and the expected query execution time is introduced. If the optimization has consumed a certain fraction of the time estimated for executing the best plan found so far, further optimization is abandoned and this plan is executed. We intend to explore two other criteria besides this one. The first involves the gradient of the last improvements. Imagine a graph with the time spent on optimization on the horizontal axis, and the estimated execution time of the currently best plan on the vertical axis. This curve certainly flattens out during the optimization process. Instead of going all the way to its end, it might be possible to stop when it has been flat for some length of time. Another termination condition we plan on evaluating is the number of nodes generated for a single query before optimization is preempted. In our experiments so far, we set a fixed limit for all queries. We intend to calculate a reasonable limit for each query individually. This limit will probably have to be exponential in the number of operators in the query.

We also plan on making several changes in the generated optimizers. The first is to recognize common

subexpressions when the final access plan is extracted from MESH. Common subexpressions are detected in MESH and optimized only once, but the procedure which extracts the access plan from MESH does not exploit this feature. Furthermore, the cost of common subexpressions is not spread over the various occurences. When common subexpressions are satisfactorily supported, optimization of multiple queries in a single optimizer run will be easy to implement. The other future change is to implement nested method expressions to allow the definition of method classes, with one operator, eg. *exact-match index look-up*, being used in all implementation rules requiring index look-up, eg. *index join*, *index selection*, etc.. This would be useful when adding a new access method to a system. In the current design, an implementation rule has to be added once to the model description file for each rule where the new access method can be used. Instead, by using a method class, the new access method only has to be added once, to the class.

We intend on exploring the idea of improving the search strategy through the introduction of phases into the search process. In the first phase, only proven heuristics would be used (ie. rules with very low expected cost factors) with a very limited amount of hill climbing and reanalyzing. When this search has ended, the query tree has hopefully improved significantly, and the currently best cost now establishes an upper bound for the second phase. This phase is a broader search, basically what was described as *the* search here, but starting with the result of the first phase instead of the initial query tree. Finally, the third phase would do work analogous to peep hole optimization in compiler technology, eg. predicate clause reordering [HANA77]. Other assignments of tasks to phases could be designed as well. The idea of phases is quite similar to (actually a generalization of) the idea of a "pilot pass" [ROSE86].

The first real test for the optimizer generator will come when it is used for a real system. The EXODUS project team intends to implement a relational database system. The first real system will be relational because relational technology is sufficiently known and systems exist for performance comparison purposes. With other data models, we would work on and experiment with EXODUS and the model simultaneously, which is probably not a good idea. We will then be able to assess more realistically whether the general design is useful, and where its most significant shortcomings are. The second real test will be when we set out to design an optimizer for one of the recently proposed new data models, eg. ABE [KLUG82], Daplex [SHIP81], Probe [DAYA85, MANO86], or LDL [TSUR86].

Finally, we realize that the optimizer generator works largely on the syntactic level of the algebra. The semantics of the data model are left to the DBI's code. This has the advantage of allowing the DBI maximal

freedom with the kind of data model to implement, but it has the disadvantage of leaving a significant amount of coding to the DBI. We therefore would like to incorporate some semantic knowledge of the data model into the description file. However, this is a long term goal which we have not yet given much attention.

7. Conclusions

The most important result demonstrated by this work on rule-based optimizer generators is that it is possible to separate the search strategy of an optimizer from the data model. Thus, it is possible to implement a generic optimizer and search algorithm that is suitable for many data models. The model of optimization chosen, algebraic optimization, is expected to fit most modern (set-oriented) data models.

The architecture of the EXODUS optimizer generator enforces a modular, extensible design of the DBI's query optimizer code. The transformation and implementation rules are independent from one another, and the property and cost functions are well defined, limited programming tasks for the DBI. As a consequence, incremental design and evaluation of a new data model's optimizer is encouraged. While most of the generator's inputs are fairly easy to design and to code, some pieces can be tricky. For example, depending on the design of the arguments, writing rule conditions and argument transfer functions can be fairly burdensome. More work is needed to achieve adequate support for the DBI in this area.

Our preliminary performance evaluation of an optimizer generated for a subset of the relational data model, demonstrates that it is not necessary to use exhaustive search in the query optimization process. While our experiments cover only one data model, we believe that this generalization is justified. Also, the DBI does not have to tune the search strategy. Instead, a good part of the tuning can be done automatically by the system. In terms of both optimization speed and quality of access plans produced, a generated optimizer appears competitive with a hand-coded optimizer. With the exception of a few cases, we found that the access plans found by our prototype for the relational model were as good as those produced by exhaustive search. We are currently designing a set of queries to compare systematically a generated optimizer for the complete relational model with an existing commercial relational query optimizer.

Acknowledgements

The authors appreciate the encouragement and the helpful suggestions by the other EXODUS project members: Michael Carey, Daniel Frank, Joel Richardson, Eugene Shekita, and M. Muralikrishna.

8. References

[ASTR76] M.M. Astrahan, et. al., "System R: Relational Approach to Database Management," ACM Transactions on Database Systems, Vol. 1(2), pp. 97-137, (June 1976).

[BARR81] A. Barr and E.A. Feigenbaum, **The Handbook of Artificial Intelligence,** William Kaufman, Inc., Los Altos, CA. (1981).

[BOBR83] D.G. Bobrow and M. Stefik, "The LOOPS Manual," in LOOPS Release Notes, XEROX, Palo Alto, CA. (1983).

[CARE85] M.J. Carey and D.J. DeWitt, "Extensible Database Systems," Proceedings of the Islamorada Workshop, (Feb. 1985).

[CARE86a] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita, "Object and File Management in the EXODUS Extensible Database System," Proceedings of 1986 VLDB Conference, pp. 91-100 (Aug. 1986).

[CARE86b] M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, J.E. Richardson, E.J. Shekita, and M. Muralikrishna, "The Architecture of the EXODUS Extensible DBMS: A Preliminary Report," Proceedings of the International Workshop on Object-Oriented Database Systems, (Sep. 1986).

[CLOC81] W. Clocksin and C. Mellish, Programming in Prolog, Springer-Verlag, New York (1981).

[COPE84] G. Copeland and D. Maier, "Making Smalltalk a Database System," Proceedings of ACM SIGMOD Conference, pp. 316-325, (June 1984).

[DAYA85] U. Dayal and J.M. Smith, "PROBE: A Knowledge-Oriented Database Management System," Proceedings of the Islamorada Workshop, (Feb. 1985).

[DEWI86] D.J. DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, and M. Muralikrishna, "GAMMA - A High Performance Dataflow Database Machine," Proceedings of 1986 VLDB Conference, pp. 228-237, (Aug. 1986).

[FORG81] C.L. Forgy, "OPS5 Reference Manual," Computer Science Technical Report 135, Carnegie-Mellon University, (1981).

[FREY85] C.F. Freytag, "Translating Relational Queries into Iterative Programs," Ph.D. Thesis, Harvard University, (Sep. 1985).

[FREY86a] C.F. Freytag and N. Goodman, "Translating Relational Queries into Iterative Programs Using a Program Transformation Approach," Proceedings of ACM SIGMOD Conference, (June 1986).

[FREY86b] C.F. Freytag, "A Rule-Based View of Query Optimization", submitted for publication, (Oct. 1986).

[HANA77] M.Z. Hanani, "An Optimal Evaluation of Boolean Expressions in an Online Query System," Communications of the ACM, Vol. 20(5) pp. 344-347, (May 1977).

[HART68] P.E. Hart, N.J. Nilsson, and B. Raphael, "A Formal Basis for Heuristic Determination of Minimum Path Cost," IEEE Transactions on SSC, Vol. 4, pp. 100-107 (1968).

[JARK84] M. Jarke and J. Koch, "Query Optimization in Database Systems," ACM Computing Surveys, Vol. 16(2) pp. 111-152, (June 1984).

[KLUG82] A. Klug, "Access Paths in the ABE Statistical Query Facility," Proceedings of ACM 1982 SIGMOD Conference, pp. 161-173, (June 1982).

[KLUG82a] A. Klug, "Equivalence of Relational Algebra and Relational Calculus Query Languages Having

[KOOI80] R.P. Kooi, "The Optimization of Queries in Relational Databases," Ph.D. Thesis, Case Western Reserve University, (Sept. 1980).

[LYNG86], P. Lyngback and W. Kent, "A Data Modeling Methodology for the Design and Implementation of Information Systems," Proceedings of the International Workshop on Object-Oriented Database Systems, (Sep. 1986).

[MANO86] F. Manola and U. Dayal, "PDM: An Object-Oriented Data Model," Proceedings of the International Workshop on Object-Oriented Database Systems, (Sep. 1986).

[NGUY82] G.T. Nguyen, L. Ferrat, and H. Galy, "A High-Level User Interface for a Local Network Database System," Proceedings of IEEE Infocom, pp. 96-105, (1982).

[RICH86] J.E. Richardson and M.J. Carey, "Programming Constructs for Database System Implementation in EXODUS," submitted for publication.

[ROSE86] A. Rosenthal, U. Dayal, and D. Reiner, "Fast Query Optimization over a Large Strategy Space: The Pilot Pass Approach," unpublished manuscript.

[SELI79] P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," Proceedings of 1979 ACM SIGMOD Conference, (June 1979).

[SHIP81] D.W. Shipman, "The Functional Data Model and the Data Language DAPLEX," ACM Transactions on Database Systems, Vol. 6(1), pp. 140-173, (Mar 1981).

[SMIT75] J.M. Smith and P.Y.T. Chang, "Optimizing the Performance of a Relational Algebra Database Interface," Communications of the ACM, Vol. 18(10), pp. 568-579, (1975).

[STON76] M. Stonebraker, E. Wong, P. Kreps, and G.D. Held, "The Design and Implementation of INGRES," ACM Transactions on Database Systems, Vol. 1(3), pp. 189-222, (Sept. 1976).

[STON86] M. Stonebraker and L.A. Rowe, "The Design of POSTGRES," Proceedings of 1986 SIGMOD Conference, pp. 340-355, (May 1986).

[TSUR86] S. Tsur and C. Zaniolo, "LDL: A Logic-Based Data-Language," MCC Technical Report, (DB-026-86)MCC, (Feb. 1986).

[WARR77] D.H.D. Warren, L.M. Pereira, and F. Pereira, "PROLOG - The language and its implementation compared with Lisp," Proceedings of ACM SIGART-SIGPLAN Symp. on AI and Programming Languages, (1977).

[WONG76] E. Wong and K. Youssefi, "Decomposition - A Strategy for Query Processing," ACM Transactions on Database Systems, Vol. 1(3), pp. 223-241, (Sept. 1976).

[YOUS79] K. Youssefi and E. Wong, "Query processing in a relational database management system," Proceedings of 1979 VLDB Conference, pp. 409-417, (Oct. 1979).

[ZANI83] C. Zaniolo, "The Database Language GEM," Proceedings of 1983 ACM SIGMOD Conference, (May 1983).