In *Proceedings 11th International Conference on Data Engineering*, pages 201–210, Taipei, March 1995

Prairie: A Rule Specification Framework for Query Optimizers[*†]

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

Abstract

From our experience, current rule-based query optimizers do not provide a very intuitive and well-defined framework to define rules and actions. To remedy this situation, we propose an extensible and structured algebraic framework called Prairie for specifying rules. Prairie facilitates rule-writing by enabling a user to write rules and actions more quickly, correctly and in an easy-to-understand and easy-to-debug manner.

Query optimizers consist of three major parts: a search space, a cost model and a search strategy. The approach we take is only to develop the algebra which defines the search space and the cost model and use the Volcano optimizer-generator as our search engine. Using Prairie as a front-end, we translate Prairie rules to Volcano to validate our claim that Prairie makes it easier to write rules.

We describe our algebra and present experimental results which show that using a high-level framework like Prairie to design large-scale optimizers does not sacrifice efficiency.

1 Introduction

Query optimization [8] is a fundamental part of database systems. It is the process of generating an efficient access plan for a database query. Informally, an access plan is an execution strategy for a query; it is the sequence of low-level database retrieval operations that, when executed, produce the database records that satisfy the query. There are three basic 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 a DBMS. This simply reflects the fact that traditional database systems are themselves monolithic: the algorithms used for storing and retrieving data are hard-wired and are rather difficult to change. The need to have extensible database systems, and in turn extensible optimizers, has long been recognized in systems like Genesis [1], EXODUS [9], Starburst [10], and Postgres [12]. Rule-based query optimizers are among the major conceptual advances that have been proposed to deal with query optimizer extensibility [6, 7, 9, 10]. The extensibility translates into the ability to incorporate new operators, algorithms, cost models, or search strategies without changing the optimization algorithm.

In this paper, we describe an algebraic framework called *Prairie* for specifying rules in a rule-based query optimizer. Prairie is similar to other rule specification languages like Starburst [10] and Volcano [7], and indeed, we have based our work on Volcano to capture most of the advantages of rule-based optimizers. However, Prairie attempts to provide some key features that, we have found, simplify the effort in writing rules:

1. A framework in which users can define a query optimizer concisely in terms of a well-defined set of operators and algorithms. *All* operators and algorithms are considered first-class objects, i.e., *any* of them can occur in any rule, and *only* these operators and algorithms can appear in rules. This scheme eliminates the need for special classes of operators and algorithms, such as enforcers in Volcano and glue in Starburst, that significantly complicate rule specification.

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

[†]An expanded version of this paper is available as Technical Report TR 94–16 by anonymous ftp from `ftp.cs.utexas.edu`.

2. A framework in which users can define a list of properties to characterize the expressions generated in the optimization process. Again, the goal here is to allow the user to treat *all* properties as having equal status. This is different from Volcano where the user must classify properties as logical, physical, or operator/algorithm arguments.

3. A framework in which users can specify mapping functions between properties concomitantly with the corresponding rules. This contrasts with existing approaches in which mappings between properties are fragmented into multiple functions and at logically different places than the corresponding rules. Research into rule-based optimizers has revealed that property-mapping functions are a major source of user effort, so this is an important goal.

4. The format (Prairie) in which users can cleanly specify rules is not necessarily the same format needed for generating efficient optimizers. Thus, there is a need for a pre-processor (written by us) that translates between these competing representations.

Prairie strives for uniformity in dealing with issues that have been a source of most user effort and potential user errors. In the following sections, we present the Prairie framework. We explain how our P2V pre-processor maps Prairie rule specifications into Volcano rule specifications that can be processed efficiently. Experimental results to support this claim are given in Section 4, where we compare implementations of the Texas Instruments Open OODB query optimizer using both Prairie and Volcano. We conclude with a summary and related research.

2 Prairie: A language for rule specification

The basic concepts and definitions that underlie the Prairie model are presented in this section. The goal is to lay a foundation for reasoning about query optimization algebraically; this is necessary for our subsequent discussion about translating Prairie specifications to those of Volcano.

2.1 Notation and assumptions

Stored Files and Streams. A file is *stored* if its tuples reside on disk. In the case of relational databases, stored files are sometimes called *base relations*; we will denote them by R or R_i. In object-oriented schemas, stored files are *classes*; we will denote them by C or C_i. Henceforth, whenever we refer to a stored file, we mean a relation or a class; when the distinction is unimportant, we will use F or F_i. A *stream* is a sequence of tuples and is the result of a computation on one or more streams or stored files; tuples of streams are returned one at a time, typically on demand. Streams can be *named*, denoted by S_i, or *unnamed*.

Database Operations. An *operation* is a computation on one or more streams or stored files. There are two types of database operations in Prairie: abstract (or implementation-unspecified) operators and concrete algorithms. Each is detailed below.

Operators. Abstract (or conceptual) *operators* specify computations on streams or stored files; they are denoted by all capital letters (e.g., JOIN). Operators have two types of parameters: essential and additional. *Essential parameters* are the stream or file inputs to an operator; these are the primary inputs to be processed by an operator. *Additional parameters* are "fine-grain" qualifications of an operator; their purpose is to describe an operator in more detail than essential parameters.

Algorithms. *Algorithms* are concrete implementations of conceptual operators; they will be represented in lower case with the first letter capitalized (e.g., Nested_loops). Algorithms have at least the same essential and additional parameters as the conceptual operators that they implement.[1] Furthermore, there can be, and usually are, several algorithms for a particular operator.

Table 1 lists some operators and algorithms implementing them together with their additional parameters.

Operator Trees. An *operator tree* is a rooted tree whose non-leaf, or *interior,* nodes are database operations (operators or algorithms) and whose leaf nodes are stored files. The children of an interior node in an operator tree are the essential parameters (i.e., the stream or file parameters) of the node. Additional parameters are implicitly attached to each node. Algebraically, operator trees are compositions of database operations; thus, we will also call operator trees *expressions*; both terms will be used interchangeably.

EXAMPLE 1. A simple expression and its operator tree representation are shown in Figure 1(a). Relations R_1 and R_2 are first RETrieved, then JOINed, and finally SORTed resulting in a stream sorted on a specific attribute. The figure shows only the essential parameters of the various operators, not the additional parameters. □

[1] Algorithms may have *tuning parameters* which are not parameters of the operators they implement.

Operator	Description	Additional Parameters	Algorithm
JOIN(S_1, S_2)	Join streams S_1, S_2	tuple_order join_predicate	Nested_loops(S_1, S_2)
			Merge_join(S_1, S_2)
RET(F)	Retrieve file F	tuple_order selection_predicate projected_attributes	File_scan(F)
			Index_scan(F)
SORT(S_1)	Sort stream S_1	tuple_order	Merge_sort(S_1)
			Null(S_1)

Table 1: Operators and algorithms in a centralized query optimizer and their additional parameters

Property	Description
join_predicate	join predicate for JOIN operator
selection_predicate	selection predicate for RET operator
tuple_order	tuple order of resulting stream, DONT_CARE if none
num_records	number of tuples of resulting stream
tuple_size	size of individual tuple in stream
projected_attributes	projected attributes for RET operator
attributes	list of attributes
cost	estimated cost of algorithm

Table 2: Properties of nodes in an operator tree

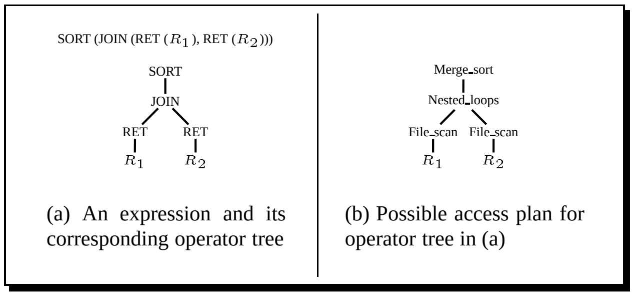
SORT (JOIN (RET (R_1), RET (R_2)))
SORT
JOIN
RET
RET
R_1
R_2
(a) An expression and its corresponding operator tree
Merge_sort
Nested_loops
File_scan
File_scan
R_1
R_2
(b) Possible access plan for operator tree in (a)

Figure 1: Example of an operator tree and access plan

Descriptors. A *property* of a node is a (user-defined) variable that contains information used by an optimizer. 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. As an example, Table 2 lists some typical properties that might be used in a descriptor. Note that descriptors for stream and stored files may have different properties. The following notations will be useful in our subsequent discussions. If S_i is a stream, then $\mathbf{D_i}$ is its descriptor. Annotations of S_i are accessed by a structure member relationship, e.g., $\mathbf{D_i}$.num_records. 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,

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

corresponds to the operator tree in Figure 1(a), and shows the descriptors of the various nodes. □

A notational simplification can be made here. Additional parameters of operators can be treated the same way as other properties of a node; essential parameters, however, are *expressions*. Thus, the term descriptor in the remainder of this paper will refer to a set of properties, including additional parameters, as shown in Table 2.

Currently, descriptor properties are defined entirely by the user; however, we envision providing a hierarchy of predefined descriptor types to aid this process.

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

EXAMPLE 3. An access plan for the operator tree in Figure 1(a) is shown in Figure 1(b). □

2.2 Prairie optimization paradigm

Prairie admits two rather different means of optimization: top-down and bottom-up. A top-down query optimizer optimizes the parents of a node prior to optimizing the node itself. A bottom-up optimizer optimizes the children of a node prior to optimizing the node. The earliest optimizers (System R [11] and R* [3]) employed the bottom-up approach.

Our research concentrates on a top-down optimization of operator trees. We have chosen this approach because we intend to translate Prairie rules into the format required by the Volcano query optimizer generator [7] which is based on a top-down strategy. Given an appropriate search engine, Prairie can potentially also be used with a bottom-up optimization strategy; however, we will not discuss this approach in this paper.

In query optimization, there are certain annotations (such as additional parameters) that are known before any optimization is begun. These annotations can be computed at the time that the operator tree is initialized, and will not change with application of rules. Our following discussions assume operator trees are initialized.

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 based on additional conditions; the transformation also results in a mapping of descriptors between expressions. We define T-rules and I-rules precisely in the following sections and illustrate them with examples. Our examples are chosen from rules that would be used in a centralized relational query optimizer; the operators, algorithms, and properties are subsets of those in Tables 1 and 2.

```
$E(x_1, \ldots, x_n) : \mathbf{D_1} \Longrightarrow E'(x_1, \ldots, x_n) : \mathbf{D_2}$                (1)
{{
        pre-test statements
}}
test
{{
        post-test statements
}}
```

(a) General form of a T-rule

```
JOIN(JOIN($S_1, S_2$) : $\mathbf{D_4}$, $S_3$) : $\mathbf{D_5}$                (2)
  $\Longrightarrow$ JOIN($S_1$, JOIN($S_2, S_3$) : $\mathbf{D_6}$) : $\mathbf{D_7}$
{{
        $\mathbf{D_6}$.attributes $=$ union ($\mathbf{D_2}$.attributes, $\mathbf{D_3}$.attributes) ;
}}
is_associative ($\mathbf{D_6}$.join_predicate, $\mathbf{D_6}$.attributes, $\mathbf{D_5}$.join_predicate)
{{
        $\mathbf{D_7} = \mathbf{D_5}$ ;
        $\mathbf{D_7}$.join_predicate $=$ $\mathbf{D_4}$.join_predicate ;
        $\mathbf{D_6}$.tuple_size $=$ $\mathbf{D_2}$.tuple_size $+$ $\mathbf{D_3}$.tuple_size ;
        $\mathbf{D_6}$.num_records $=$ cardinality ($\mathbf{D_2}, \mathbf{D_3}$) ;
}}
```

(b) Join associativity

Figure 2: T-rule

2.3 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. Equation (1) (shown in Figure 2(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[2] statements. The left-hand sides of these statements refer to descriptors of expressions on the right-hand side of the T-rule; the right-hand sides of the statements can refer to any descriptor in the T-rule. Function (called *helper* functions) calls can also appear on the right side of the assignment statements. Thus, descriptors on the *left-hand side* of a T-rule are *never* changed in the rule's actions. A *test* is needed to determine if the transformations of the T-rule are in fact 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 the T-rule.[3]

[2]The actions can be non-assignment statements (like function calls), but in this case, the P2V pre-processor (described in Section 3) needs some hints about the properties that are changed by the statement in order to correctly categorize each property. For simplicity, in this paper, we assume all actions consist of assignment statements.

```
$E(x_1, \ldots, x_n) : \mathbf{D_1} \Longrightarrow A(x_1, \ldots, x_n) : \mathbf{D_2}$                (3)
test
{{
        pre-opt statements
}}
{{
        post-opt statements
}}
```

(a) General form of an I-rule

```
SORT($S_1$) : $\mathbf{D_2} \Longrightarrow$ Merge_sort($S_1$) : $\mathbf{D_3}$                (4)
($\mathbf{D_2}$.tuple_order != DONT_CARE)
{{
        $\mathbf{D_3} = \mathbf{D_2}$ ;
}}
{{
        $\mathbf{D_3}$.cost $=$ $\mathbf{D_1}$.cost
                $+(\mathbf{D_3}$.num_records$) * \log(\mathbf{D_3}$.num_records$)$ ;
}}
```

(b) Merge-sort sort algorithm

Figure 3: I-rule

EXAMPLE 4. The associativity of JOINs is expressed by T-rule (2) in Figure 2(b). □

2.4 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 given by Equation (3) (shown in Figure 3(a)).

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 parameters 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. This *does not*, however, imply a bottom-up optimization strategy. It simply means that although I-rules are applied to parents before their children are optimized, the *cost* (and

[3]We suspect it is possible to use data-flow analysis to partition the assignment statements automatically, but for now, we let the rule-writer do the partitioning.

other properties in the post-opt section) of the parent cannot be computed until the children have been optimized.

EXAMPLE 5. Equation (4) (in Figure 3(b)) shows the I-rule that implements the SORT operator by Merge_sort. □

2.5 Null algorithm

Recall that, in Section 1, we mentioned that Prairie allows users to treat all operators and algorithms as first-class objects, i.e., all operators and algorithms are explicit, in contrast to enforcers in Volcano or glue in Starburst. This requires that Prairie provide a mechanism where users can also "delete" one or more of the explicit operators from expressions. This is done by having a special class of I-rules that have the form given by Equation (5) in Figure 4(a). The left side of the rule is a single abstract operator O with one stream input S_1. The right side of the rule is an algorithm called "Null" with the same stream input but with a different descriptor. As the name suggests, the Null algorithm is supposed to pass its input unchanged to algorithms above it in an operator tree. This is accomplished in the I-rule as follows.

The test for this I-rule is always TRUE, i.e., any node in an operator tree with O as its operator can be implemented by the Null algorithm. The actions associated with this rule have a specific pattern. The pre-opt section consists of three statements. The first statement copies the descriptor of the operator O to the algorithm Null. The second statement sets the descriptor of the stream S_1 on the right side to the descriptor of the stream S_1 on the left side. Why is it necessary to do this? The key lies in the third statement. This statement copies the property "property" of the operator O node on the left side to the "property" of the input stream S_1 on the right side. Since left-hand side descriptors cannot be changed in an I-rule, a new descriptor $\mathbf{D_3}$ is necessary for S_1 to convey the property propagation information.

The post-opt section in the I-rule has only a cost-assignment statement; this simply sets the cost of the Null node to the cost of its optimized input stream.

The Null algorithm, therefore, serves to effectively transform a single operator to a no-op.

EXAMPLE 6. Equation (6) (in Figure 4(b)) shows the I-rule that rewrites the SORT operator to use a Null algorithm. □

3 The P2V pre-processor

In Section 1, we enumerated the four primary goals of Prairie, viz., uniformity in operator and algorithms; uniformity in properties; uniformity in property-transformations;

$O(S_1) : \mathbf{D_2} \Longrightarrow \text{Null}(S_1 : \mathbf{D_3}) : \mathbf{D_4}$ (5)
TRUE
{{
$\mathbf{D_4} = \mathbf{D_2}$;
$\mathbf{D_3} = \mathbf{D_1}$;
$\mathbf{D_3}$.property $=$ $\mathbf{D_2}$.property ;
}}
{{
$\mathbf{D_4}$.cost $=$ $\mathbf{D_3}$.cost ;
}}

(a) General form of a "Null" I-rule

SORT$(S_1) : \mathbf{D_2} \Longrightarrow \text{Null}(S_1 : \mathbf{D_3}) : \mathbf{D_4}$ (6)
TRUE
{{
$\mathbf{D_4} = \mathbf{D_2}$;
$\mathbf{D_3} = \mathbf{D_1}$;
$\mathbf{D_3}$.tuple_order $=$ $\mathbf{D_2}$.tuple_order ;
}}
{{
$\mathbf{D_4}$.cost $=$ $\mathbf{D_3}$.cost ;
}}

(b) Null sort algorithm

Figure 4: The "Null" algorithm concept

and efficient generation of Prairie optimizers. The first three goals are driven by the need for conceptual simplicity; however, they alone do not necessarily generate efficient optimizers. The P2V pre-processor ensures that efficient optimizers can be realized from Prairie specifications, by translating them to the Volcano framework and then generating an optimizer by compiling with the Volcano search engine. This Prairie optimizer-generator paradigm is shown schematically in Figure 5. The pre-processor itself is 4500 lines of `flex` and `bison` code. In this section, we briefly describe the pre-processor steps and explain why the Prairie-to-Volcano transformation is non-trivial. A more detailed description of the pre-processor is given in [5].

The specification of an optimizer in Volcano consists of a set of transformation rules (called "trans_rules") and implementation rules (called "impl_rules"), a set of properties, and some support functions. The join associativity trans_rule (cf. Figure 2(b)) in Volcano is as follows[4]:

```
(JOIN ?op_arg5 ((JOIN ?op_arg4 (?1 ?2)) ?3))
    ->(JOIN ?op_arg7 (?1 (JOIN ?op_arg6 (?2 ?3))))
```

The important point to note is the use of *operator arguments* (denoted by "op_arg" in rules); these arguments contain properties used in the rule's actions, but unlike Prairie, they do not contain *all* the properties of an operator tree node. There are other property classes, like algorithm argument, logical property, system property, physical property, and cost. Thus, while Prairie uses a uniform descriptor to

[4]There are conditions and actions associated with Volcano rules that are not shown here.

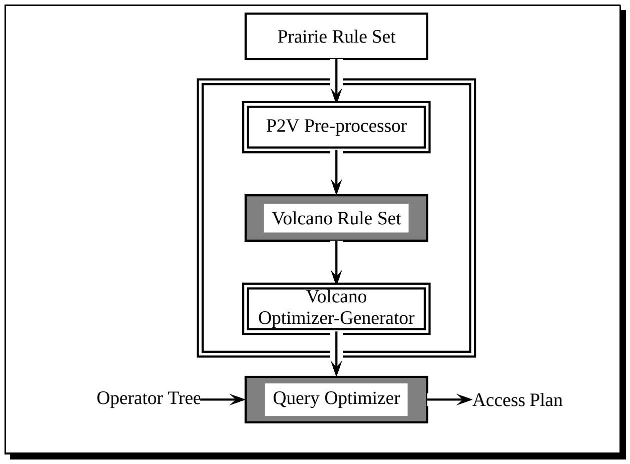

Figure 5: The Prairie optimizer-generator paradigm. Double-boxed modules represent software generators, shaded boxes represent generated programs. The outermost double-boxed portion denotes the Prairie optimizer generator.

encode properties, Volcano partitions the properties into different classes. The P2V pre-processor partitions a Prairie descriptor into the different property classes required by Volcano. This is a non-trivial task, since it requires parsing the Prairie rules and their actions.

Impl_rules in Volcano defer most of the actions associated with the rules to support functions. Each algorithm has four support functions associated with it. A Prairie specification, on the other hand, contains all the actions in the corresponding rule. The P2V pre-processor parses a Prairie I-rule, and automatically generates all the Volcano support functions from the rule. This is also a complex process, since it depends partly on the partitioning of properties mentioned in the last paragraph, and also because it requires relocating pieces of code from Prairie rules to Volcano support functions.

The third salient feature of a Volcano specification is the presence of implicit, or hidden, algorithms, called *enforcers*. In Prairie, all algorithms are explicit. Consider, for example, the Merge_sort algorithm in Figure 3(b). In a Volcano specification, this algorithm would be classified as an enforcer, since it enforces the sortedness property. The P2V pre-processor determines the Prairie algorithms that are functionally Volcano enforcers, and deletes the corresponding Prairie rules to generate the Volcano specification. This requires the pre-processor to migrate the (deleted) rule's actions to Volcano support functions.

The P2V pre-processor also generates a set of compact Volcano rules by merging Prairie rules whenever possible. Consider, for example, the following set of rules in Prairie:

$$\text{JOIN}(S_1,S_2){:}\mathbf{D_3} \Longrightarrow \text{JOPR}(\text{SORT}(S_1){:}\mathbf{D_4},\text{SORT}(S_2){:}\mathbf{D_5}){:}\mathbf{D_6}$$

$$\text{SORT}(S_1){:}\mathbf{D_2} \Longrightarrow \text{Null}(S_1{:}\mathbf{D_3}){:}\mathbf{D_4}$$

$$\text{JOPR}(S_1,S_2){:}\mathbf{D_3} \Longrightarrow \text{Nested\_loops}(S_1{:}\mathbf{D_4},S_2){:}\mathbf{D_5}$$

The first rule is a T-rule, and the next two are I-rules. The P2V pre-processor combines the above set of Prairie rules into a single I-rule,

$$\text{JOIN}(S_1,S_2){:}\mathbf{D_3} \Longrightarrow \text{Nested\_loops}(S_1{:}\mathbf{D_4},S_2){:}\mathbf{D_5}$$

and then translates it into a single Volcano impl_rule.

4 Experimental results

This section presents experimental results which demonstrate the value of Prairie in specifying rule sets of rule-based optimizers. Our experiments consist of specifying rule-based optimizers using Prairie and generating optimizers using the P2V pre-processor and the optimizer-generator paradigm of Figure 5.

In [4], we presented an implementation of a centralized relational query optimizer using Prairie. Using the P2V translator, we translated this to Volcano format and optimized several queries using the resultant optimizer. For comparison, we hand-coded the same optimizer directly in Volcano. The results presented there showed that, using Prairie (compared to directly using Volcano) resulted in approximately 50% savings in lines of code with negligible (less than 5%) increase in query optimization time. However, the optimizer was quite small in terms of the number of operators, algorithms and rules.

For a more realistic evaluation of Prairie, we needed answers to the following questions:

1. Is Prairie adequate for large-scale rule sets?
2. How is programmer productivity enhanced by the high-level abstractions of Prairie?
3. Can Prairie rule sets be translated automatically into efficient implementations?

We addressed the first question by using the Texas Instruments Open OODB query optimizer rule set, which has the largest publicly available rule set. We describe this optimizer in the next section, and then give our assessments to the last two questions in subsequent sections.

4.1 The Texas Instruments Open OODB query optimizer

The Texas Instruments Open Object-Oriented Database Management System is an open, extensible, object-oriented database system which provides users an architectural framework that is configurable in an incremental manner. The query optimizer in the Open OODB [2] is generated using Volcano. It is written as a set of trans_rules and impl_rules that define the algebra of an object-oriented database system. Currently, there are 17 transformation rules and 9 implementation rules together with about $13,000$ lines of code for support functions; this, of course, can be changed by an Open OODB user for specific needs.

4.2 Programmer productivity

Programmer productivity can be measured in different ways. An admittedly simplistic metric is the number of lines of code that must be written. But there are also less tangible measures, such as the amount of conceptual effort needed to understand a particular programming task. Our experience with the Open OODB query optimizer suggests that Prairie excels on the latter, while offering modest reductions in the volume of code that needs to be written.

We converted by hand the Open OODB query optimizer's Volcano specifications to Prairie. This was a non-trivial task because of the relatively large size of the rule set and the complexity of the support functions. This was where we found Prairie helped in conceptually simplifying the rules and actions. We then used our P2V pre-processor to reconstitute these Prairie specifications as Volcano specifications. As described in Section 3, this process involved a considerable level of complexity, partly because the Prairie specification had 22 T-rules and 11 I-rules compared to 17 trans_rules and 9 impl_rules in the Volcano specification; the reconstituted Volcano specification had the same number of trans_rules and impl_rules as the original hand-coded specification.

Converting the Open OODB optimizer rule set into Prairie format actually simplified its specification as the complexities of the Volcano model were removed. The reduction in lines of code was modest — there was about a 10% savings.[5] However, as mentioned above, savings in lines of code do not adequately reflect increases in programmer productivity. We found the encapsulated specifications of Prairie — namely, the use of a single descriptor and fewer explicit support functions — made rule programming *much* easier.

4.3 Performance results using the Open OODB optimizer

The acid test of Prairie was whether Prairie specifications could be translated into efficient optimizer implementations. Our experiments using the Open OODB consisted of optimizing 8 different queries using the two query optimizers generated, respectively, using Prairie and using Volcano directly (in the remainder of this section, we will use "Prairie" and "Volcano" to denote these two approaches). There were 4 distinct expressions that were used to generate the queries used in the experiments; these are shown in Figure 6. Each expression represents an N-way join query for varying N.

The first expression E1 is a simple retrieval and join of base classes. The second, E2, is also a join of base classes; however, after each class retrieval, an attribute has to be materialized (i.e., brought into view) before the join. The third and fourth expressions (E3 and E4) are the same as the first and second (E1 and E2) respectively, except that there is a selection of attributes (the select operator is the root of the expressions).[6]

The algebra that was used in the Prairie and Volcano optimizers for our experiments consisted of 5 relational operators SELECT, PROJECT, JOIN, RET and UNNEST (for set-valued attributes) and an object-oriented operator called MAT (for MATerialize; it is fundamentally a pointer-chasing operator for attributes of a class). There were 8 algorithms.

There are many parameters that can be varied when benchmarking a query optimizer. Since our objective was to verify that the Prairie approach did not sacrifice efficiency, our criteria for the queries was that they test a majority of the rules, with varying properties of the base classes. To this end, we tested our optimizer (and the Volcano optimizer) with 8 different queries (shown in Table 3). The eight queries Q1 through Q8 are derived from the 4 expressions in Figure 6. Each expression E1 through E4 is used to obtain two queries for a fixed number N of JOINs in the expression. The only difference between the two queries obtained from an expression is that the first one does not contain any indices on any classes, whereas the second one contains a single index on each base class occurring in the expression. In expressions where a SELECT is present (E3 and E4), the selection predicate is a conjunction of equality predicates $bc_i == const_i$, where bc_i is an attribute of class C_i, and $const_i$ is a constant (we arbitrarily set this to i, because its value doesn't affect the correctness or performance of the optimizer). In addition, for queries with a SELECT and whose base classes have indices (Q6 and Q8 in Table 3), the (single) index of each base class was chosen to be the attribute referenced in the selection predicate. For example, class C_i was chosen to have an index on attribute bc_i. The join predicates for each JOIN were chosen at random, and were always equality predicates. The choice of JOIN predicates was such that the queries corresponded to linear query graphs. In the future, we will experiment with non-linear (e.g., star) query graphs.

Table 3 also shows the number of trans_rules and

[5]The original Volcano specification had $13,400$ lines, the Prairie specification had $12,100$ lines, and the P2V-generated Volcano specification had $15,800$ lines.

[6]The most complex expression E4 consists of all operators in the algebra, except PROJECT and UNNEST. PROJECT was not considered because it appeared in only one impl_rule and no trans_rules, and thus, would not affect the size of the search space of abstract expressions. UNNEST was not considered because it appeared in exactly one trans_rule and one impl_rule; including it in our queries would have increased the number of parameters that could affect our run-times. We preferred to concentrate on simple JOIN expressions.

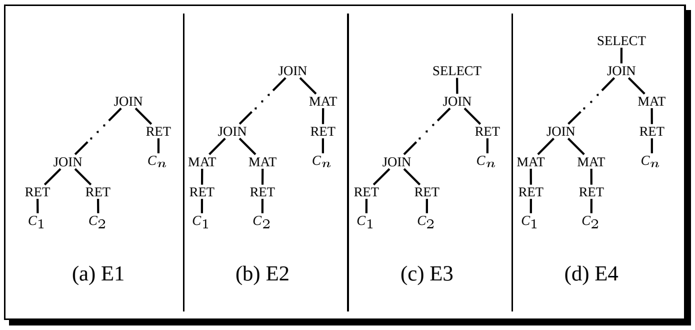

Figure 6: Expressions used in generating queries for experiments

Query	Indices?	Expression	Rules matched	
			trans_rules	impl_rules
Q1	No	E1	3	3
Q2	Yes			
Q3	No	E2	8	4
Q4	Yes			
Q5	No	E3	9	5
Q6	Yes			
Q7	No	E4	16	7
Q8	Yes			

Table 3: Queries used in experiments

impl_rules that are matched by each expression. These are the rules whose left hand sides match a sub-expression. However, not all the rules were necessarily applicable. For instance, an impl_rule with an index scan would not apply to Q3, although it might apply to Q4.

Queries Q1 through Q8 were optimized for increasing number N of JOINs. For a fixed number of JOINs in a query, we varied the cardinalities of the base classes 5 times, each time generating a query with different class properties, and averaged the run-times over the 5 query instances to generate the per-query optimization time. Thus, each point in our graphs represents the average of 5 queries. The run-times were measured[7] using the GNU `time` command. All experiments were performed on a lightly loaded DECstation 5000/200 running Ultrix 4.2.

The optimization times for each query for both approaches (Prairie and Volcano) are shown in Figures 7(a) through 7(h). The number of joins in each set of graphs was varied to a maximum of 8, or until virtual memory was exhausted.

The first set of graphs (Figures 7(a) and 7(b)) shows the performance of a simple relational-type query. The optimization times are almost identical between Prairie and Volcano, and the notable point is that the presence of an index does not change the optimizer's behavior, i.e., the two graphs are identical. This arises because the optimizer algebra had only two join algorithms (pointer join and hash join), neither of which makes use of any indices.

The second set of graphs (Figures 7(c) and 7(d)) shows the results of optimizing Q3 and Q4. Here, as in Figures 7(a) and 7(b), the presence (or absence) of indices makes no difference. Both the Prairie and Volcano approaches have comparable run-times. The sharp jump in the graphs from 7-way to 8-way joins is due to the fact that since all optimization is done in main memory, dynamic memory allocation (caused by `malloc` calls) results in a lot of thrashing at this point.

[7]Since the run-times were too small to be measured accurately with `time`, each query instance was optimized 3000 times (in a loop) and the total time was divided by 3000 to get the per-query optimization time.

We speculate that in systems with more virtual memory, the graphs will be smoother.

The third and fourth sets of graphs in Figures 7(e) through 7(h) are optimizations of queries with a selection predicate. In these cases, the presence of an index makes a difference if the index is referenced in the selection predicate (as we designed). Also, in these two figures, the performance of both Prairie and Volcano was almost identical, except that Prairie does slightly worse due to the larger number of `malloc` calls that the P2V translator introduces. Also, note that we could only go up to 3-way joins before virtual memory was exhausted. As the available memory decreases, there is increased thrashing (as shown by the sharp changes in slope in the plots) resulting in a much slower optimization process.

In all four sets of plots, we can see that Prairie performs with almost (less than 5% variation) the same efficiency as Volcano. In extreme cases, when memory is scarce, Prairie runs more slowly (about 15%) (e.g., Figure 7(f)), but we believe that this situation already represents a serious bottleneck for both Volcano and Prairie.

The results presented in this section show that Prairie optimizers need not sacrifice efficiency for clarity, even for large rule sets. More research and validation is necessary to verify that Prairie is an efficient tool for optimizer specification.

5 Related research

The System R optimizer [11] was the most important development in query optimization research. It was a cost-based centralized relational query optimizer and introduced a variety of key concepts like "interesting" expressions, cardinality estimation using selectivity factors and dynamic programming with pruning of search space. These concepts continue to be important in query optimizer research.

The query optimizer in R* [3] works in essentially the same way as that of System R, except that R* is a distributed database system which introduces some subtle complica-

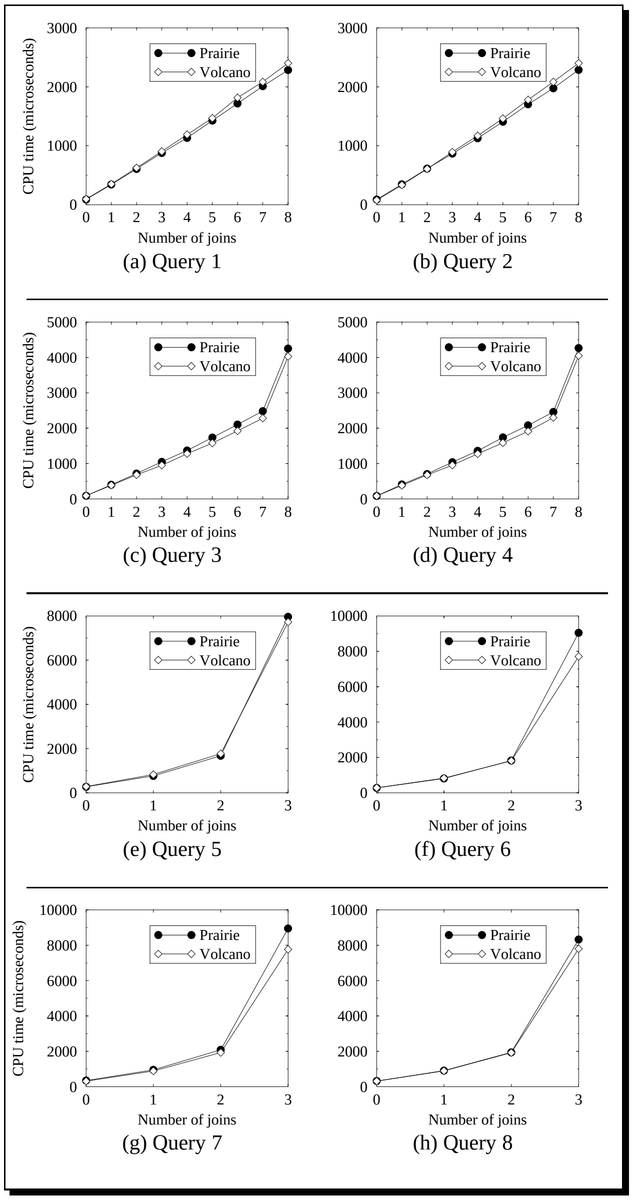

Figure 7: Query optimization times for Q1 through Q8

tions in its query optimizer.

The Starburst query optimizer [10] uses rules for all decisions that need to be taken by the query optimizer. The rules are functional in nature and transform a given operator tree into another. The rules are commonly those that reflect relational calculus facts. In Starburst, the query rewriting phase is different from the optimization phase. The rewriting phase transforms the query itself into equivalent operator trees based on relational calculus rules. The plan optimization phase selects algorithms for each operator in the operator tree that is obtained after rewriting. The disadvantage of separating the query rewrite and the optimization phases is that pruning of the search space is not possible during query rewrite, since the rewrite phase is non-cost-based.

Freytag [6] describes a rule-based query optimizer similar to Starburst. The rules are based on LISP-like representations of access plans. The rules themselves are recursively defined on smaller expressions (operator trees). Although several expressions can contain a common sub-expression, Freytag doesn't consider the possibility of sharing. Expressions are evaluated each time they are encountered. This is obviously inefficient. In addition, as in Starburst, he doesn't consider the cost transformations inherent in any query optimizer; rules are syntactic transformation rules.

EXODUS [9] provides an optimizer generator which accepts a rule-based specification of the data model as input. The optimizer generator compiles these rules, together with pre-defined rules, to generate an optimizer for the particular data model and set of operators. Unlike Freytag, the optimizer generator for EXODUS allows for C code along with definitions of new rules. This allows the database implementor the freedom to associate any action with a particular rule. Operator trees in EXODUS are constructed bottom-up from previously constructed trees.

The Volcano optimizer generator project [7] evolved from the EXODUS project. It is different from all the above optimizers in one significant way: it is a top-down optimizer compared with the bottom-up strategy of the others. Operator trees are optimized starting from the root while sub-trees are not yet optimized. This leads to a constraint-driven generation of the search space. While this method results in a tight control of the search space, it is unconventional and requires careful attention on the part of the optimizer implementor to ensure that legal operator trees are not accidently left out of the search space. We have used Volcano as our back-end search engine.

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.

We believe that rule-based query optimizers will be standard tools of future database systems. The pragmatic difficulties of using existing rule-based optimizers led us to develop Prairie, an extensible and structured algebraic framework for specifying rules. Prairie is similar to existing optimizers in that it supports both transformation rules and implementation rules. However, Prairie makes several improvements:

1. it offers a conceptually more streamlined model for rule specification;

2. rules are encapsulated, there are no "hidden" operators or "hidden" algorithms;

3. implementation hints (e.g., enforcers) are deduced automatically;

4. and it has efficient implementations.

We have explained how the first three points are important for simplifying rule specifications and making rule sets less brittle for extensibility. A consequence is that Prairie rules are simpler and more robust than rules of existing optimizers (e.g., Volcano). We addressed the fourth point by building a P2V pre-processor which uses sophisticated algorithms to compose and compact a Prairie rule set into a Volcano rule set. To demonstrate the scalability of our approach, we rewrote the TI Open OODB rule set as a Prairie rule set, generated its Volcano counterpart, and showed that the performance of the synthesized Volcano rule set closely matches the hand-crafted Volcano rule set.

Our future work will concentrate on developing higher-level abstractions using Prairie, including automatically generating Prairie rule sets, and combining multiple Prairie rule sets to automatically generate efficient optimizers.

Acknowledgments

We wish to thank Texas Instruments, Inc. for making the Open OODB source code available to us. Comments by José Blakeley, Anne Ngu, Vivek Singhal, Thomas Woo and the anonymous referees greatly improved the quality of the paper.

References

[1] D. S. Batory. Building blocks of database management systems. Technical Report TR–87–23, The University of Texas at Austin, February 1988.

[2] 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.

[3] Dean Daniels, Patricia 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.

[4] Dinesh Das and Don Batory. Prairie: An algebraic framework for rule specification in query optimizers. In *Proceedings of the Workshop on Database Query Optimizer Generators and Rule-Based Optimizers*, pages 139–154, Dallas, September 1993.

[5] Dinesh Das and Don Batory. Prairie: A rule specification framework for query optimizers. Technical Report TR 94–16, The University of Texas at Austin, May 1994.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, Boston, May 1979.

[12] Michael Stonebraker and Lawrence A. Rowe. The design of Postgres. In *Proceedings 1986 ACM SIGMOD International Conference on Management of Data*, pages 340–355, Washington, May 1986.