Paper Id 86

Rule-Based Query Optimization, Revisited[*]

Lane B. Warshaw   Daniel P. Miranker
Department of Computer Sciences and The Applied Research Laboratories
The University of Texas at Austin
Taylor Hall 2.124
Austin, TX 78712-1188
Telephone: 512-835-3840
Fax: 512-835-3100
{warshaw, miranker} @cs.utexas.edu

Abstract

We present an overview and initial performance assessment of a rule-based query optimizer written in VenusDB. VenusDB is an active-database rule language embedded in C++. Following the developments in extensible database query optimizers, first in rule-based form followed by optimizers written as object-oriented programs, the VenusDB optimizer avails the advantages of both. To date, development of rule-based query optimizers have included the definition and implementation of custom rule languages. Thus, extensibility required detailed understanding and often further development of the underlying search mechanism of the rule system. Object-oriented query optimizers appear to have achieved their goals with respect to a clear organization and encapsulation of an optimizer's elements. They do not, however, provide for the concise, declarative expression of domain specific heuristics.

Our experience demonstrates that a rule-based query optimizer developed in VenusDB can be well structured, flexible, and demonstrate good performance. We compare a relational optimizer developed with Volcano and a functionally identical optimizer developed with VenusDB. The results demonstrate comparable performance on small queries with few joins, while the VenusDB optimizer scales better and outperforms Volcano on larger join-arity queries. Since we did not have to develop a specialized rule language or consider application specific programming constructs, the source code for the optimizer is small and straightforward, about one-third the size. Similar code comparisons with an object-based optimizer, the VenusDB optimizer reveals similar benefit.

[*] This research is partially funded by DARPA, contract number: F30602-96-2-0226 and the Applied Research Laboratories, University of Texas, Internal Research and Development Program

1.0 INTRODUCTION

The ability to express a query optimizer concisely and extensibly has been an ongoing goal of experimental database research. The first generation of this research comprised a number of rule-based query optimizers [GRD87, GRM93, DID95, PHH92]. Rule-based representation is attractive since it closes the semantic gap between the specification of an optimizer and its implementation. By virtue of a declarative specification, it follows that rule-based optimizers are much easier to extend than their predecessors. However, each of these first generation optimizers considered using the general-purpose rule-based languages that were available at the time and determined that they were unsuitable as the basis of a query optimizer[1]. These languages suffered from a lack of embeddability, execution speed and/or a lack of control over a perceived need to tightly regulate general-purpose search mechanisms for rule evaluation. Thus, each project developed a rule language and execution environment for the single purpose of developing their optimizer. These specialized rule environments contain features that go beyond the use found in general purpose rule-languages by including ad-hoc elements for controlling search and limiting the scope and applicability of rules, all in an effort to increase speed. Development of query optimizers in these systems often involves elaborate knowledge of both the rule language and its underlying execution engine comprising engineering at both levels. Consequently, rule-based optimizers have been successful in providing environments friendly to extending the search space by easily adding new transformation rules, but “they do not allow for extensibility in other dimensions,” often fixing the search strategies [OZM96].

Subsequent developers have moved to object oriented technologies [KAB94, OZM96]. These systems boast a well-encapsulated and structured environment by defining the query representation and cost model as first class objects and procedures respectively. Hence, defining new database operators often only involves deriving a class from the designated super-class and specializing its methods. Though it may be true that interchanging searches and transformations are easy in these systems, these features are not declaratively expressed. Therefore, defining new searches or refining existing searches with additional heuristics is not nearly as straightforward as in the rule domain. We develop quantitative software metrics that supports this claim.

The need for extensibility and modification of search strategies has been magnified by the extension of relational database management systems into arbitrary data-types [SES97]. Given this renewed need and a number of developments in rule-based execution, we have developed the hypothesis that effective query

[1] We agree with those findings.

optimizers may be built using a general-purpose rule language, VenusDB. VenusDB is an active database extension of Venus. Unique features of VenusDB that suggested revisiting the issue of rule-based query optimizers include,

- Embeddability in C++ and thus embeddability in database systems. VenusDB' data definition language is precisely C++. Thus, OO benefits seen in the optimizers developed by [OZM96, KAB94] with respect to the representation of operator trees can be exploited identically.

- A well-structured (modular) rule environment that allows a user to assert some procedurally inspired control over the execution while maintaining fixed-point semantics [BRG94].

- The addition of the *event* qualification in the condition-action rules which enables further structuring of the search, but not as an ad-hoc mechanism.

- Improvements in the implementation techniques of general-purpose rule compilers that commonly yield better than 100 times faster execution speeds over the general-purpose compilers available at the time of the earlier work (on identical hardware). An irony is that much of this performance gain is precisely due to the introduction of relational query optimization techniques in rule execution engines [OBE95, MIR90, WAN92].

At this juncture we have written a number of query optimizers using VenusDB. A general organization is emerging as well as substantially overlapping code segments from which we expect to define a library of design patterns from which an extensible set of search strategies may be refined [GAM95]. In this paper we describe the core organizational elements and report in detail on an optimizer we developed that is operationally equivalent to the sample relational optimizer provided with the Volcano distribution. The evaluation of optimization times and plans returned by the VenusDB optimizer compares quite favorably to its Volcano counterpart. In fact, while returning similar plans, the VenusDB optimizer actually executes faster on large queries. Given the success of OO optimizers, we also compare to Opt++ [KAB94]. Since that system itself has been shown to perform comparably to Volcano, our comparisons are limited to organizational aspects. Though it is difficult to compare systems written in different programming language methodologies, we have quantitative results and can speak to the ease of extensibility with respect to the introduction of additional search heuristics.

```
module  Optimize(TREE_NODE OptTable[], TREE_NODE TransformsCont[])
{
      rule  join_logical_transformations  priority 10;
      from  OptTable[?] node;
      event modify      node;
            none        OptTable;
      if( node.nodeOp() == JOIN )

            // call join_T_rules module
            join_T_rules(node, OptTable, TransformsCont);

      rule  join_physical_transformations priority 9;
      from  OptTable[?] node;
            OptTable[*] all_nodes;
      if( node.promise() >= all_nodes.promise())

            // call join_I_rules module
            join_I_rules(node, OptTable, TransformsCont);
}
```

Figure 1. Sample VenusDB Module with Module Calls

3.0 The VenusDB Rule Language

Many of the aforementioned deficiencies of early general-purpose rule languages are not unique to their application to query engines. Besides slower implementation techniques, these systems used LISP and LISP-like data structures which make it difficult to embed them in larger systems. Venus is just one of a number of object-embedded rule languages that have been developed to address these issues [PAC95].

3.1 Venus and VenusDB Syntax

Venus rules are organized into parameterized groups called modules. See Figure 1. Modules are designated by the key word `module` followed by a list of formal parameters and a list of local variables. The formal parameters and the local variable list are made up of containers and primitive variables. A container is Venus's set data type used for inferencing. Containers are distinguished by the use of square brackets "[ ]", and their elements are defined as C++ class instances. Primitive variables are inferable object instances that are semantically treated as a container of size one.

A Venus rule is made up of three parts, a *header*, a *condition*, and an *action*. The header contains the keyword `rule` followed by a rule name, an optional priority and a declaration section. The declaration section begins with the keyword `from`, followed by a list of container names and quantified cursor declarations. Cursors can be quantified both existentially and universally. An existentially quantified cursor is represented by a "?"

within the square brackets. A universally quantified cursor is represented by a "*" within the square brackets.

VenusDB is an extension of Venus such that an *event* clause has been added to the declaration section. In effect, this introduces refinements concerning the more focussed execution behavior of active-database ECA (event, condition, action) rules [OBE96]. The event clause is used to define precisely which events trigger a reevaluation of the rule, and is typical of other active-database rule languages. If an event is not specified, the specification defaults to all events and results in behavior equivalent to Venus. The other difference between Venus and VenusDB is an API facility for mapping Venus's main-memory containers to tables or extents in databases. We have previously reported on an application developed by exploiting a heterogeneous integration of VenusDB with multiple databases [OBE96].

A rule condition is a C++ boolean expression. C++ functions and method calls can be executed from within the rule condition provided they return a boolean value and execute side effect free. A rule's action is a list of C++ expressions, possibly including the name of a Venus module and its actual parameters. C++ statements and function calls can be made modifying data elements without any explicit notification to the rule system. The Venus compiler, either directly or with the help of the target databases trigger mechanism, produces an optimal trigger filter [OBE97].

3.2 Modularity and Semantics

The entire action of a Venus rule is defined to be a single atomic transition in a state-space (a transaction), and rules fire by a fair non-deterministic policy. Control remains within a Venus module until *fixed point* is reached [CHM88]. Modules may be listed in the action of a rule and can be nested arbitrarily deep. If a rule fires and its action lists module calls, then the rules within the nested modules must achieve fixed point before the action of the rule commits. Thus, Venus semantics and nested transaction models are closely related.

Though incorrect, it is often convenient to think of the condition of a rule that lists modules as being conjoined with the conditions of the rules within the nested modules, in effect, factoring out a common condition. This is incorrect because once a single rule fires, all rules in that module must fire until fixed point is reached, despite the value of the condition in the *calling* rule. These definitions yield a lotus of rule execution that corresponds to a depth first traversal of the *module-call graph*. Though declaratively defined, the behavior is consistent with procedural intuition.

3.3 Procedural Control

In some circumstances, a non-deterministic selection of satisfied rules makes programming difficult. When the paradigm has been strictly enforced by a rule language, program developers often introduce additional conditions, coined *secret-messages,* to force sequencing. This has been shown to result in poor quality code [MCD93]. Thus, despite fixed point semantics, Venus includes several clearly procedural constructs. Such constructs are admitted to the language only if there is a clear need and an obvious macro-like expansion into an equivalent, (if not slow and cumbersome), non-deterministic program.

The most critical of these procedural constructs is priorities. Rules with higher priorities will be given precedence over rules of lower priorities. Satisfied rules with the same priority are selected non-deterministically. Other procedural constructs include a return statement which forces a module to fixed point. The event clause also falls into this category.

3.4 An Example

As an example of how these declarative constructs interact and behave, consider the module in Figure 1. The `Optimize` module contains two rules, `join_logical_transformations` and `join_physical_transformations`. The `join_logical_transformations` rule defines an existential cursor over the `OptTable` container called `node`. The `node` cursor explicitly informs VenusDB to only monitor the modification of the cursor while shutting off monitoring on the `OptTable` container. This rule filters for join nodes, and passes cursors to the `join_T_rules` module. The `join_physical_transformations` rule defines a universally quantified and an existentially quantified cursor over the `OptTable` container. The rule finds the most "promising" node in the `OptTable` and passes it to the `join_I_rules` module.

The behavior of the module proceeds as follows. If the `join_logical_transformations` rule fires, control will be passed to the `join_T_rules` module until fixed-point is reached. At that juncture, control returns to the `Optimize` module and the `join_logical_transformations` will continue to evaluate its rule condition. This sequence will continue testing each node in the `OptTable`. Consider the instance when the passed cursor to the `join_T_rules` module is modified, the modified cursor will re-trigger the evaluation of the `join_logical_transformations` rule on this element. After all possible such evaluations of the `join_logical_transformations` rule has been performed, the `join_physical_transformations` rule will then evaluate its rule condition. If during the execution of the `join_I_rules` module an element is inserted into or deleted from the `OptTable`, upon fixed point and control returning to the `Optimize` module, the rule condition in the `join_logical_transformations` rule will not be re-evaluated. This is because events have been turned off on the `OptTable`. Had event monitoring not been turned off on the `OptTable`, the `join_logical_transformations` rule would immediately be evaluated based on the modification. Execution continues in this manner, bouncing back and forth between nested module calls until the state of fixed point is reached in the `Optimize` module.

4.0 The Query Optimizer

The three basic parts of a query optimizer consists of the cost model, search space, and search strategy [DID95]. Following efforts in object-oriented query optimizers, the Venus based query optimizer separates and encapsulates each of these three elements.

4.1 Operator Tree and Cost Model

Due to its connection to C++, the Venus-based optimizer's *operator tree* (the representation of the query graph) and its associated cost model are defined in terms of first-class C++ objects. These definitions closely resemble the object oriented operator tree definitions developed in other extensible query optimizers [KAB94, OZM95].

A base class, called `TREE_NODE`, contains the C++ virtual methods and data needed to define the node's *descriptor*, the logical and/or physical description of the node. Derived algebraic operators must specialize these method calls in a manner relevant to the operator. Beyond this API, the structure of an operation may take on any shape or form, including the number of operands upon which the operator maps over.

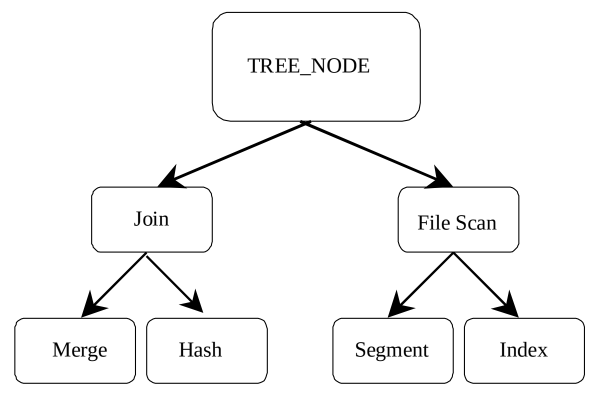

Figure 2. Sample Operator Tree

Figure 2 displays the basic structure of a relational operator tree defined using `TREE_NODE`. In the figure, the logical operations of join and file scan are derived from `TREE_NODE`. The physical implementing operations of merge join and hash are further derived from the join operation. Likewise, segment scan and index scan are derived from file scan.

4.2 Organization of the Rule Base and Search

The structure of the optimizer, both conceptually and, nearly, syntactically, is illustrated in Figure 3. The top-level module(s) define the search algorithm used to find an optimal plan. These modules heuristically choose a sub-plan(s) to optimize passing it to the optimize modules. For example, using the Volcano search, this group of modules will pick a sub-plan that has not been optimized in the look up table.

The optimize modules then applies different transformation and implementation operations on this sub-plan by exploiting procedural and heuristic elements, constrained by conditioning on the algebraic representation. If the preconditions are satisfied, the sub-plan is passed to the transformation or implementation modules. The new modified sub-plan is then passed to the post conditions modules which evaluates the sub-plan and updates appropriately. It is worth noting that due to the modular design of the system, heuristics can be added by simply adding a rule, condition, or module at any step.

Another consequence of this modular structure is that it leads us to believe that the implementation of new search methods can be well structured in the form of design patterns.

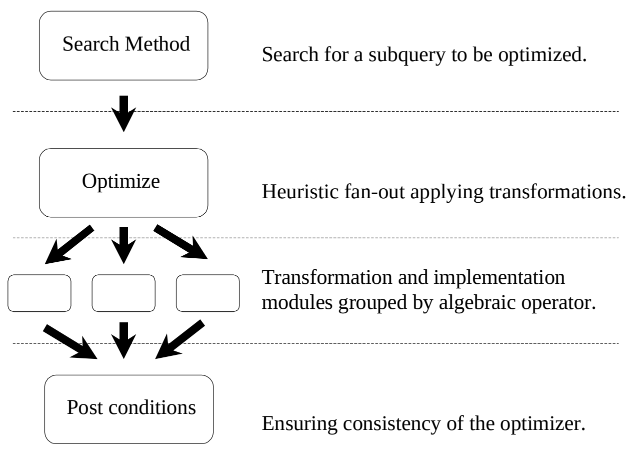
Search Method
Search for a subquery to be optimized.
Optimize
Heuristic fan-out applying transformations.
Transformation and implementation modules grouped by algebraic operator.
Post conditions
Ensuring consistency of the optimizer.

Figure 3. Rule System Architecture

```
//  Entry point module
module TransformativeSearch(// return value
                        TREE_NODE  best_plan,
                        // list of subqueries that
                        // may need to be optimized
                        TREE_NODE  OptTable[])
{
        int curr_level = 0; // level in the query graph

        rule generate priority 10;
        from OptTable[?] node;
        if( TRUE ) {

                // Search for optimizations
                Search(OptTable, curr_level);

                // increment level
                        curr_level++;
        }

        rule finished priority 0;
        if( TRUE ) {

                // grab the best plan
                GetBestPlan(OptTable, best_plan);
                return;  // force fixed point
        }
}

// Module to optimizes all nodes at the current level
module Search(TREE_NODE     OptTable[], int curr_level)
{
        TREE_NODE        TransformsCont[];  // local
container declaration

        rule Get_Node_to_Optimize;
        from OptTable[?] node;
        // turn events monitoring off on the
        // TransformsCont
        event none TransformsCont;
        if(// a node at the current level in the tree
           // has not been optimized
           node._logical->depth() == curr_level &&
           node._optimized == UN_OPTIMIZED) {

                // Link to the most “promising”
                // operands from the level bellow
                LinkNode(OptTable, node);

                // call Venus module to try to
                // Optimize the node
                Optimize(node, OptTable,
                        TransformsCont);
        }
}
```

Figure 4 Transformative Search in Venus

```
rule associate;
from OptTable[?] child;
if ( // child is a Join node
     child.NodeOp() == JOIN &&
     // and an operand of the node to optimize
     child.isOperand(op_node) &&
     // meets the requirements of associativity
     isAssociative(op_node, child)) {

         // Then apply transformation
         TREE_NODE optimized_node(Associate
                               (op_node, child));
         // Call VenusDB module which tests
         // post-condition statements
         postConditions(optimized_node,
                        OptTable);
}
```

Figure 5. Venus Associativity Rule

```
%trans_rule  (JOIN ?op_arg1 ((JOIN ?op_arg2
                                (?1 ?2)) ?3))
        ->   (JOIN ?op_arg3 (?1 (JOIN ?op_arg4
                                (?2 ?3))))
%cond_code
{{
        if (NOT (attr_in_equiv_class ((?op_arg1) ->
                            join_arg.operand1, ?2)))
                REJECT ;
}}
%appl_code
{{
        copy_operator_arg (?op_arg3, ?op_arg2) ;
        copy_operator_arg (?op_arg4, ?op_arg1) ;
}}
```

Figure 6. Volcano Associativity Rule

Design patterns are similar to that of a super-class representation of an object (only in structure because it can not be operationally, or physically encapsulated as an object to form a base class) since the pattern can be specialized for each particular search. There are results in the sub-areas of AI on planning and general purpose problem solving demonstrating that a small number of rule-based design patterns may be used as the starting to point to encode virtually all forms of search [SOL87, LAI87, LAI83, GAM95]. Our aim is to take this one more step and develop components enabling a plug and play approach to different search techniques. Our experimental implementations to date encompass top-down depth-first hill climbing, System R style bottom-up and a hybrid strategy of Volcano, (coined transformative search by [GRM93, KAB94]). We have converged, so far, to an architecture where algebraic transforms and implementation assignments are each encapsulated in rule modules whose definitions are nearly independent of search strategy. The structure of those patterns as they materialize in Venus is becoming more obvious. Our primary unknown is if we will conclude with a plug-and-play system or be satisfied with design-patterns and a library of fleshed-out examples.

The code for the transformative search is located in Figure 4, and is shown in its entirety to demonstrate the simplicity of experimenting with different search algorithms within the VenusDB rule environment. The `TransformativeSearch` module contains two prioritized rules. The highest priority rule calls the `Search` module applying optimizations while traversing the tree bottom up. (The current level pointer can be reset on lower level modules implementing recursive calls within VenusDB.) The `finished` rule, firing only after all optimizations have been processed, effectively calls a module to pick the best plan from the `OptTable` and then forces fixed point.

The `Search` module contains only one rule. The `Get_Node_to_Optimize` rule determines if a node has not been optimized. If this is the case, the node links with the most "promising" operands in the `LinkNode` module and is passed through to the `Optimize` module. As a simple example of a domain specific heuristic that can greatly effect the outcome of the optimized plan, the `LinkNode` module can encode knowledge to select operand links depending on the type of query and its structure.

Once a node is selected, two types of transformations are generally attempted, transformation operators (*T-Rules*) on the logical algebra and implementation operators (*I-Rules*) on the physical algebra. Figure 5 displays the T-rule for the association operator written in VenusDB. Figure 6 displays the equivalent operator expressed in the Volcano rule specification. One of the primary objectives in the development of the Venus rule language was to provide for a syntax that is familiar to a broader scope of people. In Figure 6, we see a direct resemblance of Volcano rules to that of the more classical LISP derived rule systems of OPS or CLIPS. Since the syntax of most of the rule languages written for query optimizers is similar to that of Volcano's, we believe that the learning curve to begin building optimizer rules in VenusDB will be significantly smaller than in earlier efforts.

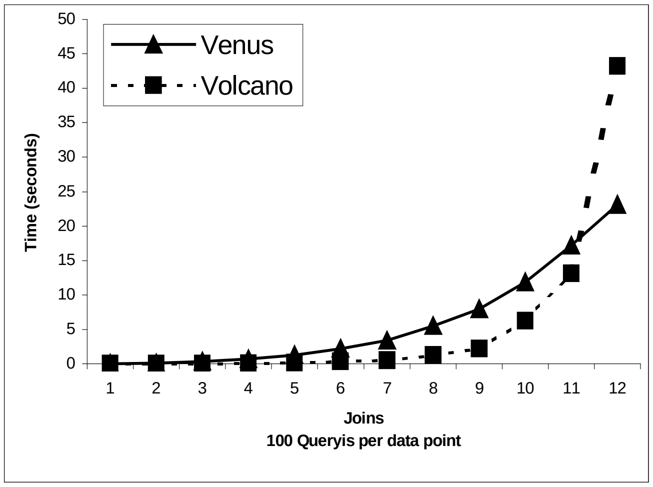

Figure 7. Optimization Time

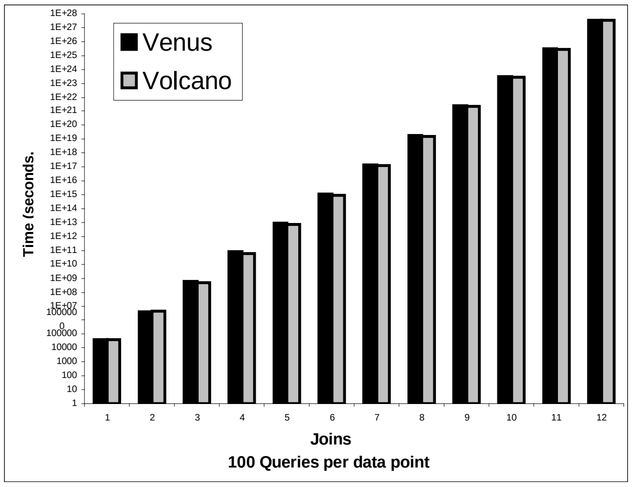

Figure 8. Estimated Cost of Optimized Plans

5.0 Performance

We chose the Volcano optimizer generator as our primary basis of comparison. Besides availability and reputation, we had in-house expertise due to a development effort layering in a new front-end [DID95][2].

Both relational optimizers contain two T-rules, and two I-rules, and the algebraic operators of scan, sort and

[2] We did not similarly execute the performance analysis of an object-oriented optimizer since they are shown to be equivalent to Volcano in [KAB94].

join. We then ran the optimizers on 12 sets of 100 randomly generated queries. Each set targeted queries from one to twelve joins respectively. We tabulated optimizer execution time and estimated plan cost. The random queries were generated using the query generator described in [BAY96] for very large data sets. The generator produces both queries and statistics concerning data distributions. Rather than purely random query graphs, the generator is concerned with producing queries consistent with those seen in real applications. We set the generators parameters, b and p, to 1.5 and 0.1. These values produce a balanced mix of loss-less foreign key like joins and their complement. The experiments were performed on a Sun Ultra 2 containing 2 x 167 MHz UltraSparc-1 processors with a 0.5 MB cache and 320 MB main memory. We did not exploit the parallelism.

Figure 7 presents the execution times of the optimizers, and Figure 8 presents the total estimated cost for the plans produced. The cost of the plans produced by VenusDB and Volcano are nearly identical. The surprising trend is that the VenusDB optimizer scales better than Volcano with respect to join-arity. We believe there are two reasons for this trend. First, we think that our abstractions in Venus allowed us to define a more complete definition of the equivalence of two sub-plans avoiding the re-computation of many sub-plans. Second, the (query-derived) optimization techniques embedded within Venus excel with larger problem size [OBM95].

Recall, one question in our hypothesis is whether the execution speed of a general-purpose rule-based language could be comparable to a specialized search engine. By reflecting as best we can the Volcano search strategy, examining total optimization time suggests we succeeded. A finer grain measure, such as number of plans generated and evaluated per second would be more definitive. We simply could not align the two implementations close enough to instrument for this result.

6.0 Quantitative Measurements

Quantitative software metrics are an objective measure used to analyze the complexity and life cycle cost of a system. We quantify three systems, optimizers developed in Opt++, Volcano, and VenusDB respectively [GRM93, KAB94]. First, we calculated the total lines of code. This metric gives evidence of a system's overall complexity [BAS79]. Next, we computed McCabe's cyclomatic complexity. Cyclomatic complexity measures the number of possible paths through a program which is indicative of control flow complexity [MCC76]. As a brief description, the cyclomatic complexity of a simple sequential procedure with no branches receives a rating of 1. Code is considered unmanageable if it is rated 11 or higher. Cyclomatic complexity was developed in the context of Fortran and C. In the case of object-oriented programming, excessive use of accessor functions significantly reduces the average case rating of a program. For this reason, we decided only to analyze procedures with a rating greater than 1, effectively factoring out well designed constructs while focusing on the more complex. Relating cyclomatic complexity to the rule-based paradigm, we developed a simple procedural model of rule execution [WAR96].

6.1 Volcano Comparison

The Volcano rule optimizer was implemented using 12,524 lines of code, of which 10,238 lines were used to define the operator tree, the rule search engine component, and the optimizer search strategy. 2,286 lines are unique to the individual relational optimizer. The VenusDB query optimizer was implemented using 4,222 lines of code of which 1,494 lines were used to define the operator tree, 2,163 lines are unique to the relational optimizer. 665 lines of rule code were needed to encode both the search strategy and transformations rules. Hence, 1/3 the amount of code is needed for VenusDB than in Volcano.

The average cyclomatic complexity of the Volcano optimizer is 20.0. The aggregate number may be decomposed to the search components as a group averaging 18.7. Elements constituting the sample optimizer rated 24.6. Thus, new Volcano developers deal with programs characterized by the highest of these ratings. The equivalent optimizer using VenusDB receives an average of 6.1. The aggregate number may be decomposed to the search components as a group averaging 6.2. Elements constituting the sample optimizer rated 5.4.

The large differences in the two systems on both measures are due to the OO structure of the operator tree and the non-specialized, declarative implementation of search within the VenusDB optimizer. The net result is a system with roughly a 1/3 of the complexity of the Volcano optimizer.

6.2 Opt++ Comparison

Citing the advances in optimizer research, we deemed it necessary to quantify the code of a VenusDB system to an established object-oriented optimizer. Because of its availability, we chose the Opt++ optimizer developed for the SHORE project [KAB94]. The example Opt++ distribution is modeled for a basic object-oriented database. For this reason, we implemented an equivalent optimizer within VenusDB by extending the optimizer described in the previous sections.

The sample Opt++ optimizer contains 9,007 lines of code, while the VenusDB equivalent contains 5,333 lines, roughly half. The McCabe measurement yielded an average complexity of 7.0 overall for the Opt++ optimizer and 6.78 for its VenusDB equivalent. The similarity of the measurement is not surprising since both systems attack the implementation of the operator tree, cost model and application of operators identically. A closer inspection reveals an average of 5.5 to implement search within VenusDB (only slightly more complex than its corresponding relational optimizer), compared to Opt++' 6.1, better than a half point improvement.

Although this is not a complete quantitative analysis of the three systems, we believe this to be telling evidence of the extensibility of the VenusDB query optimizer.

7.0 Conclusions

The development of extensible query optimizers has been an ongoing experiment in trade-offs. Systems have necessarily given up one or more dimensions of flexibility with the expectation that the net value will be higher. To summarize, rule-based query optimizers have given up flexibility in search by developing specialized rule engines. In the case of Volcano, one single tightly integrated search strategy was embraced. This system did remain extensible with respect to new operators. However, on going maintenance and new optimizer development is encumbered with the details of a specialized rule engine. Quantitative software metrics confirm the challenging life-cycle costs of using Volcano. Object-based query optimizers have succeeded in developing a transparent organization. Yet, they do not easily admit to experimentation concerning heuristic improvements. Changes in search strategy, while well encapsulated, may involve rewriting large parts of the system.

Our results to date confirm a hypothesis that language definition and implementation techniques for general-purpose rule languages have improved to a point where they may be used to build effective query optimizers. As a consequence, a large portion of the code of specialized rule-based optimizers, measured in both volume and its impact on developers, can now be omitted. Within the structure of any one optimizer written using VenusDB, a developer may easily add new operators and refine search heuristics. As on going work, we are reviewing the construction of several optimizers built using VenusDB anticipating generic search structures such that optimizer developers may easily experiment with large-scale changes is search strategy. Object related elements in the definition of VenusDB, many of them typical of other contemporary rule-based languages, provide the benefits seen in object-based optimizers. This shows up primarily in the definition of the operator tree and cost model. Objects also assist in encapsulating other organization elements of the system that are exclusively rule-based in nature.

ACKNOWLEDGMENTS

We would like to thank Dinesh Das and Don Batory for their insightful comments and their help exploring the Prairie and Volcano optimizer. We would also like to thank Tao Wang for his help in the implementation effort.

8.0 REFERENCES

[SOL87] E. Soloway, J. Bachant, K. Jensen, "Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a VERY Large Rule-Base." Association for Artificial Intelligence, *Proceedings of the National Conference on Artificial Intelligence* (AAAI-87), August 1987.

[BAS79] V.Basili, R. W. Reiter, Jr. "Evaluating Automatable Measures of Software Development." In *Proceedings on Workshop on Quantitative Software Models*, October 1979, 107-116.

[BAY96] R. Bayardo, D.P. Miranker, "Processing Queries for the First-Few Answers." *Proceedings of the 5th Conference of Information and Knowledge Management*, November 1996, 45-52.

[BRG94] J.C. Browne, et. al. "A New Approach to Modularity in Rule-Based Programming." *In Proceedings of the 6th International Conference on Tools with Artificial Intelligence,* IEEE Press, 1994, 18-25.

[CHM88] K.M. Chandy, J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, Inc., 1985.

[DID95] D. Das, D. Batory. "Prairie: A Rule Specification Framework for Query Optimizers." In *Proceedings of the 11th International Conference on Data Engineering*, 201-210, Taipei, March 1995.

[GAM95] E. Gamma, R.Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, Reading Massachusetts, 1995.

[GRD87] G. Graefe, D. J. Dewitt. "The EXODUS Optimizer Generator." In *Proceedings 1987 ACM SIGMOD International Conference on Management of Data*, San Francisco, May 1987. 387-394.

[GRM93] G. Graefe, W. McKenna. "The Volcano Optimizer Generator: Extensibility and Efficient Search." In *Proceeding of the 12th International Conference on Data Engineering*, 1993, 209-218.

[LAI83] J. E. Laird. *Universal Subgoaling*, Ph.D. dissertation, Carnegie-Mellon University, 1983.

[LAI87] J.E. Laird, A. Newell, and P.S. Rosenbloom, "SOAR: An Architecture for General Intelligence." *Artificial Intelligence*, Elsevier Science Publishers B.V. North-Holland, 1987, 1-64.

[MCC76] T. McCabe, "A Complexity Measure." *IEEE Transactions on Software Engineering*, December 1976, 308-320.

[MCD93] J. McDermott. "R1("XCON") at age 12: Lessons for an elmentary school achiver." Arrtificial Intelligenc, (59):1993, 241-247.

[KAB94] N. Kabra, D. J. Dewitt. "Opt++: An Object-Oriented Implementation for Extensible Database Query Optimization." Unpublished paper located in the SHORE papers home page, http://www.cs.wisc.edu/shore/shore.papers.html.

[MIR90] D.P. Miranker, D.Brant, B.J. Lofaso Jr., and D. Gadbois. "On the Performance of Lazy Matching in Production Systems." In *Proceedings of the 1990 National Conference on Artificial Intelligence*, AAAI, July 1990, 685-692.

[OBE95] L Obermeyer, D.P. Miranker, D. Brant. "Selective Indexing Speeds Production Systems." In *Proceedings of the 7th International Conference on Tools with Artificial Intelligence*, 1995.

[OBE96] L. Obermeyer, D.P. Miranker, “An Overview of the VenusDB Active Multidatabase System.” In the *Proceedings of of the International Symposium on Cooperative Database Systems for Advanced Applications*. Kyoto, Japan, December 1996.

[OBE97] L. Obermeyer, D.P. Miranker. “Evaluating Triggers Using Decision Trees.” To appear in CIKM 97, Las Vegas, Nevada, November 1997.

[OZM96] M. Tamer Özsu, Adriana Munoz, Duana Szafron. “An Extensible Query Optimizer for an Objectbase Management System.” In *Proceedings of the 4th International Conference on Information and Knowledge Management*, 1995, 188-196.

[PAC94] Pachet, F. ed. *Proceedings of the OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems* (EOOPS), Technical Report LAFORIA 94/24. Laboratoire Formes et Intelligence Artifcielle, Institut Blaise Pascal. Dec. 1994.

[PHH92] H. Pirahesh, J. M. Hellerstein, and W. Hasan. “Extensible/Rule Based Query Rewrite Optimization in Starburst.” In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, 39-48, San Diego, California, June 1992.

[SES97] P.Sesshadri, M. Livny and R.Ramakrishnan. The Case for Enhanced Abstract Data Types Praveen Seshadri, VLDB 97.

[STO92] M. Stonebraker. “The Integration of Rule Systems and Database Systems.” *IEEE Transactions on Knowledge and Data Engineering*, 415-423, October 1992.

[WAN92] Y-W Wang and E. Hanson. “A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions.” In *Proceedings of the Eighth International Conference on Data Engineering*, 1992.

[WAR96] L.B. Warshaw, D. P. Miranker. “A Case Study of Venus and a Declarative Bases for Rule Modules.” In *Proceedings of the 5th Conference on Information and Knowledge Management*, November 1996.