EROC: A Toolkit for Building NEATO Query Optimizers

William J. McKenna
Bell Laboratories
bmckenna@research.bell-labs.com

Louis Burger
NCR Corporation
louisb@sandiegoca.ncr.com

Chi Hoang
NCR Corporation
ckh@elsegundoca.ncr.com

Melissa Truong
NCR Corporation
melissa.truong@sandiegoca.ncr.com

Abstract

EROC (Extensible, Reusable Optimization Components) is a toolkit for building query optimizers. EROC's components are C++ classes based on abstractions we have identified as central to query optimization, not only in relational DBMSs, but in extended relational and object-oriented DBMSs as well. EROC's use of C++ classes clarifies the mapping from application domain (optimization) abstractions to solution domain (EROC) abstractions, and these classes provide: (1) complex predicate definition and manipulation; (2) representations for common operators, such as join and groupby, and associated property derivation functions, including key derivation; (3) management of catalog and type information; (4) implementations of common algebraic equivalence rules, and (5) System R- and Volcano-style search strategies. The classes are designed to provide optimizer implementors reusability and extensibility through layering and inheritance. EROC provides much more functionality than previous optimization tools because all of EROC's optimization classes are extensible and reusable, not just the search components.

In addition to describing EROC's architecture and software engineering, we also show how EROC's classes were extended to build NEATO (New EROC-based Advanced Teradata Optimizer), a join optimizer for a massively parallel environment. Based on the extensions required we give an indication of the savings EROC provided us. To show NEATO's efficiency and effectiveness, we present results of optimizing complex TPC/D benchmark queries and show that NEATO easily searches the entire space of query execution plans. We outline plans for extensions to NEATO and overview how the flexibility of EROC will enable these extensions.

**Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996**

1 Introduction

Optimizer development is generally agreed to be a time-consuming and complex process, and once developed, optimizers tend to be difficult to understand and extend with new features. While reflecting on the Open OODB optimizer development [BMG93], we observed that although use of the Volcano Optimizer Generator [McK93, GM93] spared the implementor the relatively complex task of writing a search engine, a great deal of code that was not specific to the OpenOODB system had to be developed. In particular, the code to represent and manipulate operator arguments (e.g., predicates), and to perform various support (e.g., catalog) functions, comprised approximately 60% of the code written by the optimizer implementor [MGB93]. We felt strongly that carefully defined and implemented abstractions (besides

the search engine) should be reusable, and eliminate reimplementation of common code.

Given these observations about optimizer development, along with the facts that we wanted to develop an optimizer for Ode[AG89] and also needed an implementation platform for many of the optimization techniques developed at Bell Labs (e.g., predicate movearound [LMS94], theta semijoins [SHL+96]), we decided to develop a core collection of reusable software components, EROC, to meet these needs. The following list gives examples of some important functionality EROC provides to assist the optimizer implementor:

- a general, yet efficient, representation of *expressions*, which is used as the underlying representation of logical and physical algebra trees (queries and query execution plans), arithmetic and aggregate expressions (e.g., 'Avg(Dept.budget * 2)'), and predicates (e.g., 'Sum(Price*(1-Discount)) > 10,000'),
- an efficient representation of a set of expressions (predicates, queries, etc.), called an *expression space*,
- representations for several relational and SQL operators (e.g., groupby) and their associated property derivation functions (including key derivations),
- a general purpose rule-based *expression enumerator* that can be used to generate an optimizer search space, as well as perform predicate transformations (because of the common underlying representation of predicates and queries),
- an implementation of the Volcano costing algorithm, an efficient yet general optimization algorithm, and
- an implementation of a Starburst-based enumeration algorithm [OL88], an efficient algorithm for fast enumeration of join orders.

EROC supports extensibility and reuse of all optimizer components, not just the search engine, so unlike the implementation described in [LVZZ94], we intend for others to be able to extend EROC's classes to customize them for a particular data model and execution environment. Another way of viewing EROC is as a far more complete population of the optimization framework described in [McK93, GM93] and [GD87, Gra87].

To demonstrate the extensibility of EROC, we show how various EROC components were extended and combined to produce NEATO, a join optimizer customized for the Teradata [ATT95] parallel environment. By comparing (1) the number of lines of code required to extend EROC classes to develop NEATO and (2) the number in EROC, we give an indication of the effort required to produce a custom optimizer using EROC and our savings in development time. We also describe how EROC's search components were composed to form its highly efficient hybrid search engine. To our knowledge, this is the first search engine to combine a Starburst-style ('bottomup') join enumerator and Volcano's goal-driven ('topdown') costing algorithm, two techniques which have been viewed as mutually exclusive. We show that for complex TPC/D [Raa95] join queries NEATO gives excellent performance, allowing us to exhaustively search the entire space of query execution plans in seconds.

The remainder of this document is organized as follows. Section 2 gives an overview of EROC and its central components. Section 3 describes how (and why) we extended EROC's base classes and combined various classes to get NEATO, and presents experimental results obtained by optimizing a portion of the TPC/D benchmark using NEATO. Section 4 covers related work and then, in Section 5, we present a summary, conclusions, and future directions.

2 EROC Components

This section outlines the architecture of EROC by describing the abstractions on which it is based and their implementations as reusable toolkit components. We show how the basic components can be layered and combined to form other components, including optimizers themselves.

The classes we describe are only those that are related specifically to optimization, called *optimization classes*. There is another group of classes, called *support classes*, that includes implementations of lists, hash tables, stacks, queues, directed graphs, dynamic arrays, and a fast memory-allocator. These classes were used extensively in building the optimization classes. Space does not permit us to describe each class, or indeed to even completely describe the classes we present, but rather our goals are (1) familiarize the reader with the important components, and with the design principles which guided their development, namely *identification of key abstractions* and the *reuse of components through layering and inheritance*, (2) give sufficient description of the search components so we can later show how to compose search strategies.

2.1 Expression Representations

Expressions are a basic building block for many of the optimization classes. To understand the EROC's search components, we first present the classes that implement expressions, and then those that support efficient storage of groups of related expressions.

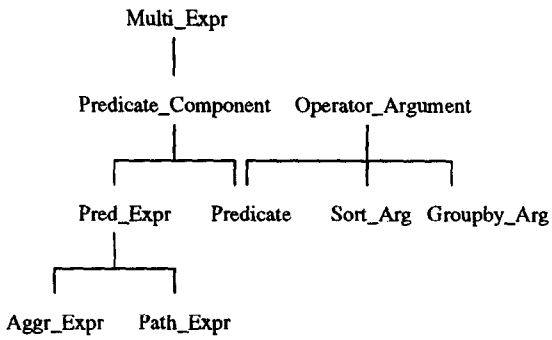

Figure 1: Partial Expression Hierarchy

2.1.1 The Multi_Expr Class

In EROC, expressions include

- queries, or logical algebra expressions ('join R.a=S.b (R S)'),
- query execution plans, or physical algebra expressions ('index_nested_loops R.a=S.b (R S)'),
- path expressions ('City.mayor.name', 'E.salary'),
- arithmetic expressions ('R.a*7'),
- aggregate expressions ('Avg(E.salary*2)'), and
- predicates ('Sum(Price*(1-Discount))>10,000').

The implementations of all expressions utilize the Multi_Expr ('multiple expression') class. EROC's implementation of expressions was inspired by the representation of *terms* in *term rewriting systems*, where terms can be represented as directed graphs whose nodes are variables and constants [Pv93]. EROC's Multi_Expr class is therefore layered on top of the Directed_Graph class. Figure 1 shows that portion of EROC's expression and operator argument hierarchy relevant to the expressions listed above[1]. (Operator arguments are described below.)

The reuse achieved through inheritance and layering gets us additional 'free' functionality for expressions. Having a common, extensible representation for all expressions allows any class that operates on Multi_Exprs to also operate on their subclasses. For example, rules (e.g., join associativity) can be applied not only to queries but also to predicates, for example, to implement a semantic optimization rule such as 'E.salary < 5000 => False'. Layering expressions on top of the Directed_Graph class permits us to easily implement directed graph-type functions, such as depth-first search, on the nodes in a query. Other functionality we can exploit because of the reuse of the Multi_Expr class is that provided by the Expr_Class and Expr_Space classes to efficiently store groups of related expressions, for example to prevent redundant allocation of predicates or compactly store an optimizer's search history. We describe these classes next.

[1] We note that EROC uses C++ inheritance mechanisms to reflect application domain subtyping (*public* inheritance in C++) and to enable code reuse where subtyping does not apply (*private* inheritance in C++). An example of application domain subtyping is the Operator_Argument and Predicate relationship, while code reuse is enabled by the Multi_Expr/Predicate_Component relationship. See [Cop92] for a discussion of object reuse in C++.

2.1.2 The Expr_Class and Expr_Space Classes

Expressions may be grouped together for various purposes. For example, different representations of a query can be grouped together if they are algebraically equivalent. The Expr_Class class is an abstraction to support such groupings. The Expr_Space abstraction also groups logically related expressions, but provides the following additional features:

- *compactness*, because the semantics of the class are that the same expression will never be stored twice, and inputs to Multi_Exprs can be classes of expressions (i.e., an Expr_Class), not simply single Multi_Exprs, and
- *fast lookup* of expressions.

An important use of the Expr_Space class in EROC is for compactly storing an optimizer's search history. In the next section we describe the classes that support EROC's implementation of search space *enumerators*, which generate equivalent alternative representations of a query, and show how spaces of logical expressions produced by these enumerators can be mapped to corresponding spaces of physical expressions based on an abstraction we call a *mapper*.

2.2 Search Components - Expression Space Enumerators and Mappers

In this section we describe the two EROC classes that are fundamental to building search engines. The Enumerator class generates spaces (Expr_Space instances) of alternative representations of a query, while the Mapper class maps these alternatives to another space (i.e., a space of physical plans) based on cost.

2.2.1 The Enumerator and Rule Classes

We present two Enumerator instances included in EROC, a generative enumerator and a transformational enumerator. The algorithms they implement are not novel - rather the purpose of describing the enumerators is (1) to show reusable functionality available to optimizer developers who use EROC, and (2) to familiarize the reader with the two types of enumeration

as a basis for understanding how we composed search components to form NEATO's hybrid search engine.

EROC's Enumerator class is an abstract base class [Str91]. In the current version of the toolkit, there are two subclasses of Enumerator, BU_Enumerator ('bottomup', also known as a 'generative' enumerator) and Trans_Enumerator ('transformational' enumerator). The former is an implementation of a variant of the Starburst join enumeration algorithm, while the latter is a general purpose rule-based enumerator similar to the one in Volcano.

The BU_Enumerator Class

A BU_Enumerator instance accepts as parameters (to a member function 'apply_bu')

- a target Expr_Space instance,
- a list of join predicates, instances of the class Join_Predicate, a subclass of Predicate,
- a list of non-join predicates, instances of the class Predicate (e.g., selection predicates),
- a list of input relations, instances of the Composite_Collection class (a subclass of Collection),
- a sort constraint (final sort order),
- a projection list (those attributes that should be in the output),
- a groupby argument, an instance of the Groupby_Arg class, and
- a having clause, which is a Predicate instance.

When apply_bu completes, the Expr_Space instance will contain a set of expressions represented as Multi_Exprs. The number of Multi_Expr nodes will be the number of *feasible joins* [OL90] for the input query, plus additional nodes to represent groupby and having clauses. The implementation supports the search parameters of the Starburst algorithm, i.e., options to limit the number of relations in the inner join input and to enable/disable Cartesian products. The current implementation places non-join predicates as low in expressions as possible, although it is simple to extend the implementation to find other interleavings of selections and joins (e.g., to handle expensive predicates [HS93]).

The Trans_Enumerator and Rule Classes

A Trans_Enumerator instance is invoked with a set of transformation rules (instances of the Rule class), a Multi_Expr (the query to be optimized), and an Expr_Space instance. After application, the Expr_Space contains a representation of all expressions derivable using the transformation rules. For example, given a complete set of join transformation rules and a join expression, a Trans_Enumerator instance will produce the same space of feasible joins as a BU_Enumerator instance. The current implementation of Trans_Enumerator follows the left-to-right depth first rule application strategy of Volcano. EROC provides complete implementations of many common relational algebra transformation rules so implementors do not have to redevelop them.

We note that the Trans_Enumerator class is applicable to predicates (because the Predicate class is derived from the Multi_Expr class). In EROC we exploit this reusability to transform predicates (specifically, conjunctive join predicates) to be sure we detect equivalences among join expressions whose predicates may contain the same conjuncts in different sequences. For example, this ensures that the following two join queries are detected as equivalent because both predicates will be in the same Expr_Class instance (in the Expr_Space for predicates).

- join (R.a=S.b $\wedge$ R.a1=S.b1 $\wedge$ R.a2=S.b2) (R S)
- join (R.a1=S.b1 $\wedge$ R.a2=S.b2 $\wedge$ R.a=S.b) (R S)

2.2.2 The Mapper Class

Logical expressions are mapped by an optimizer search engine to a space of physical expressions. In the EROC architecture this mapping function is captured by the abstraction mapper. Given a (source) logical expression space, a (target) physical expression space, and a goal (an instance of Goal), a Mapper instance will return a physical expression that meets whatever constraint is specified by the goal. A goal is (1) a query, an expression space, *or* an expression class, together with (2) a set of properties the optimized plan must return For example, a goal may be the query 'join R.a=S.b R S' with the constraint 'sorted on S.b', which may be mapped to 'merge_join R.a=S.b (sort/partition R.a R) (sort/partition S.b S)'.

There is an additional class Map that records the mappings from goals to physical algebra expressions, i.e., the mappings from expressions in the logical search space to expressions in the physical search space. There is one Map instance for each Expr_Class in the logical search space. The Map class supports dynamic programming (in the Volcano_Mapper, for instance) because goals are only solved once and the solution (physical plan) stored. If a goal is requested again, the solution is returned.

EROC currently has one Mapper subclass, Volcano_Mapper, which is an implementation of the Volcano costing algorithm (described in detail in

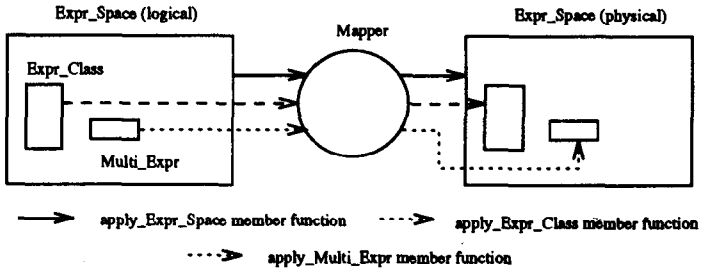

Figure 2: The Mapper Class

[McK93]). We will later show that in EROC this mapper is not restricted to being used with a transformational (rule-based) enumerator as it was in Volcano, but can be used with the BU_Enumerator class to form NEATO's search component, a highly efficient hybrid join optimizer. Figure 2 gives a high-level view of a Mapper and its member functions.

We make a final observation on EROC's search classes. EROC's Expr_Class and Expr_Space classes are similar to Volcano's equivalence class and MEMO data structures. However, in EROC the mapping from optimization abstractions to solution abstractions is much clearer. For example, in Volcano the MEMO *data structure* represented both the logical and physical search spaces, and there was no clear mapping from any particular optimization abstraction to this structure. EROC, on the other hand, makes this mapping much clearer, and from this we were able to better understand the composition of optimizer search strategies and use this understanding to develop NEATO's search engine. In the next section we present additional solution domain abstractions EROC provides to an optimizer implementor.

2.3 Other Important EROC Classes

We now describe other extensible EROC classes that provide common functionality which can be shared by optimizer developers.

2.3.1 Operator Classes

EROC provides several predefined operator classes, namely: 'bulk' operators, which consume and produce collections, such as Join, MergeJoin, NestedLoopsJoin, Groupby, FileScan, IndexScan; predicate operators, such as >, <, AND, OR, etc.; aggregate operators, such as Sum, Avg, Max, Min, Count; and arithmetic operators, such as +, −, /, modulo, etc.

The reason for predefining these operators is that in many data models the semantics of the operators are similar. For example, a join operator typically produces an output type[2] that is the concatenation of its input types. As another example, a merge join operator requires its inputs to be sorted on appropriate joining attributes, and produces sorted output. In other optimization tools, this functionality has to be coded by the optimizer implementor. In EROC, however, a reasonable semantics is provided for operators but an optimizer implementor is free to redefine (via virtual functions) some or all operator semantics, or add a completely new operator class to the Operator hierarchy.

Each operator has a 'derive_property' member function that determines the output (collection[3]) properties of an expression involving the operator. An important property derivation EROC provides is *key derivation*, which permits optimizations such as elimination of SQL distinct operations. (See [Sha92] for a discussion of key derivation and elimination of SQL distinct operators.)

2.3.2 The Operator Argument Classes

EROC provides the following common operator argument classes:

- sort arguments (class Sort_Arg), layered on ordered lists of Predicate_Components (class Predicate_Component_List);
- groupby arguments (class Groupby_Arg), layered on lists of Aggr_Exprs and Path_Exprs;
- predicates;
- partitioning and projection arguments (both layered on class Predicate_Component_List).

All Operator_Argument classes guarantee that duplicate instances will not be allocated. These classes also provide member functions to support syntax checking, i.e., to see if an operator/argument pair is valid given one or more input collections.

[2] Support for type information is provided by the Type_Info class, which is also (re)used to provide run-time type identification (RTTI) [Str91] for EROC classes.

[3] The Collection class encodes meta-information an optimizer needs about collections (e.g., files, relations), such as cardinality, type, partitioning, sort order, etc.

3 The NEATO Join Optimizer

In this section we discuss the motivation for NEATO's development, extensions we made to EROC to create NEATO, and performance of NEATO on optimizing two complex TPC/D queries.

The current Teradata optimizer gives excellent performance and has been highly tuned over many years of use. However, extending the optimizer with new techniques (e.g., new search strategies, theta semijoins, predicate movearound) is not as easy as it would be in an EROC-based optimizer. Out of this desire for extensibility the NEATO project was born. The goals of the NEATO project were (1) show that an EROC-based optimizer does not degrade (and hopefully improves) performance, and (2) EROC is indeed extensible enough to allow incorporation of Teradata-specific optimizations and operators, as well as optimization techniques not found in the current optimizer.

3.1 Search Strategy

NEATO implements a hybrid search strategy, using EROC's BU_Enumerator class for join order enumeration and the Volcano_Mapper to perform mapping from logical to physical operators. To our knowledge this is the first implementation that combines these two strategies, which have before been seen as mutually exclusive. The strategy first enumerates all join (and, optionally, Cartesian product) orders, and the mapper then traverses and maps this space. We decided to use the Starburst-style join enumerator because it is relatively easy to understand and performs better than the transformational enumerator if joins *and* Cartesian products are being considered. We decided on the Volcano mapping algorithm because it is very efficient and its top-down costing and constraint passing allow discovery of plans that are harder to find with a bottomup mapping strategy [McK93]. Together, this enumerator and mapper generate the entire space of join expressions and considers all possible mappings to physical operators. Such a strategy guarantees we will find the execution plan with minimum estimated cost, and the complexity of the join problem for TPC/D benchmark queries is sufficiently low to permit exhaustive search.

3.2 Extensions to EROC

Table 1 shows the number of lines of code for EROC optimization classes[4] and NEATO extensions to these classes. The entries show the amount of code to implement: basic expressions (including expression classes and spaces); enumerators (both transformational and Starburst-based) and the Volcano mapper; transformation rules (including associated pre- and post-condition code); operators (including property derivation); collections; types; predicate components (predicates, path exprs., etc); other operator arguments (e.g., sort and groupby); miscellaneous classes. The extensions listed here were the *only* additional code we required to build NEATO[5].

We note that search algorithm implementations comprise only 2418 lines, or 7% of the toolkit optimization code. The mechanisms to store expressions (e.g., search history) comprise another 1656 lines, or 5% of the total (counted as part of basic expression code in Table 1). This implies that only providing a generic search engine (or engines) and a mechanism to store search history, as other optimization tools do, leaves an optimizer implementor a great deal of complex code that must be implemented. EROC, on the other hand, eliminates the need for much of this redevelopment. We also note that development of the search components required a relatively small amount of total development time. For example, both the Starburst-style and transformational enumerators required approximately one person-week each to develop, while the Volcano mapper took about three person-weeks to code and test.

NEATO currently optimizes for Teradata's two most important join algorithms, merge join and product (nested loops) join, full file scan, and four of Teradata's index access methods[6] The primary extensions we made to EROC to create NEATO were to (1) incorporate Teradata cost calculations for these seven operators, and (2) encode Teradata-specific mappings from goals to partial solutions (which are used by EROC's Volcano-based mapper class). These major extensions make up the value in the 'Oper.' column of Table 1 for NEATO.

The first extension was easily accomplished by creating seven new subclasses, namely TeraDataMergeJoin (a subclass of EROC's MergeJoin class), TeraDataProductJoin (a subclass of EROC's NestedLoopJoin class), TeraDataFileScan (a subclass of EROC's FileScan class), TeradataPrimaryIndexScan (a subclass of EROC's IndexScan operator), etc. For each new subclass the 'cost' member function was redefined.

For the second extension, we simply derived the new classes TeradataJoin (a subclass of EROC's Join class) and TeradataGet (a subclass of EROC's Get operator) and redefined the superclasses' virtual functions that map goals to partial solutions. This member function is called 'get_partial_solution' and given a Goal (see Section 2.2.2 above) returns a set of physical opera-

[4] EROC support classes comprise another 7212 lines.

[5] There is additional code to *interface* NEATO with the Teradata parser and execution engine.

[6] At the time of writing we are introducing the Teradata index nested loops algorithm and other access methods.

Table 1: Lines of Optimization Class Code

System	Expr.	Enums. and Map.	Rules	Oper.	Colls.	Types	Pred. Comps.	Other Args.	Other	Total
EROC	5366	2418	4121	4792	3524	4193	4350	2709	1129	32602
NEATO	0	0	0	6977	318	222	408	0	406	8331

tors and associated input constraints that partially or fully satisfy the Goal. (See [McK93] for a complete explanation of Volcano's goal-driven search algorithm.) This partial solution code is straightforward to implement. However, there are many mapping choices in a parallel system (because of the combinations of partitioning/replication/sorting strategies), and therefore more partial solutions than one would have to encode for a non-parallel DBMS. These mapping choices account for the relatively large number of lines of code for NEATO under the heading 'Oper.'.

3.3 Performance

Tables 2 and 3 shows NEATO performance on optimizing TPC/D queries 5 and 8 for a 300 GB database. We chose these two queries because the number of joins is large (for the benchmark), with Query 5 containing joins of 6 relations and Query 8 containing joins of 8 relations. The optimization times do not include the time to parallelize the plans since this is done in a post-optimization phase in Teradata.

Table 2 shows exhaustive[7] optimization without Cartesian products, while Table 3 shows results from considering all joins plus Cartesian products. The columns in the tables show enumeration, mapping, and total optimization times, estimated execution costs[8] maximum heap space, and the numbers of Multi_Exprs and Expr_Classes in the logical and physical expression spaces at the end of optimization. The experiments were run on a 150 MHz Sparc 20 with a memory size of 128 MB.

We note several points about NEATO's performance and plan quality. First, for Query 5 Cartesian product introduction reduced the estimated plan cost by approximately 2%, while the plan cost for Query 8 is the same in both cases. We found the additional investment in optimization time is worthwhile because of the net savings in combined optimization and estimated execution time. Second, NEATO's plan quality is guaranteed to be at least as good as that of Teradata's current optimizer because all plans are considered (using NEATO's current search strategy), and NEATO in fact finds mappings that the current (non-exhaustive) optimizer does not.

Third, we note the complexity of the search space for these queries. Table 2 shows that the size of the logical search space for Query 5 (8) is between the number of feasible joins for a 6-relation (8-relation) linear-shaped join and a 6-relation (8-relation) star-shaped join. (See [OL88] for a discussion of feasible joins). Table 3 shows that NEATO considers the maximum number of feasible joins for each of the queries[9]. The complexity of Query 8 with all Cartesian products considered is between the complexities of a 10- and an 11-relation star join (without Cartesian products), and between the complexity of a 26- and a 27-relation linear join query (without Cartesian products). This shows that even using its current exhaustive search algorithm NEATO can easily find optimal plans for queries of these complexities.

Finally, NEATO's optimization time is dominated by the mapping phase. The reason is the relatively large number of mappings of goals to partial solutions in a parallel DBMS (like Teradata's). This suggests we focus research into faster mapping algorithms rather than faster enumeration algorithms. We believe a prime candidate for this research is the parallelization of the Volcano mapping algorithm. We summarize by noting that EROC was easy to extend to produce an optimizer tailored to the Teradata environment, and gives excellent performance on optimizing complex TPC/D queries.

During NEATO's development the question arose as to how to extend NEATO's search strategy to handle queries whose complexity is clearly too high for exhaustive search. Two strategies we consider promising to handle these queries are randomizing [IK90, GLPK94] and greedy [CLR89] algorithms. We plan to implement these strategies using combinations of EROC's existing enumeration and mapping classes (or extensions to these), and discuss these plans in the future work section below.

[7] That is, all physical operators and partitioning/replication /sorting schemes were considered for all left-deep, right-deep, and bushy join trees.

[8] For proprietary reasons we do not specify the cost units.

[9] The number of points in the logical search space is slightly higher than the number of feasible joins because additional points are allocated for non-join expressions such as groupby.

Table 2: No Cartesian Products

Query	Enum. Time (sec.)	Map. Time (sec.)	Total Time (sec.)	Est. Cost	Heap (MB)	Logical Multi Exprs	Logical Expr Classes	Physical Multi Exprs	Physical Expr Classes
5	0.06	0.26	0.32	3536	3.2	74	30	47	47
8	0.10	1.19	1.29	3609	4.4	124	44	142	142

Table 3: All Cartesian Products

Query	Enum. Time (sec.)	Map. Time (sec.)	Total Time (sec.)	Est. Cost	Heap (MB)	Logical Multi Exprs	Logical Expr Classes	Physical Multi Exprs	Physical Expr Classes
5	0.20	1.31	1.51	3473	4.9	307	63	130	130
8	1.35	10.39	11.74	3609	17.9	3033	255	583	583

4 Related Work

Both Volcano and EXODUS are based on transformational, algebraic optimization. They differ primarily in their mapping algorithms and interleaving of enumeration and mapping. They provide implementors with a single generic search engine, a rule code generator, and an extensible framework. OPT++ [KD95] offers an implementor a variety of search engines (randomizing, System R-style, Volcano-style) in an extensible framework, and was influenced by object-oriented design to support extensible search strategies found in [LV91]. All three of these tools provide one or more search strategies and leave the optimizer implementor the tasks of writing operator code, predicate and other operator argument code, catalog and property derivation functions, etc. We have shown that in EROC only about 12% of the code is devoted to providing both Volcano- and Starburst-style search components, and the remainder is devoted to supporting those functions absent from these previous systems. While EROC provides search components, it can also be viewed as complementary to these other optimization tools. For example, EROC's operator argument and catalog classes could be used with the Volcano Optimizer Generator to create an OpenOODB-type optimizer.

In [LVZZ94] a query optimizer for a parallel database system is described, namely the EDS optimizer. This optimizer, like EROC, is built on the principle of extensibility through object-oriented techniques. The authors see this approach as a contrast to the declarative rule-based approach to extensibility taken by EXODUS and Volcano, while EROC's extensibility encompasses the rule-based approach. Their implementation is aimed at allowing extensibility by themselves rather than outside implementors and, as in OPT++, they have achieved extensibility primarily in the search strategy.

Cascades [Gra95] is an optimizer framework being used as the basis for optimizers for Tandem's NonStop SQL [CKP+93] and Microsoft's SQL Server. This framework is based on object-oriented principles and includes a new optimization algorithm, based on the ordering of tasks, that permits exhaustive and heuristic transformation-based (rule-based) optimization. EROC differs from Cascades in that EROC focuses on providing building blocks that can be used to construct any type of optimizer search strategy rather than on providing a single optimization algorithm. It is also difficult to compare the two since it is unclear (1) how much reusable code Cascades provides, (2) how much effort is required to build an optimizer based on Cascades, and (3) what the performance of a Cascades-based optimizer is.

5 Summary, Conclusions, and Future Directions

EROC maps optimization abstractions (predicate, search space, enumerator, mapper, collection, type, etc.) to C++ classes, and mixes application domain subtyping, inheritance of implementations, and layering to achieve reuse and extensibility. EROC proved valuable in building an efficient new join optimizer for the Teradata massively parallel DBMS. This optimizer, NEATO, not only gives excellent performance and finds low-cost plans for complex TPC/D queries, but is understandable and extensible. EROC's reusable classes saved us a great deal of development effort since most of the code needed for the new join optimizer was already provided, and the extensions (customizations) we made were straightforward. We are also continuing development of an optimizer for

Ode using EROC, and finding that EROC's classes are sufficiently general to support development of an OODBMS optimizer. This finding is not surprising since EROC evolved in part from experiences building the OpenOODB optimizer. EROC provides more functionality than previous optimization tools by extending good modeling and reuse techniques to all aspects of optimizer development rather than limiting its scope to the search problem.

Future Work

We plan to enhance EROC's functionality by making the following extensions:

- implementations of randomizing and greedy search algorithms;
- a new subclass in the Predicate_Component hierarchy to support nested queries, and addition of unnesting techniques [Kim82, GW87, Day87, Mur92] to the toolkit[10];
- rules to support theta semijoin transformations;
- predicate inferencing support;
- a GUI to support optimizer development and use, e.g., to display and manipulate expressions, catalog information (e.g., collection and type information), etc.

We are using EROC to build an automated tuning tool, which will suggest alterations to a physical database design based on a given query suite. A tuning option we are exploring is caching of solutions for common subexpressions in a query suite. EROC's Expr_Space class is being used as a basis for this cache since it supports common subexpression detection. EROC is also providing an experimental platform for research in classifying and analyzing optimization search algorithms [GLL+95].

NEATO's search strategy will be extended to support optimization of queries whose complexity is too high to permit exhaustive search. To make these extensions we plan to build on our observations that many common search strategies can be described by their enumeration and costing methods, the interleaving of these methods, together with application of special search techniques, such as dynamic programming and branch-and-bound pruning. For example, greedy search techniques typically use a bottomup enumeration strategy interleaved with costing and pruning of both the logical and physical search space. System R also uses a bottomup enumerator and interleaves costing, but does not prune the logical space as aggressively as greedy search techniques, and augments the search with dynamic programming. Volcano uses a non-interleaved strategy with a transformation-based enumerator. Since we have the basic building blocks (enumerators, costing algorithm, common search space representation) we should be able to add alternative strategies easily. We also plan to extend the toolkit with a component to estimate query complexity and invoke the most appropriate search strategy.

We also want to experiment with techniques for commuting groupbys and joins [CS94, GHQ95]. Our strategy is to add another step to NEATO's hybrid strategy. This step will consist of invoking a transformational enumerator with a rule to commute groupby and join on an expression space produced by a BU_Enumerator instance. We believe that implementing this technique is more natural using transformation rules than integrating it *into* a bottomup enumerator. We have a common representation for a search space, so are free to mix transformational and generative enumeration strategies that operate on the representation.

[10] The current Teradata optimizer uses sophisticated unnesting techniques developed by Teradata. The NEATO join optimizer will be coupled with this existing unnesting code to form a complete SQL optimizer. The unnesting algorithms we are going to add to EROC are similar to those found in the literature, and our intention is to spare EROC users from having to reimplement these common techniques.

Acknowledgements

We wish to thank Latha Colby, Dennis Shasha, and Divesh Srivastava for their excellent suggestions on the paper. We thank Pekka Kostamaa and Ramesh Bhashyam for providing information about the Teradata architecture. We wish to thank NCR management, notably Alan Chow and Malla Reddy, for their support and commitment to this project. A special thanks to Narain Gehani for his support, suggestions, and encouragement during the EROC and NEATO projects.

References

[AG89] R. Agrawal and N. Gehani. ODE (Object Database and Environment): The language and the data model. In *Proc. ACM SIGMOD Conf.*, page 36, Portland, OR, May-June 1989.

[ATT95] ATT Global Information Solutions Company. *Teradata DBS Concepts and Facilities Manual (UNIX) - Product ID D1-4636-A*, 1995.

[BMG93] J. Blakeley, W. McKenna, and G. Graefe. Experiences building the Open OODB query optimizer. In *Proc. ACM SIGMOD*

Conf., page 287, Washington, DC, May 1993.

[CKP+93] A. Chen, Y. F. Kao, M. Pong, D. Sak, S. Sharma, J. Vaishnav, and H. Zeller. Query processing in NonStop SQL. *IEEE Data Eng. Bull.*, 16(4):29, December 1993.

[CLR89] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, NY, 1989.

[Cop92] J. Coplien. *Advanced C++: Programming Styles and Idioms.* Addison-Wesley, Reading, MA, 1992.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. Intl' Conf. on Very Large Data Bases*, page 354, Santiago, Chile, August 1994.

[Day87] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proc. Intl' Conf. on Very Large Data Bases*, page 197, Brighton, England, August 1987.

[GD87] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *Proc. ACM SIGMOD Conf.*, page 160, San Francisco, CA, May 1987.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate processing in data warehousing environments. In *Proc. Intl' Conf. on Very Large Data Bases*, page 358, Zurich, Switzerland, September 1995.

[GLL+95] P. Gibbons, A. Levy, D. Lieuwen, Y. Matias, W. McKenna, I. S. Mumick, D. Srivastava, S. Ganguly, S. Sudarshan, R. Bhashyam, L. Burger, C. Hoang, P. Kostamaa, and A. Witkowski. A survey of query processing techniques with recommendations for the Teradata database. Technical report, Bell Laboratories, Murray Hill, NJ, December 1995.

[GLPK94] C. Galindo-Legaria, A. Pellenkoft, and M. Kersten. Fast, randomized join-order selection-why use transformations? In *Proc. Intl' Conf. on Very Large Data Bases*, page 354, Santiago, Chile, August 1994.

[GM93] G. Graefe and W. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. IEEE Int'l. Conf. on Data Eng.*, page 209, Vienna, Austria, April 1993.

[Gra87] G. Graefe. *Rule-Based Query Optimization in Extensible Database Systems.* PhD thesis, Univ. of Wisconsin-Madison, August 1987.

[Gra95] G. Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19, September 1995.

[GW87] R. A. Ganski and H. K. T. Wong. Optimization of nested SQL queries revisited. In *Proc. ACM SIGMOD Conf.*, page 23, San Francisco, CA, May 1987.

[HS93] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. ACM SIGMOD Conf.*, page 267, Washington, DC, May 1993.

[IK90] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proc. ACM SIGMOD Conf.*, page 312, Atlantic City, NJ, May 1990.

[KD95] N. Kabra and D. DeWitt. OPT++: An object oriented implementation for extensible database query optimization. *unpublished manuscript*, 1995.

[Kim82] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443, September 1982.

[LMS94] A. Levy, I. Mumick, and Y. Sagiv. Query optimization by predicate move-around. In *Proc. Intl' Conf. on Very Large Data Bases*, page 96, Santiago, Chile, August 1994.

[LV91] R. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *Proc. Intl' Conf. on Very Large Data Bases*, page 363, Barcelona, Spain, September 1991.

[LVZZ94] R. Lanzelotte, P. Valduriez, M. Zait, and M. Ziane. Industrial-strength parallel query optimization: Issues and lessons. *Inf. Sys.*, 19(4):411, 1994.

[McK93] W. J. McKenna. *Efficient Search in Extensible Database Query Optimization: The Volcano Optimizer Generator.* PhD thesis, University of Colorado-Boulder, May 1993.

[MGB93] W. McKenna, G. Graefe, and J. Blakeley. Experiences building the Open OODB query optimizer. In J. Blakeley, editor, *Proc. of the Workshop on Database Query Optimizer Generators and Rule-base Optimizers*, page 13, Dallas, TX, September 1993.

[Mur92] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proc. Intl' Conf. on Very Large Data Bases*, page 91, Vancouver, BC, Canada, August 1992.

[OL88] K. Ono and G. M. Lohman. Extensible enumeration of feasible joins for relational query optimization. Research Report RJ 6625 (63936), IBM, December 1988.

[OL90] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. Intl' Conf. on Very Large Data Bases*, page 314, Brisbane, Australia, August 1990.

[Pv93] R. Plasmeijer and R. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, Reading, MA, 1993.

[Raa95] Francois Raab, editor. *TPC Benchmark D - Standard Specification.* Transaction Processing Performance Council, Shanley Public Relations, 777 N. First Street, Suite 600, San Jose, CA 95112-6311, May 1995.

[Sha92] D. Shasha. *Database Tuning: A Principled Approach.* Prentice-Hall, Englewood Cliffs, NJ, 1992.

[SHL+96] P. Seshadri, J. Hellerstein, T. Y. Leung, H. Pirahesh, R. Ramakrishnan, D. Srivastava, P. Stuckey, and S. Sudarshan. Cost-based optimization of complex queries: The magic of theta-semijoins. In *Proc. ACM SIGMOD Conf.*, Montreal, Canada, 1996.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1991.