Synthesizing Rule Sets for Query Optimizers from Components*

Dinesh Das† Don Batory
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712–1188
{ddas,batory}@cs.utexas.edu

Abstract

Query optimizers are complex subsystems of database management systems. Modifying query optimizers to admit new algorithms or storage structures is quite difficult, but partly alleviated by extensible approaches to optimizer construction. Rule-based optimizers are a step in that direction, but from our experience, the rule sets of such optimizers are rather monolithic and brittle. Conceptually minor changes often require wholesale modifications to a rule set. Consequently, much can be done to improve the extensibility of rule-based optimizers.

As a remedy, we present a tool called *Prairie* that is based on an algebra of *layered optimizers*. This algebra naturally leads to a building-blocks approach to rule-set construction. Defining customized rule sets and evolving previously defined rule sets is accomplished by composing building-blocks. We explain an implementation of Prairie and present experimental results that show how classical relational optimizers can be synthesized from building-blocks, and that the efficiency of query optimization is not sacrificed.

1 Introduction

Query optimization [11, 14, 19] is a fundamental part of database systems. It is the process of generating an efficient access plan (i.e., an execution strategy) for a database query. There are three aspects that define and influence query optimization: the search space, the cost model, and the search strategy.

The *search space* is the set of logically equivalent access plans that can be used to evaluate a query. All plans in a query's search space return the same result; however, some plans are more efficient than others. The *cost model* assigns a cost to each plan in the search space. The cost of a plan is an estimate of the resources used when the plan is executed; the lower the cost, the better the plan. The *search strategy* is a specification of which plans in the search space are to be examined.

Traditionally, query optimizers have been built as monolithic subsystems of database management systems (DBMSs). This simply reflects the fact that traditional database systems are themselves monolithic: the algorithms that are used for storing and retrieving data are hard-wired and are

*This research was supported in part by grants from The University of Texas Applied Research Laboratories, Schlumberger, and Digital Equipment Corporation.

†Current address: Oracle Corporation, 500 Oracle Parkway, Box 659413, Redwood City, CA 94065, USA

difficult to change. Customizability of such optimizers is almost impossible without an enormous effort by the database implementor (DBI). This difficulty has lead to the development of rule-based query optimizers, whose primary purpose is to achieve query optimizer extensibility [9, 10, 12, 13]. The basic idea is that the actions of a query optimizer are defined as a set of rewrite rules that progressively optimize expressions which define how queries can be evaluated.

From our experience, the rule sets of such optimizers are rather brittle. If a new feature (e.g., retrieval or join algorithm) is to be added to an optimizer, it is not quite a simple matter of adding one or more rules. For example, in the Volcano rule-based optimizer [10], rule implementations are not encapsulated. Consequently, conceptually simple modifications to a rule set often require significant effort including new function definitions to characterize the new feature.

Modifying rule sets is actually quite important for systems like Starburst [15], Open OODB [5], or P2 [3, 4] where adding a new feature to a DBMS involves plugging a component into the DBMS itself.[1] For obvious reasons, updating a DBMS in such a manner should not require DBIs to hack rule sets in order to make the resulting DBMS execute correctly; rule sets should be automatically updated as a consequence of the addition or removal of a component. Thus, it is imperative that better ways to structure rule sets be found that make rule sets themselves more easily extensible and automatically updatable.

In this paper, we describe extensions to the Prairie tool and rule specification language [7, 8] that meet this demand for rule-set extensibility. Monolithic rule sets can be modularized as compositions of primitive rule sets. By encapsulating primitive rule sets into components called *layers*, and by composing layers in different ways, we are able to generate large families of customized rule sets, where each family member targets a different DBMS implementation. We describe the process of rule set generation as one of compacting and optimizing a layered specification. Experimental results are presented that show that a layered optimizer can be just as efficient as a monolithic one. We conclude by discussing related work.

2 Prairie: A Rule Specification Language

Prairie is a front-end to the Volcano [10] optimizer generator. Prairie is similar to Starburst [15] in using a rule-based approach to optimizer design in that both rely on a model of rewrite rules with corresponding actions. Prairie is different than Starburst and Volcano in using a building-blocks methodology (as described in this paper). We have based our work on Volcano because it is freely available and because of Prairie's demonstrated usefulness [8] in re-engineering large Volcano rule sets such as the Open OODB optimizer [5].

As described in [8], Prairie provides three key features that simplify the effort in writing rules. First, **abstractions** (like rules and actions) capture the design and semantics of an optimizer. This has the advantage that changes to an optimizer consists of changing the *implementation* of its abstractions, not the abstractions themselves. Second, the **extensibility** (i.e., modifying the optimizer when its target DBMS changes) of Prairie optimizers is facilitated not only by the abstractions, but also by the uniform treatment of the rules and actions.[2] Third, good **performance** of Prairie optimizers

[1] Components are called *extensions* in Starburst, *policies* in Open OODB, and *layers* in P2.

[2] In Volcano, for example, there are two different kinds of rules: implicit and explicit. Implicit rules are inferred by the Volcano model; explicit rules are those that DBIs must define. The distinction of implicit vs. explicit rules can be a source

(i.e., the time to optimize a query) is ensured by having *efficient* implementations of the abstractions. Experimental results demonstrating these goals were achieved are presented in [7, 8].

This paper describes a fourth important enhancement to the Prairie model, namely to generate **reconfigurable** rule sets from components. That is, rule sets are modularized as *building-blocks* that can be arranged in various ways to construct a customized rule set. These building-blocks encapsulate primitive implementations of basic optimizer abstractions. This means that the rule set of an optimizer can be modified quickly, simply by changing the composition of building-blocks that define the optimizer's rule set. The ability to generate reconfigurable rule sets almost "on-the-fly" not only means that Prairie can be used to construct highly customized rule sets, but also that it can be used to build "throw-away" optimizers for one-of-a-kind applications (something that is not possible with monolithic optimizers) [3, 4].

In [7, 8], we discussed how Prairie achieves the first three goals. In this paper, we describe how an extension to Prairie achieves the fourth goal: namely, how reconfigurable rule sets can be constructed from building-blocks. Before doing this, however, we begin with a brief presentation of the concepts and notation employed by Prairie.

2.1 Notation and Assumptions

Relations and Streams. Relations reside on disk and are denoted by R_i. A *stream* is a sequence of tuples and is the result of a computation on one or more streams or relations. Streams can be *named* (denoted by S_i) or *unnamed*.

Database Operators. An *operator* is a computation on one or more streams or relations. There are two types of operators in Prairie. *Abstract* (or conceptual) *operators* are computations on streams or relations; they are denoted by all capital letters (e.g., JOIN). *Algorithms* are concrete implementations of abstract operators; they are represented in lower case with the first letter capitalized (e.g., Merge_join). There can be, and usually are, several algorithms for a particular operator.

Operator Trees. An *operator tree* is a rooted tree whose non-leaf, or interior, nodes are abstract operators or algorithms, and whose leaf nodes are relations. The children of an interior node in an operator tree are the inputs (i.e., streams or relations) of the node. Algebraically, operator trees are compositions of database operators. Thus, we will also call operator trees *expressions*; both terms will be used interchangeably.

EXAMPLE 1 A simple expression is SORT (JOIN (RET (R_1), RET (R_2))). Tuples of relations R_1 and R_2 are first RETrieved, and then JOINed, and finally SORTed resulting in a stream sorted on a specific attribute. □

Descriptors. A *property* of a node is a (DBI-defined) variable that contains information used by an optimizer. For example, the tuple order of a stream or the number of tuples of a relation or stream are properties. An *annotation* is a ⟨*property, value*⟩ pair that is assigned to a node. A *descriptor* is a list of annotations that describes a node of an operator tree; every node has its own descriptor. The

of confusion, particularly when debugging rule sets. Prairie, in contrast, has no implicit rules.

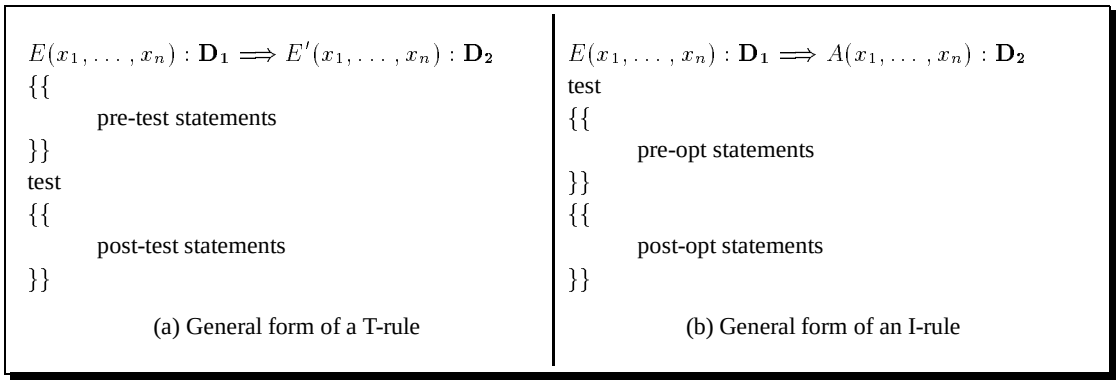


Figure 1: Prairie rewrite rules

following notations will be useful in our subsequent discussions. If S_i is a stream, then $\mathbf{D_i}$ is its descriptor. Also, let E be an expression and let $\mathbf{D}$ be its descriptor. We will write this as $E : \mathbf{D}$.

EXAMPLE 2 The expression of Example 1 that is annotated with descriptors is:

SORT(JOIN(RET(R_1) : $\mathbf{D_3}$, RET(R_2) : $\mathbf{D_4}$) : $\mathbf{D_5}$) : $\mathbf{D_6}$

$\mathbf{D_3}$ and $\mathbf{D_4}$ are the descriptors of the two RETrievals respectively, $\mathbf{D_5}$ is the descriptor of the JOIN, and $\mathbf{D_6}$ is the descriptor of the SORT. □

Access Plans. An *access plan* is an operator tree in which all interior nodes are algorithms.

EXAMPLE 3 A possible access plan for the expression in Example 1 is:

Merge_sort(Nested_loops(File_scan(R_1), File_scan(R_2)))

Relations R_1 and R_2 are each retrieved using the File_scan algorithm, joined using Nested_loops, and finally sorted using Merge_sort. □

2.2 Rules in Prairie

There are two types of algebraic transformations (or *rewrite rules*) in Prairie: T-rules ("transformation rules") and I-rules ("implementation rules"). Each rule transforms an expression into another conditionally; the transformation also results in a mapping of descriptors between expressions. T-rules and I-rules are defined in the following sections.

2.2.1 Transformation Rules

Transformation rules, or T-rules for short, define equivalences among pairs of expressions; they define mappings from one operator tree to another. Let E and E' be expressions that involve only abstract operators. Figure 1(a) shows the general form of a T-rule. The actions of a T-rule define the equivalences between the descriptors of nodes of the original operator tree E with the nodes of the

output tree E'; these actions consist of a series of (C or C++) assignment statements that define the descriptors of E'.

A *test* is used to determine if the transformations of a T-rule are applicable. Purely as an optimization, it is usually the case that not all statements in a T-rule's actions need to be executed prior to a T-rule's test. For this reason, the actions of a T-rule are split into two groups; those that need to be executed prior to the T-rule's test, and those that can be executed after a successful test. These groups of statements comprise, respectively, the *pre-test* and *post-test* statements of a T-rule.

2.2.2 Implementation Rules

Implementation rules, or I-rules for short, define equivalences between expressions and their implementing algorithms. Let E be an expression and A be an algorithm that implements E. The general form of an I-rule is shown in Figure 1(b).

The actions associated with an I-rule are defined in three parts. The first part, or *test*, is a boolean expression whose value determines whether or not the rule can be applied.

The second part, or *pre-opt statements*, is a set of descriptor assignment statements that are executed only if the test is true and *before* any of the inputs x_i of E are optimized. Additional properties of nodes are usually assigned in the pre-opt section. This is necessary before any of the nodes on the right side can be optimized.

The third part, or *post-opt statements*, is a set of descriptor assignment statements that are executed *after* all x_i are optimized. Normally, the post-opt statements compute cost properties that can only be determined once the inputs to the algorithm are completely optimized and their costs known.

3 Layered Rule-Based Optimizers

3.1 Layers

In the Prairie framework described in Section 2, optimizers are specified using rules (T-rules and I-rules). The rule engine treats all rules as belonging to a single set, so at any given stage, the rule engine transforms an expression using all applicable rules. Rule conditions determine the search space to be generated. A shortcoming of this model is that the behavior of the optimizer can be changed only by the modification of individual rules; there is no simple way to selectively modify a *set* of rules.

This section describes a building-blocks approach to the construction of rule sets for rule-based optimizers using Prairie. The goal is to generate families of rule sets quickly and automatically where each family member corresponds to a DBMS with a unique set of features (e.g., retrieval and join algorithms). We discuss the model, together with a few simple examples, and describe how efficient implementations can be quickly generated from primitive Prairie specifications.

Rules that implement a basic feature of database system construction (e.g., relation distribution, relation replication, relation implementation) are encapsulated in components called *layers*. Layers can either be defined by a DBI, or can exist in pre-defined component libraries. Each layer is a collection of T-rules and I-rules and has well-defined import and export interfaces that consist of database operators. The general form of a layer is shown in Figure 2(a). By applying rewrite rules, a layer translates an abstract expression consisting of *abstract* operators $\{O_1, \ldots, O_n\}$ to a set of concrete

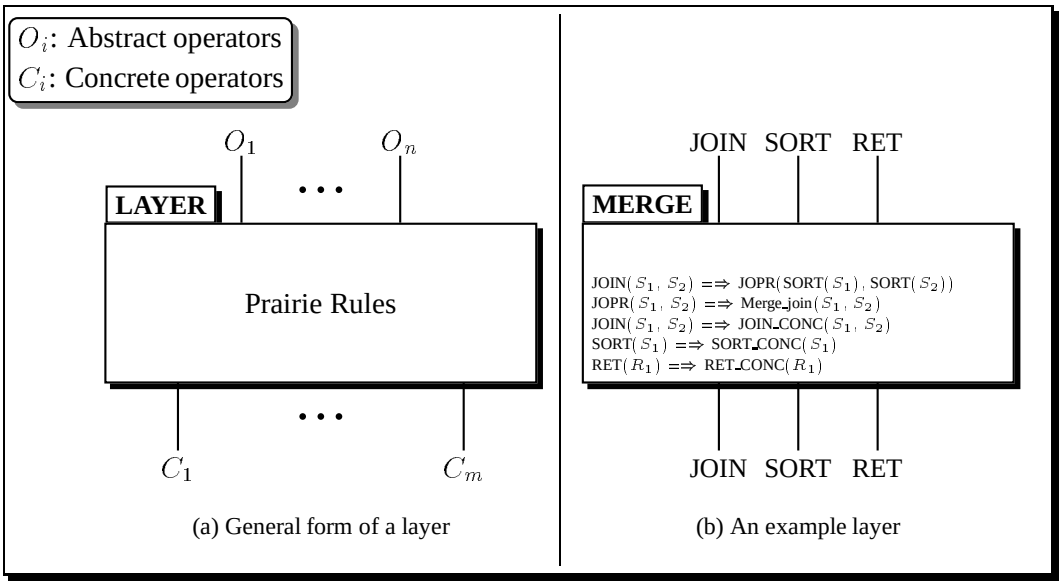


Figure 2: General form of a Prairie layer and an example

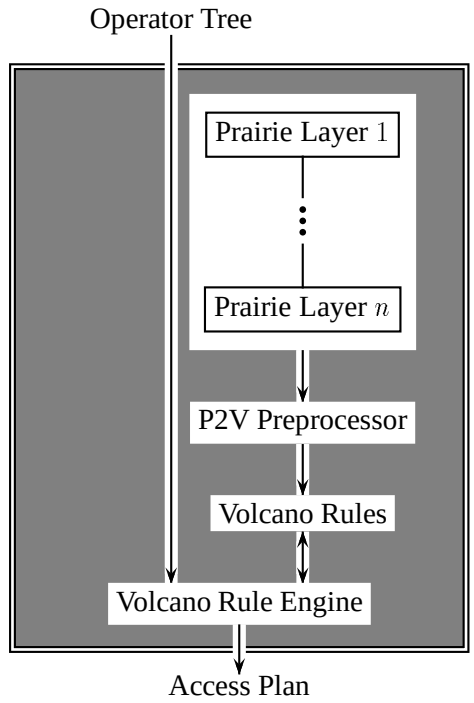
expressions, each consisting of one or more *concrete* operators $\{C_1, \ldots, C_m\}$ or algorithms. This represents a one-to-many mapping between expressions, and is typically the method used by an optimizer to construct its search space. The term concrete refers to the fact that they are obtained by transforming abstract operators through the use of rules; concrete operators of a layer can also be viewed as calls to abstract operators of lower layers.

Viewing complex rule sets as compositions of primitive layers is an example of the GenVoca paradigm of software generation [2]. In GenVoca, building-blocks of software systems are layers that import and export standardized interfaces; a layer transforms abstract programs (operator trees) that call operators of its export interface into more concrete programs (operator trees) that call operators of its import interface. Importing and exporting standardized interfaces enables layers to "snap" together like legos. Different compositions of building-blocks define different systems (or, in our case, different optimizer rule sets). A key feature of GenVoca is the use of *symmetric* layers; i.e., layers that export and import the same interface. Symmetric layers have the important feature that they can be composed in virtually arbitrary orders. In the case of Prairie, symmetry offers DBIs many ways to construct different rule sets using a small set of layers.[3]

To allow optimizer specifications using layers, the Prairie specification language of Section 2 was extended in two ways. First, rules can now be declared as belonging to a specific layer (a layer declaration demarcates rule definitions). Second, the rule set of an optimizer can be defined as a linear composition of layers. Layer compositions are described in more detail in the next section.

An example layer is shown in Figure 2(b). This layer, called **MERGE**, transforms three abstract operators, JOIN, SORT, and RET into one algorithm (Merge_join) and three concrete operators (JOIN, SORT, and RET). The **MERGE** layer consists of four T-rules and one I-rule. The purpose of the layer is to either transform the JOIN operator into the Merge_join algorithm, or to a concrete

[3]Not all the compositions of GenVoca layers are necessarily meaningful or correct. Methods for validating the consistency of compositions are discussed in [1].



(a) Schematic representation

%optimizer SEQUENTIAL [MERGE [SORT [RET]]]

%layer SEQUENTIAL
- **%trule** JOIN(S_1, S_2) ==>! JOIN(S_2, S_1)
- **%trule** JOIN(JOIN(S_1, S_2), S_3) ==> JOIN(S_1, JOIN(S_2, S_3))
- **%trule** JOIN(S_1, S_2) ==> JOIN_CONC(S_1, S_2)
- **%trule** SORT(S_1) ==> SORT_CONC(S_1)
- **%trule** RET(R_1) ==> RET_CONC(R_1)

%layer SORT
- **%irule** SORT(S_1) ==> Merge_sort(S_1)
- **%irule** SORT(S_1) ==> Null(S_1)
- **%trule** JOIN(S_1, S_2) ==> JOIN_CONC(S_1, S_2)
- **%trule** SORT(S_1) ==> SORT_CONC(S_1)
- **%trule** RET(R_1) ==> RET_CONC(R_1)

%layer MERGE
- **%trule** JOIN(S_1, S_2) ==> JOPR(SORT(S_1), SORT(S_2))
- **%irule** JOPR(S_1, S_2) ==> Merge_join(S_1, S_2)
- **%trule** JOIN(S_1, S_2) ==> JOIN_CONC(S_1, S_2)
- **%trule** SORT(S_1) ==> SORT_CONC(S_1)
- **%trule** RET(R_1) ==> RET_CONC(R_1)

%layer DISTRIBUTION
- **%trule** JOIN(S_1, S_2) ==> JOIN_CONC(XFER(S_1), XFER(S_2))
- **%irule** XFER(S_1) ==> Ship(S_1)
- **%irule** XFER(S_1) ==> Null(S_1)
- **%trule** JOIN(S_1, S_2) ==> JOIN_CONC(S_1, S_2)
- **%trule** SORT(S_1) ==> SORT_CONC(S_1)
- **%trule** RET(R_1) ==> RET_CONC(R_1)

%layer REPLICATION
- **%trule** RET(R_1) ==> RET_CONC(R_{11})
- **%trule** RET(R_1) ==> RET_CONC(R_{12})

%layer RET
- **%irule** RET(R_1) ==> File_scan(R_1)

(b) Syntactic specification. Rule actions are omitted for clarity.

Figure 3: The Prairie layered optimizer paradigm

JOIN operator that will be transformed into an algorithm by another (lower) layer.

Note that **MERGE** is *symmetric*, i.e., it exports the same set of operators that it imports. To distinguish exported operators from imported operators in rules, Prairie requires a DBI to append "_CONC" to an operator to refer to an imported operator. Thus, "JOIN" refers to the exported join operator, and "JOIN_CONC" refers to the imported join operator.

3.2 Composing Layers

As mentioned earlier, Prairie is a front-end to the Volcano search engine [10]. DBIs specify high-level rule sets in Prairie, and a preprocessor (described below) compacts and optimizes this rule set into a (low-level) form that can be efficiently processed by Volcano. The extension that we have made to Prairie is to offer an alternative to specifying monolithic rule sets. Instead, complex Prairie rule sets can be generated from a linear composition of predefined layers that implement primitive features common to many DBMSs.

Figure 3(a) presents a schematic overview of Prairie. A linear[4] composition of layers is fed into

[4] A more general model of composition (one that is advocated by GenVoca) is to allow nonlinear compositions of layers. Linear compositions are sufficient for most applications that we have encountered for generating rule sets for query optimizers.

the P2V (Prairie-to-Volcano) preprocessor, and a monolithic (and optimized) Volcano rule set is produced. Operator trees are then optimized by this Volcano rule set.

The Prairie syntax for specifying rules in individual layers and specifying layer compositions is shown in Figure 3(b). The composition shown in this figure, for example, represents an optimizer with the **SEQUENTIAL**, **MERGE**, **SORT**, and **RET** layers stacked in that order. (The semantics of these layers will be discussed shortly; for now, we briefly describe their operational functionality.)

Given an expression to optimize, the rule engine applies rules to the expression in the order that layers are composed. Thus, rules in a layer are applied to an expression until no further rule applications are possible; the rule engine then applies rules in the next layer, and so on. Thus, layer composition defines a sequence of rule sets to be applied to an expression, and it is this sequence that defines the search space of the optimizer.

Although the semantics of layer composition can be understood as a pipeline of optimizations, optimizing expressions in phases is not the most efficient implementation of layered rule sets. (Layered specifications of rule sets tend to have many rules, and typically the greater the number of rules in a rule set, the longer it takes to optimize an expression.) To generate high-performance rule sets in Volcano requires layered specifications to be reduced (i.e., optimized and compacted) into monolithic rule sets that are suitable for Volcano execution. This is the role of the P2V preprocessor. As currently implemented, it has four key responsibilities:

- Establishing the correspondence between the various concepts of Prairie to similar ones in Volcano. Specifically, this means that the P2V preprocessor must translate relations, streams, operators, algorithms, operator trees, access plans, and descriptors into Volcano format.

- Translating T-rules into Volcano transformation rules. This includes translating the actions (tests and property transformations). Note that because descriptors are translated into Volcano property structures, Prairie rule actions that reference descriptor properties must also be translated into Volcano rule actions that reference the appropriate Volcano structures.

- Translating I-rules into Volcano implementation rules. As above, this includes translating an I-rule's actions into Volcano format.

- Generating a *compact* Volcano rule set from a Prairie specification. This means that the P2V preprocessor transforms a layered rule specification into a monolithic rule set, removes unused rules, and consolidates rules that generate a transitive closure of operator tree transformations. This step is not necessary for the correct generation of Volcano specifications; it is, however, a means of generating smaller rule sets, and consequently, faster optimizers.

Details are found in [7].

A question that might arise is the faithfulness of our layer compaction algorithm; that is, whether compacted layered optimizers have the same search space as a monolithic hand-coded optimizer. While it is difficult to prove this in a formal sense (since the rule actions for layered optimizers are different from those in a monolithic optimizer because the P2V preprocessor adds additional statements to preserve the hierarchical ordering of layers), all the layered optimizers that we constructed resulted in exactly the same rule set as the corresponding monolithic optimizer. Moreover, although the corresponding rule actions are not exactly the same (since the P2V preprocessor adds some extra

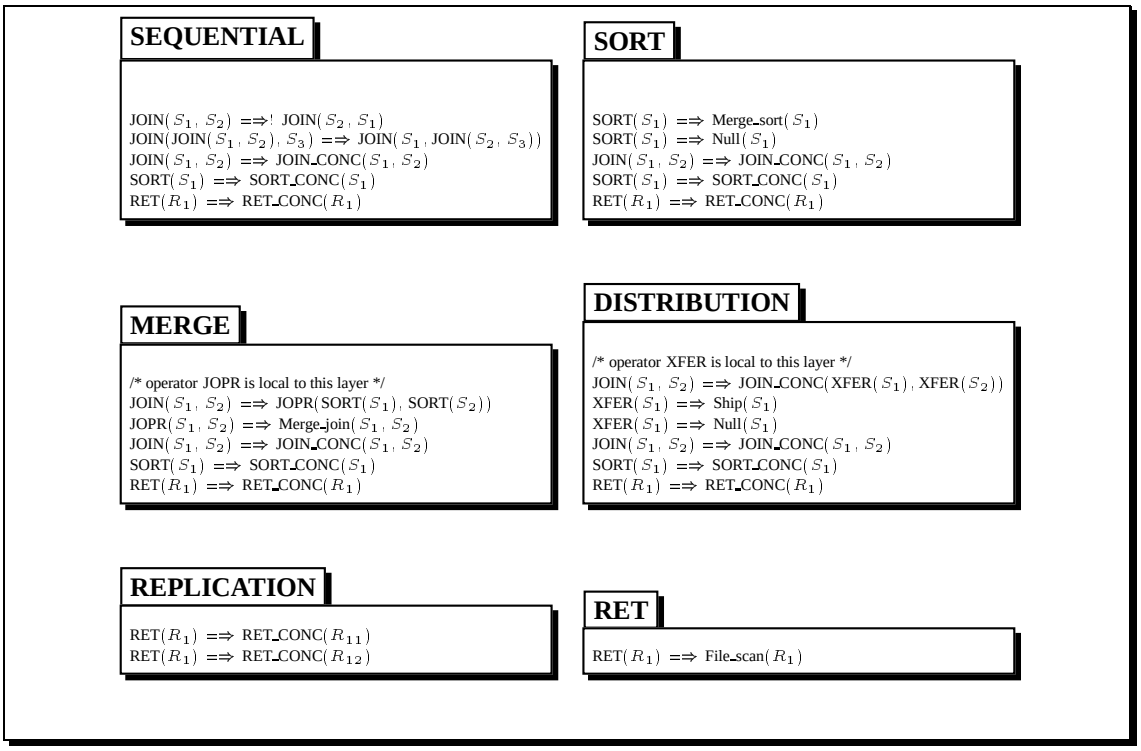


Figure 4: Example layers. For clarity, all rule actions are omitted. Local operators (i.e., those neither exported nor imported) are identified in a comment.

statements) in the two approaches, the number of expressions in the search spaces were exactly the same for all the queries that we optimized using the layered optimizers and their monolithic counterparts. This lends credence to the hypothesis that a properly specified layered optimizer is semantically equivalent to a monolithic (non-layered) specification. In other words, we believe that *any* monolithic optimizer can be expressed as a composition of layered rule sets. The question is whether these layers are general enough to be reusable. It is always possible to start from a monolithic rule set and decide which rules can be encapsulated as a single layer or which rules already exist in a predefined layer. Once a DBI has decided on specific layer definition, the layer compaction algorithm then compacts the layered rule set which can then be compared to the original monolithic rule set to verify that it specifies an equivalent rule set. Our experiences suggest that it is possible to design general layers that, when composed, yield practical monolithic optimizers; any differences between the layered specification and a desired optimizer are ignorable.

3.3 Examples of Layered Optimizers

This section describes several variations of traditional relational optimizers constructed using layers.

3.3.1 Example Layers

Some examples of layers in a Prairie specification are shown in Figure 4.[5] These layers specify transformations typically found in traditional relational databases. There are six different layers shown, **SEQUENTIAL**, **SORT**, **MERGE**, **RET**, **DISTRIBUTION**, and **REPLICATION**.

The **SEQUENTIAL** layer encapsulates transformations that are typically found in centralized optimizers. Join commutativity and associativity T-rules are included in this layer. The remaining rules simply transform the abstract operators into their concrete counterparts, to be transformed by lower layers.

The **SORT** layer encapsulates implementations of the SORT operator. In Figure 4, the SORT operator is transformed to either the Merge_sort or the Null algorithm. Other sort algorithms can either be introduced in this or other **SORT** layers. The remaining rules transform abstract operators into concrete operators.

The **MERGE** layer transforms the JOIN operator into the Merge_join algorithm. Other join algorithms can either be encapsulated in the **MERGE** layer, or in a separate layer.

The **DISTRIBUTION** layer encapsulates the distribution of relations in distributed databases. It transforms the JOIN operator such that if its inputs are located at different sites, they are first transfered to the home site (i.e., the site where the JOIN was issued) before the join is performed. The XFER operator denotes the transfer of streams between sites; one algorithm that implements the XFER operator is Ship. The Ship algorithm here is assumed to be a block transfer of streams (as in R* [6]); other transfer strategies (e.g., tuple-at-a-time) could be defined in this or other layers encapsulating rewrites of distributed query processing.

The **REPLICATION** layer models replicated databases. Its imported interface is a RET operator that simulates a centralized, non-replicated database. That is, it gives the illusion of a single relation for each relation in the database. The **REPLICATION** layer translates a relation reference into a reference to one of the physical replicas of the relation. (In Figure 4, we assume each relation is replicated twice.)

The **RET** layer transforms a RET operator into the File_scan algorithm. Note that there are no other rules transforming abstract operators into other concrete operators. This means that the **RET** layer is not symmetric, i.e., it exports the RET operator, but doesn't import any concrete operator. This, in turn, implies that the **RET** layer, as defined, is always the last in a layer composition.

In the following sections, we will show how these layers can be used to construct rule sets for simple optimizers. As mentioned earlier, symmetric layers admit more composition possibilities, since the exported and imported interfaces are the same.

3.3.2 An Optimizer for a Centralized Database

An optimizer for a centralized database is shown in Figure 5. It is formed by the composition **SEQUENTIAL** [**MERGE** [**SORT** [**RET**]]]. The **SEQUENTIAL** layer applies the join associativity and commutativity T-rules to a join expression. The joins of this expression are then transformed into the Merge_join algorithm by the **MERGE** layer. The **SORT** layer then transforms the SORT operator into the Merge_sort algorithm, and finally the **RET** layer transforms the RET operator into the File_scan algorithm. An example of such a transformation is shown in Figure 5. (Horizontal

[5] For clarity, we omit all rule actions in the descriptions of these layers.

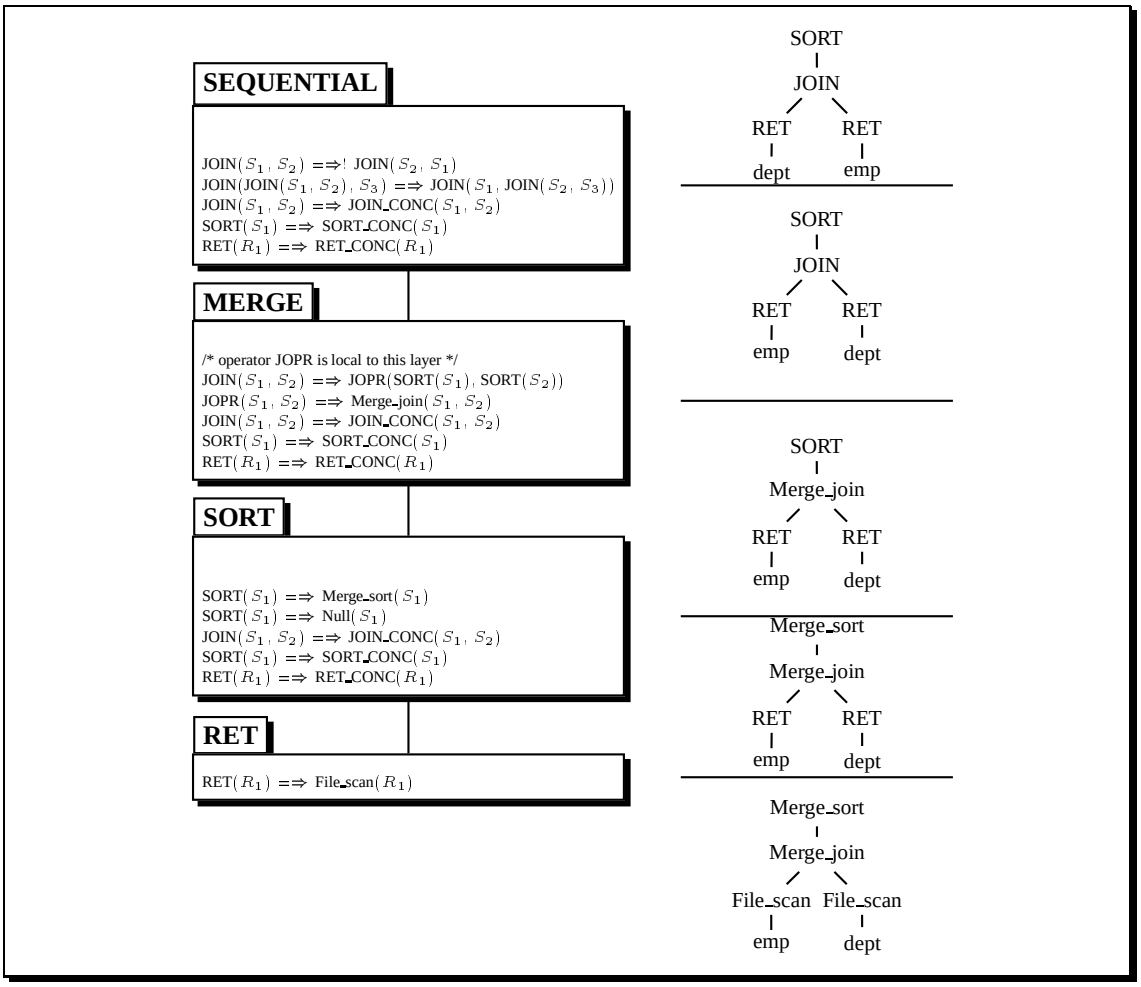


Figure 5: An optimizer for a centralized database and an example transformation

lines separate the input and output operator trees for each layer.) Note that each of the layers has "dummy" T-rules that transform abstract operators into concrete operators. Thus, for instance, the **SEQUENTIAL** layer can either apply the join commutativity rule to a join expression, or pass the expression unchanged to the **MERGE** layer. The example transformation shown in Figure 5 is, thus, *one* of many in the search space produced by the centralized layered optimizer.

Note that some operators (e.g., JOIN_CONC and SORT_CONC in the **SORT** layer) have no implementations in lower layers. The rules that generate these operators are discarded by the P2V preprocessor for reasons to be explained later; this optimization or rule-set simplification is described in Section 3.4.

3.3.3 An Optimizer for a Distributed Database System

A distributed database consists of relations at various sites. An optimizer for such a database must ensure that the required relations and streams are transfered to a common site before a join operation can be performed. A possible layer composition that captures the semantics of a rule set for a

distributed query optimizer is **SEQUENTIAL** [**DISTRIBUTION** [**MERGE** [**SORT** [**RET**]]]]. In other words, it is obtained by adding the **DISTRIBUTION** layer to the centralized optimizer in Figure 5. The **SEQUENTIAL** layer applies the join associativity and commutativity T-rules to a join expression. The **DISTRIBUTION** layer then ensures that join streams are shipped to the join site by using the Ship algorithm. The JOIN operator is then transformed into the Merge_join algorithm by the **MERGE** layer. The **SORT** layer then transforms the SORT operator into the Merge_sort algorithm, and finally the **RET** layer transforms the RET operator into the File_scan algorithm.

3.3.4 An Optimizer for a Replicated Database

An optimizer specification for a replicated DBMS is obtained from the centralized optimizer in Figure 5 by inserting the **REPLICATION** layer to form the composition **SEQUENTIAL** [**MERGE** [**SORT** [**REPLICATION** [**RET**]]]]. The **SEQUENTIAL** layer applies the join associativity and commutativity T-rules to a join expression. The **MERGE** layer transforms the JOIN operator into the Merge_join algorithm. The **SORT** layer then transforms the SORT operator into the Merge_sort algorithm. The **REPLICATION** layer transforms references to logical relations into their physical replicas. Finally, the **RET** layer transforms the RET operator into the File_scan algorithm.

3.4 Compacting Layered Optimizers

In the previous sections, we described how layers can be used to define small rule sets for optimizers, and how these layers can be composed to construct an optimizer. However, as seen from the example layers in Section 3.3, even simple layered optimizers can consist of a large number of rules. A naive implementation of such a specification can result in an inefficient optimizer if the implementation contains a large number of identity transformations (i.e., an operator transformed into its concrete counterpart). In this section, we discuss how the P2V preprocessor can be used to compact layered specifications of rule sets to obtain a monolithic rule set.

Layers have two primary goals: to translate abstract operators into concrete ones, and to define a hierarchy of rules (i.e., to establish rule precedence). Any compaction of layers has to preserve the semantics of these two goals. The P2V preprocessor accomplishes both of these goals. Broadly speaking, there are two responsibilities of the P2V preprocessor in compacting layers. The first is to compact the rules themselves, and the second is to ensure that the compaction of rule actions generates a semantically equivalent rule set. Below, we discuss these two steps in greater detail.

The translation of abstract operators into concrete ones by a layer implies that there is a one-to-one correspondence between a concrete operator of one layer and an abstract operator of the layer immediately below it. Thus, in the centralized optimizer of Figure 5, the JOIN_CONC operator in the **SEQUENTIAL** layer corresponds to the JOIN operator in the **MERGE** layer. Once this correspondence is established, the P2V preprocessor can use the rule compaction techniques described in [7] to combine all the layers together into a single, monolithic rule specification. The complexity of the compaction algorithm lies primarily in the process of combining rule actions; for more details, see [7]. For the centralized optimizer shown in Figure 5, the compaction process results in the layer shown in Figure 6. It is interesting to note that the monolithic rule set obtained by layer compaction is the same[6] (except for rule actions; see [7]) as one that might have been hand-written by a DBI; the

[6]Note, especially, that rules (e.g., the JOIN to JOIN_CONC transformation in the **MERGE** layer) that simply transform

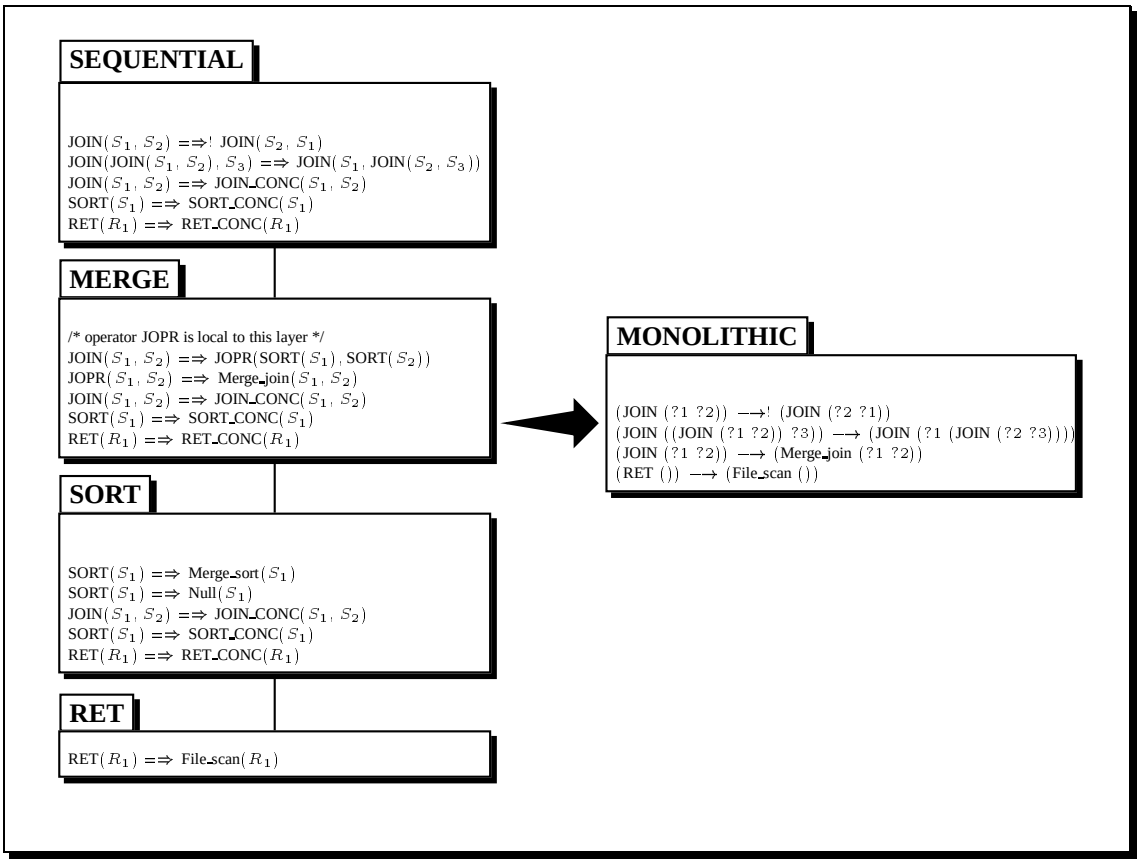


Figure 6: Compacting the layered centralized Prairie rule set in Figure 5. Rule actions are again omitted for clarity.

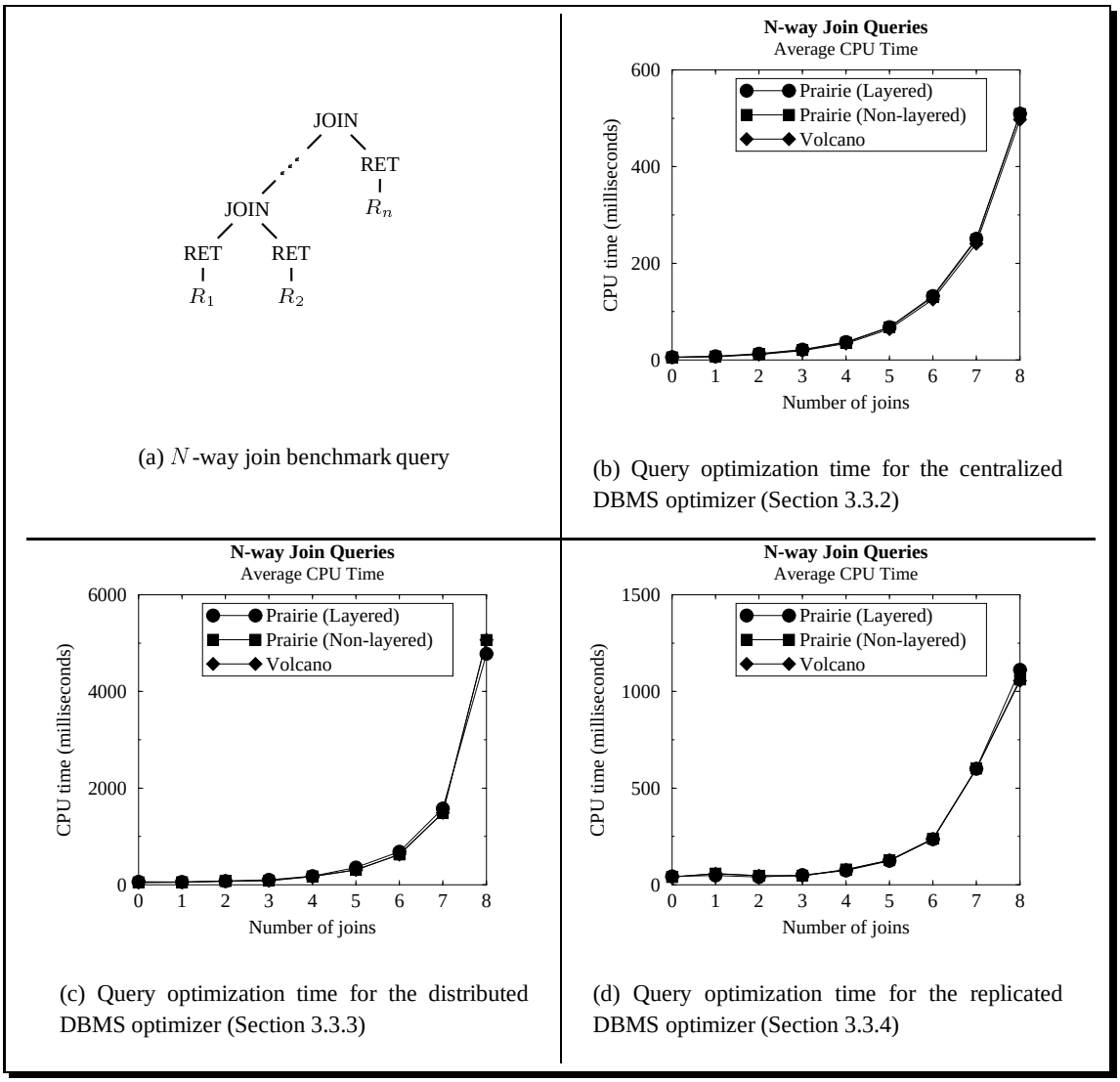
layered specification, however, affords more degrees of extensibility and easier customizability.

Another aspect of a layered optimizer specification that must be preserved by compaction is that of the hierarchical nature of rules. That is, the semantics of layered rule sets (rules in a layer applied before rules in a lower layer) must be maintained when the layers are compacted. This is accomplished by introducing an annotation for each node in an expression that tracks the last layer that transformed the node. Rules in a layer above this last layer are not applied to the node again. Thus, in the centralized optimizer of Figure 5, an expression that has been transformed by the **MERGE** layer cannot be transformed by the **SEQUENTIAL** layer subsequently.

4 Benchmarking Layered Optimizers

In the previous section, we described the compaction of layered rule set specifications to generate monolithic rule sets. The question that naturally arises is whether this method of rule set construction (layered specification followed by compaction) results in slower optimization times as compared to a monolithic (i.e., non-layered) optimizer. In this section, we present preliminary experimental results

abstract operators to concrete operators are discarded as a direct consequence of the compaction process.



(a) N-way join benchmark query

(b) Query optimization time for the centralized DBMS optimizer (Section 3.3.2)

(c) Query optimization time for the distributed DBMS optimizer (Section 3.3.3)

(d) Query optimization time for the replicated DBMS optimizer (Section 3.3.4)

Figure 7: Benchmarking layered optimizers

that demonstrate that the efficiency of an optimizer is not sacrificed.

Consider the layered centralized optimizer shown in Figure 5. As shown in Figure 6, this can be compacted to a monolithic rule set. To verify that the efficiency of optimizers generated from layered specifications is not sacrificed, we conducted experiments involving optimizer specifications for centralized, distributed, and replicated DBMSs. Each optimizer was specified in three ways: layered Prairie, non-layered Prairie, and hand-coded Volcano. Each resulting optimizer was benchmarked using a set of randomly generated queries. The results are reported below.

The experiments consisted of optimizing left-deep N-way join operator trees, for varying values of N, as shown in Figure 7(a). Random queries were generated using a uniform random number generator. The set of relations, along with their attributes, cardinalities, and record widths was fixed. The order of relations in the left-deep tree was varied. Each join had a single equijoin predicate, with the two join attributes chosen at random from the set of eligible attributes of the outer and inner

streams. For the distributed DBMS optimizer, we simulated a database with four sites; each relation was randomly assigned to a site. For the replicated DBMS, each relation had two replicas; each replica was assumed to be sorted on different attributes.

For each value N of the number of joins, we generated 10 different queries,[7] and optimized each query using optimizers generated from three different specifications: layered Prairie, non-layered Prairie, and hand-coded Volcano. The run times were measured using the GNU `time` command, and averaged over the 10 queries to generate the per-query optimization time. Each point in the graph, thus, represents the average CPU time for optimizing the queries. All experiments were performed on a lightly loaded DECstation 5000/200 running Ultrix 4.2.

The optimization times for the layered Prairie, non-layered Prairie, and hand-coded Volcano optimizers are shown in Figures 7(b), 7(c), and 7(d) for the centralized, distributed, and replicated DBMS optimizers described earlier, respectively. In each case, we can see that the three specifications are virtually equally efficient, supporting the hypothesis that compacted layered rule sets need not sacrifice any performance. More experiments are needed, however, to verify that the performance is good when scaled to compositions of a *large* number of layers (i.e., compositions of 10-20 layers [4]).

Another metric useful in measuring the benefits of layered rule set specifications is the ease with which they can be tailored to different applications. In this case, the layered approach is very helpful because it helps a DBI to clearly see the effects of any change on the search space of an optimizer. For instance, the Open OODB optimizer [5] results in an extremely large search space for certain queries. If it were implemented using layers, then the DBI could more easily experiment with adding new rules and layers in various configurations. It is in this respect that we believe that the layered approach will yield the most productivity gains.

5 Related Work

Optimizer design and implementation using pre-defined building-blocks has been proposed by other researchers. All follow a rule-based paradigm. In this section, we briefly review some of these approaches. With notable exceptions (e.g., Starburst and Open OODB), the main problem with these approaches is that they are paper designs, so it is not immediately clear how well they might work. Moreover, from our work, it would seem to be important in all such building-block approaches that there should be a compiler that can generate efficient, compact rule sets from specifications of component compositions. None of the proposals discussed in this section (except, again, for Starburst and Open OODB) describe how that is done, or even if it is possible.

Starburst [15] is an optimizer that uses functional rules to allow DBIs to specify transformations of user-queries. It has two phases. The query rewrite phase generates a search space for the optimizer based on heuristic, non-cost-based transformations. This is followed by the optimization phase that uses cost-based rules to transform operator trees into access plans. This suggests a form of component-based optimization, but has the drawback that the goals, number, and order of the two phases are not changeable by the DBI. Moreover, there are limitations on the types of rules that can occur in each phase; for example, cost computations are not permitted in the rewrite phase.

[7] We chose 10 instead of a larger number since the average run time was not substantially different for more queries.

Starburst and Open OODB [5] do not provide a framework for DBI-defined components that can be composed to generate a rule set. The result is that rule sets in current optimizers defined using these models are monolithic consisting of a large number of rules. For example, the Open OODB optimizer [5] contains 26 rewrite rules without any mechanism to define more manageable rule set modularizations. The result is a cumbersome rule specification scheme that is highly error-prone and hard to modify in a streamlined manner. Without a mechanism to add or remove sets of rules and automatically generate new rule sets quickly, such systems are extremely limited in the scope and speed of customization.

Sciore and Sieg [18] describe an optimizer generator model that allows a DBI to construct a rule-based optimizer using *modules*. Each module consists of *term rewrite rules* (with conditions) to transform *terms* in a relational algebra. Each module has exported and imported interfaces which consist of terms.

Each module in Sciore and Sieg's framework is allowed to have rewrite rules in its own relational algebra. A module can also specify its search space, cost model, individual search strategy (e.g., heuristic, exhaustive, simulated annealing, etc.), termination policy, and rule properties (e.g., rule priorities that define the order in which rules are applied in each module). There are also *knobs* that a DBI can set before optimization starts. These knobs are essentially initialization steps that set various parameters of a module.

An optimizer is constructed by stacking various modules together. The order of modules defines the order of term rewrite rules that are applied to a term. A module can request another to optimize a term and return its results. Thus, communication between modules is bi-directional.

In theory, Sciore and Sieg propose a very general framework for optimizer design. However, since this model has not been implemented (to our knowledge), it is unclear whether this approach is used as an interpreter (thus degrading performance), or to generate a monolithic optimizer. The algorithm for the latter is not described, so the performance of the resulting optimizer is hard to predict. Also, it is not evident whether the general nature of the framework makes it hard and difficult to use.

Mitchell, Dayal, and Zdonik [16, 17] propose a framework called Epoq in which optimizers are constructed using extensible *regions*. A region is defined by a stated *goal* (e.g., lower cost, join reorder, etc.). Each region defines a control strategy that transforms a query into alternative forms based on its internal transformation rules. A region can also call a child region to transform a subquery. In this approach, an optimizer is specified as a rooted, directed, acyclic graph of regions.

The root region is responsible for optimizing a user query. A parent region controls which of its child regions transforms a query. Thus, the expansion of the search space depends on two factors: the internal transformations of a region, and the parent's determination of the most appropriate region (based on its stated goal) to effect a transformation.

The transformation rules in a region consist of applicability conditions together with a test to check whether a transformed query meets the region's stated goal. If not, then transformation rules are applied repeatedly (perhaps based on some heuristic) until either the goal is satisfied, or the region fails and returns to its parent.

As in the Sciore and Sieg approach, the model proposed by Mitchell, Dayal, and Zdonik has not been implemented. Thus, it is not clear whether such a general framework can generate optimizers that are efficient and that encompass a large domain of commonly available optimizers. It is also

not clear whether the interaction between regions can be expressed in a compact framework so that regions can be reused in various optimizer configurations.

6 Conclusion and Future Work

Current rule-based query optimizers do not provide a very intuitive and conceptually streamlined framework to define rules and actions. Our experiences with the Volcano optimizer generator suggest that its model of rules and the expression of these rules is much more complicated and too low-level than it needs to be. As a consequence, rule sets in Volcano are fragile, hard to write, and debug. Similar problems may exist in other contemporary rule-based query optimizers.

In this paper, we presented a general approach to constructing optimizers using pre-fabricated components (layers). We also presented preliminary experimental evidence from simple layered optimizers that demonstrate that optimizer performance is not sacrificed by using reusable and extensible layers. More experiments are necessary to validate conclusively that a layered optimizer can be just as efficient as (and more extensible than) monolithic optimizers. Our primary goal in this paper was to quickly design and implement a *practical* framework for specifying layered optimizers. Future work can add more generality to this approach. An important goal is to use Prairie to automatically generate efficient optimizers in P2 [3, 4].

We believe that rule-based query optimizers will be standard tools of future database systems. The pragmatic difficulties of using conventional rule-based optimizers led us to develop Prairie and its philosophy of using small, well-defined building-blocks for rule set specification. This results in several improvements over existing rule-based optimizers:

1. it offers a conceptually more streamlined model for rule specification;
2. optimizer specification and generation is simplified;
3. and it has efficient implementations.

The primary motivation behind Prairie's building-blocks technology is to make the process of optimizer construction as automated, simple, and error-proof as possible. Our experiences suggest that Prairie is a very useful step in that direction. These advantages greatly enhance optimizer extensibility and make rule sets less brittle. A consequence is that Prairie rules are simpler and more robust than rules of existing optimizers (e.g., Volcano).

Our future work will concentrate on generalizing the layer paradigm to allow non-linear compositions and the generation of non-relational (e.g., object-oriented) DBMS optimizers.

References

[1] Don Batory and Bart Geraci. Validating component compositions in software system generators. Technical Report TR 95–03, The University of Texas at Austin, 1995.

[2] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[3] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *Proceedings 1993 ACM SIGSOFT Conference*, pages 191–199, Los Angeles, December 1993.

[4] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca model of software-system generators. *IEEE Software*, 11(5):89–94, September 1994.

[5] José A. Blakeley, William J. McKenna, and Goetz Graefe. Experiences building the Open OODB query optimizer. In *Proceedings 1993 ACM SIGMOD International Conference on Management of Data*, pages 287–296, Washington, May 1993.

[6] Dean Daniels, Pat Selinger, Laura Haas, Bruce Lindsay, C. Mohan, Adrian Walker, and Paul Wilms. An introduction to distributed query compilation in R*. In *Proceedings 2nd International Conference on Distributed Databases*, pages 291–309, Berlin, September 1982.

[7] Dinesh Das. *Making Database Optimizers More Extensible*. PhD thesis, The University of Texas at Austin, 1995.

[8] Dinesh Das and Don Batory. Prairie: A rule specification framework for query optimizers. In *Proceedings 11th International Conference on Data Engineering*, pages 201–210, Taipei, March 1995.

[9] Johann Christoph Freytag. A rule-based view of query optimization. In *Proceedings 1987 ACM SIGMOD International Conference on Management of Data*, pages 173–180, San Francisco, May 1987.

[10] Goetz Graefe. Volcano, an extensible and parallel query evaluation system. Technical Report CU–CS–481–90, University of Colorado at Boulder, July 1990.

[11] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[12] Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In *Proceedings 1987 ACM SIGMOD International Conference on Management of Data*, pages 387–394, San Francisco, May 1987.

[13] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in Starburst. Research Report RJ 6610, IBM Almaden Research Center, December 1988.

[14] Won Kim, David S. Reiner, and Don S. Batory, editors. *Query Processing in Database Systems*. Springer-Verlag, 1985.

[15] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proceedings 1988 ACM SIGMOD International Conference on Management of Data*, pages 18–27, Chicago, June 1988.

[16] Gail Mitchell, Umeshwar Dayal, and Stanley B. Zdonik. Control of an extensible query optimizer: A planning-based approach. In *Proceedings 19th International Conference on Very Large Data Bases*, pages 517–528, Dublin, August 1993.

[17] Gail Mitchell, Stanley B. Zdonik, and Umeshwar Dayal. An architecture for query processing in persistent object stores. In *Proceedings of the Hawaii International Conference on System Sciences*, volume II, pages 787–798, January 1992.

[18] Edward Sciore and John Sieg, Jr. A modular query optimizer generator. In *Proceedings 6th International Conference on Data Engineering*, pages 146–153, Los Angeles, February 1990.

[19] C. T. Yu and C. C. Chang. Distributed query processing. *ACM Computing Surveys*, 16(4):399–433, December 1984.