Making Database Optimizers More Extensible

by

Dinesh Das, B.Tech., M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 1995

Making Database Optimizers More Extensible

Approved by
Dissertation Committee:

Acknowledgments

I am deeply indebted to my advisor, Don Batory, for many years of encouragement and advice, for providing constant direction and focus to my research, and for reading countless drafts of my dissertation. I am a much better researcher because of his excellent guidance.

I am also thankful to my committee members, Professors Don Fussell, Dan Miranker, and Avi Silberschatz, and, especially, to Dr. José Blakeley for their meticulous reading of my dissertation and probing questions.

To my friends Eleonora Drakou, G. Neelakantan Kartha, Sharad Mehrotra, Rajeev Rastogi, Probal Shome, Nandit Soparkar, Thomas Woo, and Yogesh Wagle, I owe a special round of thanks for the many hours of late-night philosophizing, pleasant conservations, and perspective on life. Vivek Singhal, Jeff Thomas, and Millie Villareal provided a very enjoyable working environment. Finally, my parents were a constant source of support and encouragement.

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

DINESH DAS

The University of Texas at Austin
May 1995

Making Database Optimizers More Extensible

Publication No. ____________________

Dinesh Das, Ph.D.
The University of Texas at Austin, 1995

Supervisor: Don Batory

Query optimizers are fundamental components of database management systems (DBMSs). An optimizer consists of three features: a search space, a cost model, and a search strategy. The experience of many researchers has shown that hard-wiring these features results in an optimizer that is very inflexible and difficult to modify.

Rule-based optimizers have been developed to alleviate some of the problems of monolithic optimizers. Unfortunately, contemporary rule-based optimizers do not provide enough support to enable database implementors (DBI) to fully realize the potential of open systems. We have identified four requirements that a rule-based optimizer should satisfy to address these needs. First, rules should be specified using high-level abstractions, insulating the DBI from underlying implementation details. Second, rule sets should be easily extensible, with a minimum of reprogramming required. Third, rule sets should be easily reconfigurable, that is, changeable to meet a variety of user needs, interfaces, database schemas, etc. Fourth, rule-based optimizers should be fast, that is, performance should not be sacrificed for the sake of high-level specifications.

In this dissertation, we describe Prairie, an environment for specifying rules for rule-based optimizers that satisfies all four of the above requirements. The Prairie specification language is presented and we show how it allows a DBI to design an easily extensible rule set for a rule-based optimizer. Experimental results are presented using the Texas Instruments Open OODB optimizer rule set to validate the claim of good performance using Prairie. Finally, a building blocks approach of constructing rule sets is presented; this results in easily reconfigurable rule sets whose features are changeable simply by assembling the blocks in various ways.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Database management systems (DBMSs) are basic tools for information storage and retrieval. A well-designed and implemented DBMS can not only act as a data repository, but also facilitate efficient querying and gathering information about the stored data. A good user interface is critical for this purpose.

Query processing is a fundamental part of DBMSs. It is the process of retrieving data that match user-specified requirements. Figure 1.1 shows the three basic steps in evaluating a query. A compiler parses and translates a query (expressed in a high-level language like SQL) into a representation known as an operator tree. A query optimizer then transforms this operator tree into an access plan. Finally, the execution module executes the access plan to return the results of the SQL query.

As shown in Figure 1.1, query optimization [29, 34, 35, 51] is an important step in query processing. 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.

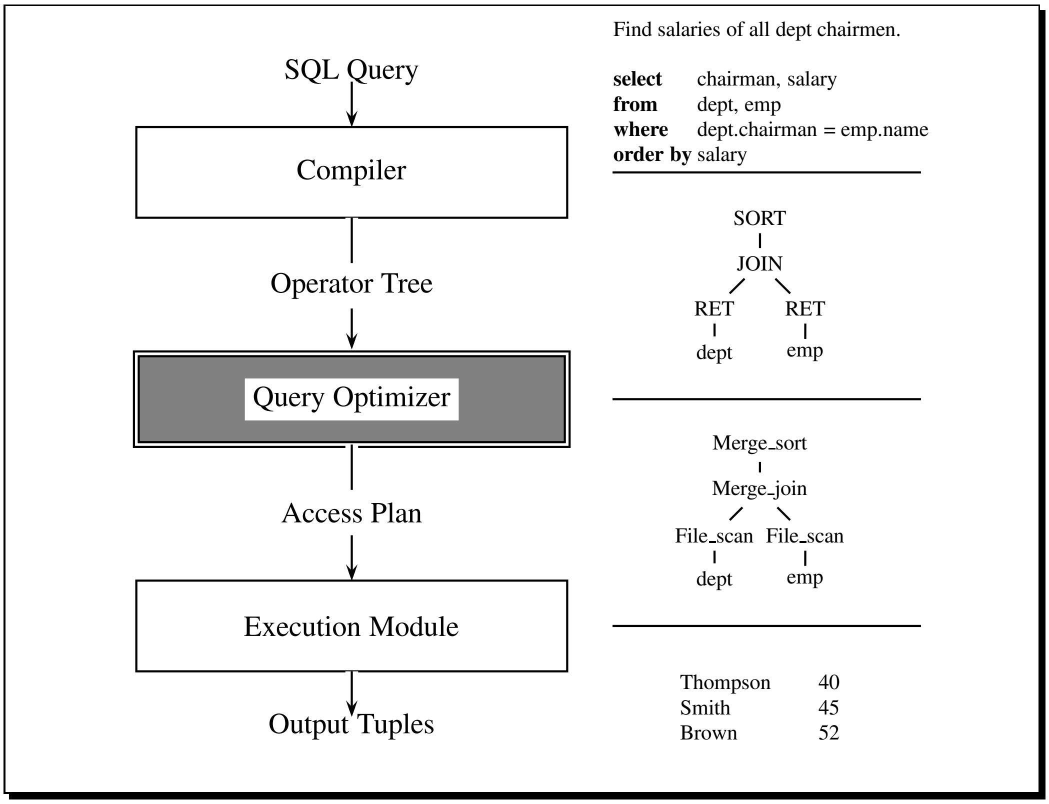


Figure 1.1: Query processing in a database system and an example

The *search space* is the set of access plans that can evaluate a query. All plans of 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. If the search space is small, a typical search strategy is to enumerate and compare the costs of all plans against one another. However, most search spaces, even for simple queries, are enormous, and thus query optimizers often need heuristics to control the number of plans to be examined.

Query optimizers have traditionally been built as monolithic subsystems of DBMSs.

This simply reflects the fact that traditional database systems are themselves monolithic: the algorithms that are used to store and retrieve 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 EXODUS [17], Starburst [43], Genesis [4], and Postgres [49]. Rule-based query optimizers [25, 28, 30, 31, 33, 36] are among the major conceptual advances that have been proposed to deal with query optimizer extensibility. A primary advantage of rule-based optimizers over traditional ones is the ability to introduce new functionality in a query algebra without changing the optimization algorithm. We will review some well-known query optimizers, both traditional and rule-based, later in this chapter.

DBMSs are increasingly being used to store and retrieve not only larger amounts of information, but more novel types of data as well (object-oriented, multimedia, etc.). To ensure that a DBMS scales well to these emerging needs, it is important to better design and implement "open" DBMSs which have well-designed components with clearly defined interfaces that are easily adaptable. Since optimizers are critical components of DBMSs, it is necessary to ensure that they meet four specific goals:

- **Abstractions.** Optimizers should be constructed using clearly defined abstractions that encapsulate fundamental concepts of optimizers. That is, the abstractions should represent the distinct steps inherent in an optimization. Moreover, these abstractions should be sufficiently high-level that a change in their implementation does not fundamentally change the design of the optimizer. This has the advantage that any changes to an optimizer consists of changing the *implementation* of abstractions, not the abstractions themselves.

- **Extensibility.** As mentioned earlier, optimizers in next-generation DBMSs will increasingly be required to deal with a wider range of data operators and data types. It is imperative, therefore, that optimizers should be designed and constructed in a manner that is amenable to easy and quick changes. This property is known as *extensibility*. Abstractions mentioned above help in the *conceptual* design of an optimizer. The *framework* used to build (i.e., specify) optimizers should also ensure that the constructs used define interfaces which closely represent the abstractions defined above.

- **Performance.** Optimizers generate a "good" access plan for a user query. This metric is defined by the cost model of the optimizer, and measures the estimated resources used by the execution module (see Figure 1.1) to process the query. It is also important that an optimal access plan be found efficiently, i.e., that the optimizer be fast. This, in turn, requires that the abstractions embodied in the optimizer specification have efficient implementations.

- **Reconfigurability.** To be able to optimize a wider and more diverse set of queries, and to facilitate easy and *seamless* changes to an existing optimizer, optimizers should be specified using *building-blocks* that can be arranged in various ways to construct an optimizer. These building-blocks can be used to encapsulate abstractions. This means that an optimizer can be changed quickly simply by changing the arrangement of the building-blocks, or by changing the abstractions encapsulated in an existing building-block.

In this dissertation, we propose a well-defined, algebraic framework, called Prairie, for specifying rules in a query optimizer that meets all of the four goals listed above. The algebra that we propose is similar to the rule specification languages in Starburst [36] and Volcano [31], but provides a cleaner abstraction of the actions of an optimizer; as such, it is much easier to write and read rules in our proposed model. The algebra allows a database implementor (DBI) to specify transformations of a query using rewrite rules that may have conditions. The rules determine the search space and cost model of the optimizer. We do not propose a search strategy; we intend to implement a preprocessor that can translate rules in our model to those in Volcano, since Volcano has an efficient search strategy and is freely available.

Below, we briefly review some related work on optimizers.

1.2 Related Work

1.2.1 Traditional Query Optimizers

The System R optimizer [46] was one of the earliest query optimizers proposed and implemented. It was built for the System R database system [1]. System R is a centralized, relational DBMS where users specify their queries in SQL. The System R optimizer is still *the de facto* industry standard; it was the first to show the practical viability of query optimization in a commercial setting.

The basic philosophy of the System R optimizer was a bottom-up exhaustive search strategy with dynamic programming. Some of the salient features it embodied are listed below:

- It employed a bottom-up strategy. That is, children of nodes in an operator tree are optimized before the node itself is optimized.

- It used two join algorithms, nested loops and merge join (first introduced in [16]), and two relation retrieval algorithms, segment scan and variations of index scan.

- The concept of "interesting" orders was introduced to generate only those access plans in the search space that were likely to be part of other access plans. It also limited the search space by considering only left-deep operator trees (in which the inner relation was always a stored file), and by delaying cross-products as far as possible.

- It introduced a fairly elaborate scheme using "selectivity factors" to estimate cardinalities of streams generated by computations on other streams.

- Dynamic programming was used to control the expansion of the search space. Basically, the optimizer maintained an equivalence class of access plans, and as each plan was generated, its cost was computed, and if its cost was greater than the minimum cost of any plan in its equivalence class, it was discarded since it would not be a subplan of another optimal access plan. This process ensured that the optimization time was exponential (in the number of joining relations) as opposed to a factorial time complexity (see Appendix A for a proof of the algorithm complexity).

- The System R optimizer also had a fairly elaborate cost model involving a weighted sum of CPU and I/O costs.

R^* is a distributed relative of System R. The stored relations are located at distributed sites. R^*'s query processor [20, 37, 45] works in essentially the same way as that of System R, except for some subtle complications introduced by the distribution of relations. These complications arise mostly in authentication and catalog sharing between remote sites. Conceptually, however, the R^* optimizer builds upon the System R optimizer.

Like System R, the search strategy in R^*'s optimizer also employs an exhaustive examination of its search space to find an optimal plan. Heuristics are used to limit the space. The retrieval methods available are segment scan and index scan. Single site joins (i.e., joins in which both streams are located at the same site) are optimized in the same way as in System R using nested loops or merge join.

The case of joins in which the two input streams are located at different sites is what distinguishes R^* from System R. R^* handles this case by transfering both streams to a common site before joining. Two transfer strategies are considered, tuple-at-a-time and whole.

The cost model used by R^*'s optimizer is similar to System R, except for multi-site joins in which case the cost of transfering relations is also added.

1.2.2 Rule-Based Optimizers — The Next Generation

Both System R and R^* have existed for a long time. Lately, however, researchers have been looking at extensible query optimizers. This is in keeping with the trend toward constructing extensible DBMSs. Extensibility, in short, is the process of augmenting or removing features easily from a system in order to customize it for an application. Extensibility of query optimizers refers to the ease of constructing optimizers for extensible DBMSs. It also refers to the easy customizability of an existing query optimizer to a new application.

Rule-based query optimizers have been proposed as a means of constructing extensible optimizers. The primary advantage of rule-based query optimizers is the ability to add new operators and algorithms without a costly rewrite of the entire optimizer. Below, we review a few of the more well-known rule-based optimizers.

The Starburst query optimizer [33, 36, 43] 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 transformations. 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. As the designers of Starburst point out, 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. Also, the rewrite phase uses heuristics to prune the search space before the optimization phase; this can lead to a sub-optimal plan.

Lohman describes rules for the optimization phase of Starburst [36]. These rules represent alternative access paths, join algorithms, and site choices in a distributed DBMS. However, even though these rules transform an operator tree into a valid access plan, the cost computation is not done until all rules are applied. In other words, the rewrite rules of Starburst are purely syntactic transformations of one query representation into another. Thus, all operator trees are subjected to all applicable rules before costs are computed and the search space can be pruned.

Freytag [25] 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. In addition, as in the rewrite phase of Starburst, he doesn't consider the cost transformations inherent in any query transformation; rules are syntactic transformation rules.

The EXODUS project [17] has similar goals as those of Starburst, to provide a framework in which DBMSs can be easily implemented as extensions of existing DBMSs, or to design completely new DBMSs. The query optimizer in EXODUS [28, 30] is, in fact, an *optimizer generator* which accepts the specification of the data model and operators in a description file. 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 DBI the freedom to associate any action with a particular rule.

Operator trees in EXODUS are constructed bottom-up from previously constructed sub-trees. Common sub-expressions are shared as far as possible. Each access plan in the search space has a cost factor associated with it; plans are examined based on their cost factors. It is not clear if these cost factors have any relation to the actual costs of the plans (as estimated by the cost model), and if so, what the relation is. The EXODUS optimizer uses the cost factor with an exhaustive search strategy to guide the exploration of access plans.

The Volcano optimizer generator project [31] 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 DBI to ensure that no valid operator trees are accidently left out of the search space. We will discuss the Volcano optimizer generator in greater detail in Chapter 2.

Fegaras, Maier, and Sheard [24] describe a declarative optimizer generator framework for specifying query optimizers. The premise in their work is that much of the specification in current optimizers consists of procedurally defined actions; making these actions declarative results in a cleaner specification language. To this end, Fegaras et al use a reflective functional programming language, called CRML, as the basis for their specification language. The term reflective describes an environment that not only can specify an optimizer, but also enables a DBI to embed metadata (or parameters) to guide the optimizer generator in generating an optimizer for a specific target. An optimizer is specified by using rewrite rules that are based on pattern matching. In addition to this syntactic transformation, rules also consist of semantic context-dependent conditions. However, the framework described by Fegaras et al still contains some implementation-level details at the specification level. These drawbacks parallel those in Volcano (which is described in more detail in Chapter 2), and mainly concern the representation and transformation, using rewrite rules, of the various expressions and their abstractions.

1.3 Outline of Dissertation

All of the rule-based query optimizers discussed in Section 1.2.2 take an *ad hoc* approach to the specification of rules. We introduce a *well-defined* and *structured algebra* called Prairie to specify rules for a rule-based optimizer. Because of a rigorous algebra, it is easier for a DBI to write rules, as well as for readers to read and understand the semantics of the rules. However, rules by themselves do not constitute an optimizer; we need a search strategy also. Since search strategies are well-understood, we do not propose to study them. Instead, we will use the Volcano search engine to drive our optimizer, since Volcano has a very efficient search strategy. However, this requires that we translate Prairie rules into Volcano rules.

The Volcano optimizer generator is described in Chapter 2. We describe Prairie in Chapter 3 and show how it can be used to specify optimizers using high-level abstractions. Chapter 4 describes the process of translating Prairie rules into Volcano rules. Chapter 5 presents some experimental results validating the efficiency of Prairie optimizers. Chapter 6 describes how layered optimizers can be built using Prairie, and how this leads to easier reconfigurability. Finally, we end with some conclusions and future work in Chapter 7.

Chapter 2

The Volcano Optimizer Generator

This chapter describes the Volcano optimizer generator. The features that are relevant to this dissertation are presented, but more details can be found in [31, 38].

2.1 Overview

Volcano is a rule-based query optimizer generator that is designed to be flexible and extensible to specific database architectures. (Henceforth, when we talk of Volcano, we are referring to the "Volcano optimizer generator".) It implements a top-down query optimizer in the sense that parents of nodes in an operator tree are optimized before the node itself is

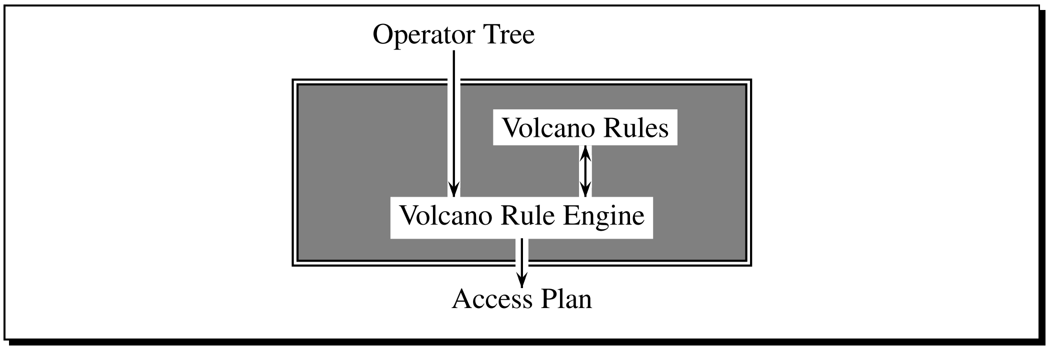


Figure 2.1: Schematic representation of the Volcano optimizer generator

optimized. Roughly speaking, Volcano provides two major components of an optimizer: a search engine and a rule specification language. The schematic design of the Volcano optimizer generator is depicted in Figure 2.1. A DBI writes rules in Volcano's specification language, which are then compiled with the rule engine to generate an optimizer.

The search engine is hard-coded and is not changeable by the DBI. The search strategy is exhaustive, meaning that all operator trees that are generated by application of rules are evaluated before an optimal plan is returned. Dynamic programming is used to prune the search space as much as possible. Since this process of pruning is similar to the one in System R [46], we will not describe it further.

The rule specification language is the part of Volcano that allows a DBI to specify how operator trees are transformed to generate access plans. The optimizer is specified as a set of operators, algorithms, and rules with associated actions. Viewed as a state-transformation operation, the optimizer transforms an initial state (operator tree) into a final one (access plan) while also translating associated state information in the process. The following sections describe the Volcano rule specification language in greater detail.

2.2 Definitions

Before we describe the Volcano rule specification language, we need a few definitions.

Stored Files and Streams. A relation or file is *stored* if its tuples reside on disk. 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 are either named, denoted by $?n$, where n is an integer, or unnamed operator trees (defined below). In Volcano, all operations accept zero or more streams as input, or one or more stored files as input.

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

Operators. Abstract *operators* specify computations on streams or stored files; they are denoted in this dissertation by all capital letters (e.g., JOIN). Associated with every operator is an *operator argument* which specifies additional information needed to execute the operator. The type of an operator argument can be virtually anything; Volcano allows it to be defined as an arbitrary C struct. As examples, some operators are described below; for each, we describe what the operator argument might be.

- RET retrieves tuples of a stored file. The operator argument might specify the name of the stored file to be retrieved.
- JOIN joins two streams. The operator argument of JOIN could specify the join predicate. In our examples, we will assume that the operator argument for JOIN specifies an equijoin predicate of the form $a = b$ where a is an attribute of the outer stream and b is an attribute of the inner stream.

Algorithms. *Algorithms* are concrete implementations of abstract operators; they are represented in lower case with the first letter capitalized (e.g., Nested_loops). A single operator can be implemented by several algorithms, and a single algorithm can implement many operators. Corresponding to the operator arguments of operators, algorithms have *algorithm arguments*. In many cases, algorithm arguments are the same as the operator arguments of the operators that they implement; however, sometimes algorithms don't implement any particular operator (see below), so, in general, algorithm arguments are different from operator arguments.

Enforcers. *Enforcers* are special algorithms that are not implementations of any particular operator; rather, they are algorithms that can accept their input from other algorithms and return an output that can be fed to other algorithms or enforcers.

Operator	Description	Operator/Algorithm Argument	Algorithm
JOIN	Join two streams	join_predicate	Nested_loops
			Merge_join
RET	Retrieve stored file	relation name	File_scan
			Index_scan
	Sort stream	tuple order	Merge_sort

Table 2.1: Operators and algorithms in a centralized Volcano query optimizer and their operator/algorithm arguments. Note that Merge_sort does not implement any operator and is an enforcer.

Table 2.1 lists some operators and algorithms implementing them together with their operator/algorithm arguments. Note that Merge_sort does not implement any particular operator because it is actually an enforcer. These operators, algorithms, and enforcers will be used in subsequent examples in this chapter.

Operator Tree. An *operator tree* is a rooted tree whose non-leaf, or *interior*, nodes are database operations (operators, algorithms, or enforcers) and whose leaf nodes are stored files. Operator trees (also called *expressions*) are represented in LISP-like prefix notation form.

EXAMPLE 1. The expression,

(JOIN ?op_arg1 ((RET ?op_arg2 ()) (RET ?op_arg3 ()))

denotes a stream that first RETrieves two stored relations, and then JOINs them. ?op_arg2 and ?op_arg3 are the operator arguments of the two RETs respectively, and are the names of the stored files to be retrieved. ?op_arg1 is the operator argument of the JOIN operator and denotes the join predicate. □

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

Properties. *Properties* are information associated with each node in an operator tree. Each node has a specific value for every property associated with it, and the complete property set specifies a node uniquely. In Volcano, in addition to the operator/algorithm arguments that we have seen above, there are four sets of properties:

- **Logical Properties.** *Logical properties* are those properties of a node that can be uniquely determined *prior* to optimization. Logical properties of an abstract operator are computed bottom-up, i.e., from the leaves of an operator tree. The logical properties of an algorithm are the same as the logical properties of the abstract operator that it implements, and the logical properties of an enforcer are the same as those of its input. For instance, the list of attributes of a stored file or a stream (without projections) can be determined from database catalogs before optimization.

- **System Properties.** *System properties* are a special class of logical properties, consisting of the two properties "cardinality" and "record_width". For all practical purposes, these two properties are exactly like logical properties. The rationale for treating them differently is not clearly defined in the Volcano literature.

- **Physical Properties.** *Physical properties* are properties associated with the data produced by an algorithm or enforcer. Physical properties are propagated bottom-up, i.e., physical properties of a node are computed as a function of the properties (logical, system, and physical) of its inputs.

- **Cost.** This property represents the cost of a node; it has a meaningful value only for algorithm and enforcer nodes. Cost is not a logical or system property since its value cannot be determined solely from abstract operators.[1]

The DBI has to specify the list of properties, and the type (logical, system, physical, or cost) of each. Generally speaking, if the property value of *each* node in an operator tree can be

[1] In many real-world query optimizers (e.g., System R [46] and R* [45]), the cost of a node is a function of the costs of its inputs. Thus, in Volcano's terminology, cost should really be a physical property. However, Volcano treats cost as a fourth category of property.

Property	Type	Description
tuple_order	Physical	tuple order of resulting stream, DONT_CARE if none
cardinality	System	number of tuples of resulting stream
record_width	System	size of individual tuple in stream
attributes	Logical	list of attributes
cost	Cost	estimated cost of algorithm

Table 2.2: Properties of nodes in an operator tree in Volcano

determined before any rules are applied to it, it is a logical property (or system property if it is one of two special properties), and if a property value can be determined only when the node is an algorithm or an enforcer, it is a physical property. If the property computes the relative merit of an access plan, then the property is the cost. Furthermore, logical, system, and physical properties are computed bottom-up; the difference is that logical and system properties are determined prior to optimization, whereas physical properties are determined after a tree (or subtree) has been optimized.

It is worth noting that some properties (such as attributes of a stream) can be either physical or logical depending on the semantics of the operators in the database schema. Thus, altering the semantics of an optimizer requires the DBI to re-examine the partition of properties. This, as it turns out, is problematic for Volcano optimizer designers.

Table 2.2 lists the different properties and their types that we will use in our examples. The only logical property is the list of "attributes", and the system properties are "cardinality" and "record_width". The single physical property is the "tuple_order" of the output stream. The cost property will be called "cost".

Constraints. *Constraints* are requirements on physical property values that are imposed on an operator tree. An access plan for an operator tree is acceptable if and only if it satisfies the constraints that are imposed upon it. There are two types of constraints in Volcano. The first type is represented by a *needed property vector*. This specifies the list of physical property values that an access plan *must* have to be an acceptable plan. The second type of constraint is represented by an *excluded property vector* and it specifies a list of physical property values that a plan *must not* have to be acceptable. Each node in an operator tree may have different constraints on it; however, it is the constraints on the root of an operator tree that must be satisfied for an access plan to be acceptable.

EXAMPLE 2. Consider the expression,

(JOIN ?op_arg1 ((RET ?op_arg2 ()) (RET ?op_arg3 ()))

to be optimized. If "need_pv" denotes the needed property vector, and "excl_pv" denotes the excluded property vector, then we can specify the constraints by setting appropriate values for the physical properties in need_pv and excl_pv. Thus, setting

need_pv.tuple_order = DONT_CARE
excl_pv.tuple_order = b

specifies that any access plan that implements the JOIN expression above can return tuples in any order (DONT_CARE), as long as they are not in b order (b has to be an attribute of at least one of the two relations RETrieved). An access plan that does not satisfy these constraints would not be a valid implementation of the operator tree. □

Volcano uses both needed property vectors and excluded property vectors to select a plan. In the above example, suppose there exists an access plan of the given tree with a tuple_order of b. If we only looked at the needed property vector, we would be inclined to accept this plan, since the needed property vector specifies that the plan return a stream in DONT_CARE order (i.e., any order). However, the excluded property vector specifies that this plan is not acceptable.

2.3 Volcano Optimization Paradigm

Volcano employs a top-down query optimization paradigm that rewrites operator trees starting from the root. Parents are considered before children for optimization. Informally, abstract operators are transformed into algorithms top-down until an access plan is obtained.

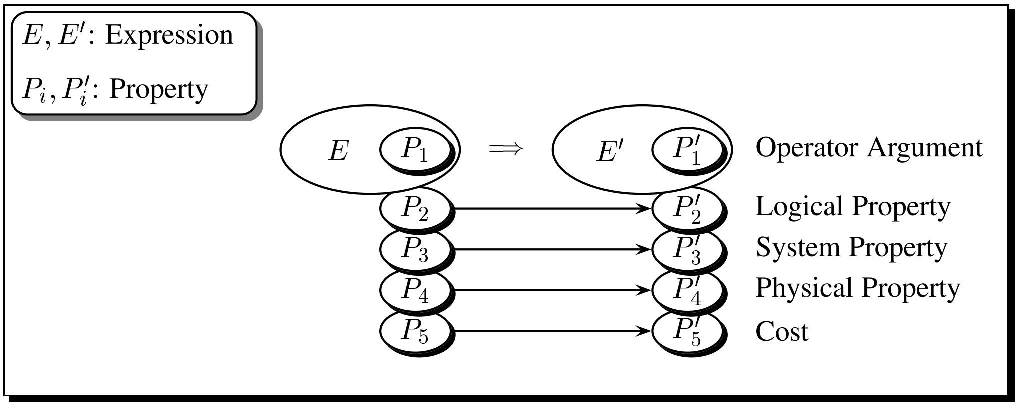


Figure 2.2: General form of a Volcano rule. Single arrows denote one or more DBI-defined support functions. The properties that are translated by the rule or by support functions are listed on the right.

The search engine provided by Volcano uses dynamic programming to prune the search space by discarding operator trees that are determined to lead to a sub-optimal access plan.

Before the start of optimization, certain properties of the original operator tree are initialized. As described in Section 2.2, logical and system properties of nodes of an operator tree can be determined prior to optimization. The initialization of these properties for each subtree is done by DBI-defined functions; these are defined in Section 2.7.

2.4 Rules in Volcano

Rules in Volcano correspond to rewrites between pairs of expressions, or between an expression and an access plan. Figure 2.2 shows the general format of a Volcano rule. This general rule results in two types of transformations (or *rewrite rules*) in Volcano: transformation rules and implementation rules. Each rule transforms an expression into another based on additional conditions; the transformation also triggers execution of other DBI-defined functions that results in a mapping of properties between expressions. This is shown clearly in Figure 2.2, where the single arrows represent functions executed as a result of application of a Volcano rule. Transformation and implementation rules are defined precisely in Sec-

```
(E ?op_arg1 (?1 ...?n)) → (E' ?op_arg2 (?1 ...?n))                    (2.1)
%cond_code
{{
        test
}}
%appl_code
{{
        post-test statements
}}
```

Figure 2.3: General form of a transformation rule in Volcano

tions 2.5 and 2.6 and are illustrated with examples. The examples are chosen from rules that would be used in a centralized relational query optimizer; the operators, algorithms, and enforcers are subsets of those in Table 2.1. DBI-defined functions are described in more detail in Section 2.7.

2.5 Transformation Rules

Transformation rules, or trans_rules, in Volcano define mappings from one operator tree to another. Let E and E' be expressions that involve only abstract operators. Equation (2.1) (shown in Figure 2.3) shows the general form of a trans_rule in Volcano. The expression E on the left side is transformed into the expression E' on the right side. The actions of a trans_rule define equivalences between the operator arguments of the input expression E with the operator arguments of the output expression E'. A test is needed to determine if the transformation is indeed applicable.

The first part of the actions associated with a trans_rule is the test, called *condition code* in Volcano. This is any arbitrary section of C code[2] that tests whether the transformation rule can indeed be applied to the expression E. The test can reference the operator arguments of the expressions on the left side and/or the logical or system properties of any of the input streams. If the rule does not apply, then the expression E is left unchanged.

If the trans_rule does apply to E, then post-test actions, called *application code* in

[2]The condition code doesn't return a boolean value. It succeeds if a REJECT statement is not processed, and fails otherwise.

```
(JOIN ?op_arg5 ((JOIN ?op_arg4 (?1 ?2)) ?3)) → (JOIN ?op_arg7 (?1 (JOIN ?op_arg6 (?2 ?3))))    (2.2)
%cond_code
{{
        ?op_arg6 = predicate_intersect (?2, ?op_arg5) ;
        if (empty (?op_arg6)) REJECT ;
}}
%appl_code
{{
        ?op_arg7 = ?op_arg4 ;
}}
```

Figure 2.4: Join associativity transformation rule

Volcano, are executed. This is an arbitrary piece of C code that sets the operator argument of the expression E' on the right side. The application code is executed immediately if the condition code is satisfied.

As mentioned in Section 2.3, a trans_rule in Volcano triggers two DBI-defined functions for determining the logical and system properties of any new expressions obtained by application of the rule. Since a trans_rule only transforms logical expressions, physical properties and cost (as shown in Figure 2.2) are not transformed in a trans_rule.

EXAMPLE 3. The associativity of JOINs is expressed by trans_rule (2.2) in Figure 2.4. The condition code of trans_rule (2.2) determines the operator argument (i.e., the join predicate) of the inner join on the right side. If it is empty, implying a cross-product, the rule is rejected, and the expression on the left side remains unchanged. If, however, the test is successful, the application code assigns the operator argument (join predicate) of the outer join on the right side. Logical and system properties of the new subexpressions on the right side of Equation 2.2 are computed by DBI-defined functions, externally to the rule. □

2.6 Implementation Rules

Implementation rules, or impl_rules, in Volcano specify implementations of abstract operators. Let O be an operator and A be an algorithm that implements O. Equation (2.3) (shown in Figure 2.5) shows the general form of an impl_rule in Volcano. A test is needed

```
(E ?op_arg1 (?1 ...?n)) → (A ?al_arg1 (?1 ...?n))                    (2.3)
%cond_code
{{
        test
}}
%appl_code
{{
        post-test statements
}}
```

Figure 2.5: General form of an implementation rule in Volcano

to determine if the implementation rule is indeed applicable.

The actions associated with an impl_rule are specified in two parts, as in a trans_rule. The first part, the test, called *condition code*, is used to test if the rule applies to the expression on the left side. It is an arbitrary piece of C code that references the operator arguments and/or any logical or system properties of subexpressions on the left side. As in trans_rules, the REJECT keyword is used to reject the rule. However, unlike trans_rules, impl_rules have to satisfy another test *in addition* to the one in the cond_code section; this extra test is specified by a DBI-written support function (called do_any_good) described in Section 2.7.

The second part of the actions, the post-test statements, called *application code*, is executed immediately if the condition code is satisfied. The application code sets the algorithm argument of the algorithm on the right side.

The cost of using algorithm A for the operator O in (2.3) is computed by a DBI-defined function for algorithm A (this function is described in Section 2.7). The cost of an algorithm node in Volcano is implicitly assumed to be the sum of costs of all its inputs plus an additional term that is a function *only* of the logical and system properties of A and its inputs.

EXAMPLE 4. Impl_rule (2.4) in Figure 2.6 shows an implementation rule. It selects Nested_loops as an implementation for the JOIN operator. The condition code of this rule is empty, implying that the rule is applied immediately. The application code sets the algorithm argument (i.e., the join predicate) of the Nested_loops node to be the same as the

```
(JOIN ?op_arg1 (?1 ?2)) → (Nested_loops ?al_arg1 (?1 ?2))                                    (2.4)
%cond_code
{{

}}
%appl_code
{{
        ?al_arg1 = ?op_arg1 ;
}}
```

Figure 2.6: Nested loops implementation rule

operator argument (join predicate) of the JOIN node. □

2.7 Support Functions

As mentioned earlier, and as can be seen from Figure 2.2, most of the actions and property computations in Volcano rules are done by *support* functions defined by the DBI. These functions are defined for each operator, algorithm, or enforcer and are triggered upon application of rules. Some are executed only when trans_rules are applied, some when impl_rules are applied, and others when neither is applied (i.e., enforcers).

The required support functions that appear in a Volcano optimizer are summarized in Table 2.3.

2.8 Drawbacks of Volcano

In the previous sections, we have seen how Volcano uses abstractions to allow a DBI to specify a rule-based query optimizer. However, these abstractions are not sufficiently high-level to insulate the DBI from implementation-level details. Moreover, some of the constructs in the rule specification language are motivated not by *conceptual* details of the optimizer, but by *implementation* details with an eye toward constructing efficient optimizers. The result is often an optimizer that is fast, but often quite brittle and inflexible. In this section, we describe some of the concepts in Volcano that are bottlenecks to a clean specification.

Function	Frequency	Description
derive_log_prop	Operator	Compute logical properties of node given logical properties of children
derive_sys_prop	Operator	Compute system properties of node given system properties of children
derive_phy_prop	Algorithm Enforcer	Compute physical properties of node given physical properties of children
get_input_pv	Algorithm Enforcer	Compute needed physical properties of node given needed physical properties of parent
do_any_good	Algorithm Enforcer	Test whether algorithm or enforcer applies given needed physical properties
cost	Algorithm Enforcer	Compute cost of algorithm or enforcer

Table 2.3: Support functions in a Volcano optimizer. The frequency column denotes whether a function is defined for each operator, algorithm, or enforcer.

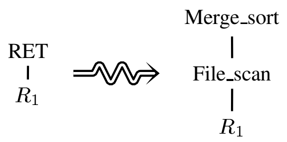


Figure 2.7: An implicit rule in Volcano. Merge_sort is an enforcer.

2.8.1 Explicit vs. Implicit Rules

The general form of a Volcano rule is shown in Figure 2.2. This figure shows that Volcano rules transform an abstract expression into either an abstract expression or an algorithm. Such rules are called *explicit rules*. An example of an explicit rule is the impl_rule 2.4 in Figure 2.6. An explicit rule involves abstract operators, and sometimes, algorithms. However, there is another class of transformations, *implicit rules*, that don't have the form shown in Figure 2.2. Such transformations typically involve enforcers since enforcers are not implementations of any specific operators (e.g., Merge_sort in Table 2.1).

Consider the transformation in Figure 2.7. File_scan is an implementation for RET and assume there is an impl_rule for this transformation. However, since Merge_sort is an enforcer, there is no rule involving it. Nevertheless, the transformation shown in Figure 2.7 is a valid rewrite in Volcano. This is an example of an implicit rule.

Implicit rules are present in Volcano for efficiency and to aid in generating an optimizer. However, since enforcers (i.e., algorithms in implicit rules) behave like algorithms for all practical purposes, their separate classification introduces an additional detail that creates an asymmetric framework of rewrite rules. In other words, transformations (both between operator trees and their associated property sets) are different for explicit and implicit rules. This, in turn, leads to two other problems:

- Are new operators, algorithms, or rules explicit or implicit? That is, when an existing Volcano optimizer is modified by adding operators, algorithms, or rules, the DBI has to re-examine the *entire* new schema (i.e., operators, algorithms, enforcers, properties, rules) to determine which rules are explicit and which are implicit.

- Since implicit rules do not have the same form as explicit rules (i.e., the one shown in Figure 2.2), property transformations in implicit rules are also different from those in explicit rules. As before, this means that extensibility of an optimizer is accomplished differently when explicit rules are added than when implicit rules are introduced.

The solutions to these problems is obvious: treat all operators and algorithms as first-class objects. This implies that enforcers are treated just like regular algorithms, with explicit rules implementing them, and with similar property transformation mechanisms.

2.8.2 Property Representation and Transformation

Properties are crucial for storing state (operator tree) information, and for guiding the search. When an operator tree is transformed into another, so are the properties associated with the operator tree. The more separate property structures, the greater the number of property transformations. As can be seen from Figure 2.2, most of the property translations in Volcano rules are done by DBI-defined functions (as described in Section 2.7). In fact, this general form of a rule results in two distinctly different forms for trans_rules and impl_rules in Volcano as shown in Figure 2.8. This results in a somewhat large number of support functions which often carry a greater burden of property transformations than the

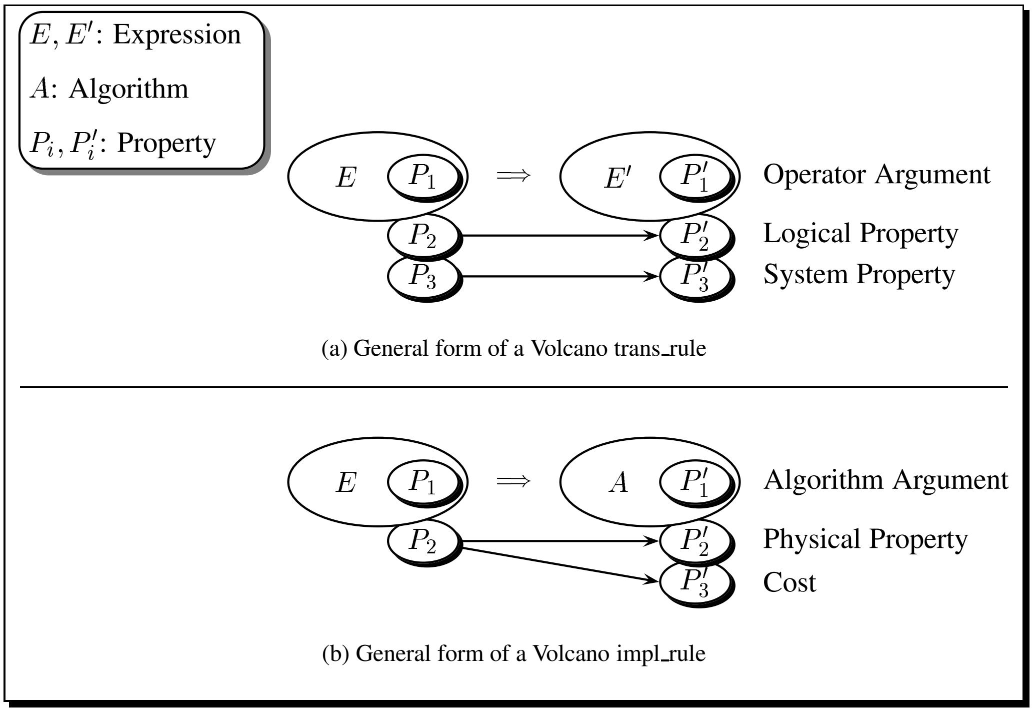


Figure 2.8: General form of Volcano trans_rules and impl_rules (cf. Figure 2.2). Single arrows denote one or more DBI-defined support functions. The properties that are translated by the rule or by support functions are listed on the right.

rules themselves. This decidedly asymmetric manner of treating trans_rules and impl_rules and the large number of property structures leads to a few problems:

- Deciding which set a particular property belongs to is implementation-dependent.
- Modifying any existing operator, algorithm, enforcer, rule, or property might entail a repartitioning of the property sets. This can lead to a cascading effect of having to modify the large number of support functions — hardly conducive to easy extensibility.
- Extensibility in Volcano is complicated by its extensive use of support functions for transformations of its multiple property sets. For example, from Figure 2.3, we can see that addition of a single algorithm requires the DBI to define *four* new support

functions for property transformations. Even addition of an operator requires defining two new support functions.

Property partitions are geared toward generating efficient optimizers. However, as we have seen, they result in poor extensibility. The solution to this problem is also obvious: treat all properties as equivalent, with property transformations being done at the rule level (rather than a plethora of support functions). However, noting that property partitions *do* result in fast optimizers, a preprocessor should be able to automatically partition property sets based on need.

2.9 Summary

In this chapter, we have seen the rule specification framework for Volcano. Volcano is the only public-domain rule-based query optimizer. (To our knowledge, there are two other rule-based optimizers, the EXODUS optimizer generator which is an earlier version of Volcano, and the Starburst rule-based optimizer. The Starburst optimizer is not publicly available.)

Volcano provides an efficient search strategy for optimizing queries. However, as we have seen it also presents the DBI with an inflexible and hard-to-extend framework for specifying optimizer rules and actions. This results in optimizers which consist of a mix of high-level abstractions (operators, algorithms, rules) together with implementation-level details (enforcers, property partitions, support functions for property transformations, implicit rules). Extensibility in such an environment is difficult to accomplish. The next chapter describes Prairie, our solution to the problem of providing a rule specification environment consisting of high-level abstractions, yet which can also be used to generate a fast optimizer.

Chapter 3

Prairie: A Language for Rule Specification

Prairie is an algebraic framework and language for specifying rewrite rules for rule-based query optimizers. We present the framework and language in this chapter. We begin by introducing basic concepts and notation and then present a means by which the mappings of search spaces and cost models are expressed as rewrite rules. Thus, the goal of this chapter is to lay a foundation for reasoning about query optimizers algebraically.

3.1 Overview

In the previous chapter, we described Volcano's rule specification language and listed some of its shortcomings. In particular, Volcano does not provide sufficiently high-level abstractions that would insulate a DBI from low-level details and provide a comfortable environment for extensibility. To remedy this, we propose a rule specification language called Prairie [21–23]. The Prairie optimizer paradigm is depicted schematically in Figure 3.1. As we will see in this chapter, rules in Prairie are specified at a conceptual rather than implementation level. This high-level specification is translated to generate a Volcano specification by a Prairie-to-Volcano (or P2V) preprocessor (as shown in Figure 3.1). The P2V preproces-

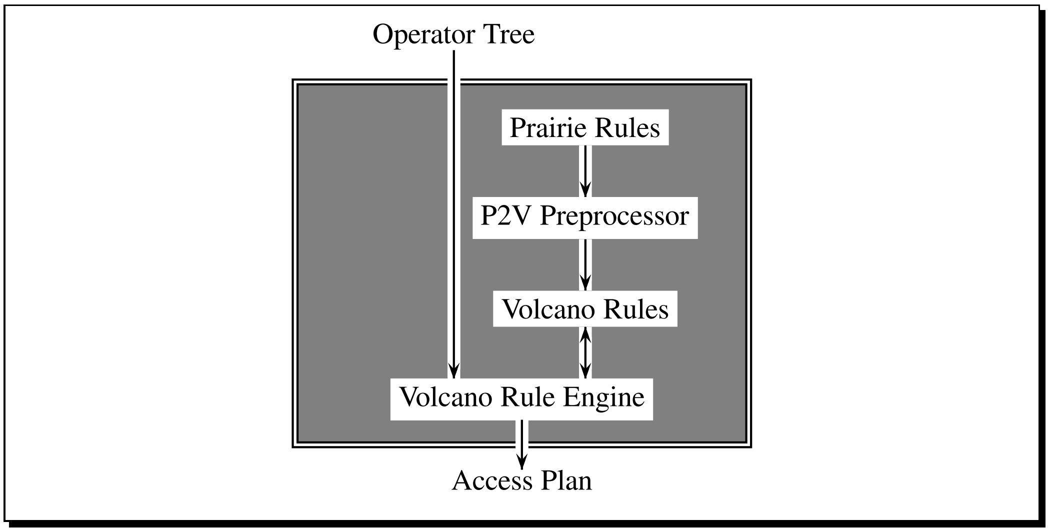


Figure 3.1: Schematic representation of the Prairie optimizer paradigm

sor is described in greater detail in Chapter 4. This chapter describes the rule specification language of Prairie.

3.2 Notation and Assumptions

This section lists the terminology used in Prairie. Some of the concepts are similar to those used in Volcano (see Section 2.2). However, there are some semantic differences.

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 of an operator. *Additional parameters* are "fine-grain" qualifications of an operator; their purpose is to describe an operator in more detail than essential parameters. As examples, some operators are given below; for each, we explicitly indicate their essential parameters and parenthetically note their additional parameters.

- SORT(S_1) sorts stream S_1. The sorting attribute is an additional parameter of SORT.
- RET(F) retrieves tuples of stored file F. Additional parameters to RET include the selection predicate, the projected attributes list, and the output tuple order.
- JOIN(S_1, S_2) joins streams S_1 and S_2. (S_1 denotes the *outer stream* and S_2 denotes the *inner stream*.) Additional parameters to JOIN include the join predicate and output stream tuple order.

Other operators are defined as they are needed.

Algorithms. *Algorithms* are concrete implementations of conceptual operators; they are 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. For example, File_scan and Index_scan are valid algorithms that implement the RET operator, and Merge_join

[1] Algorithms may have *tuning parameters* which are not parameters in 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 3.1: Operators and algorithms in a centralized Prairie query optimizer and their additional parameters

and Nested_loops are algorithms that implement the JOIN operator. Different algorithms offer different execution efficiencies.

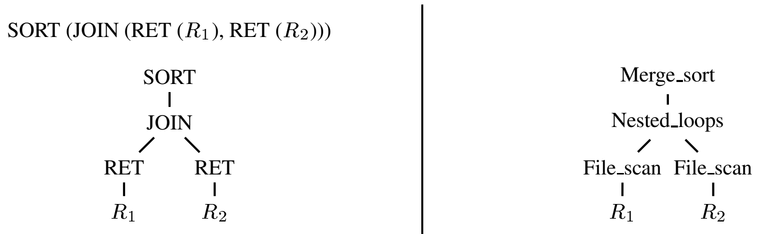
Table 3.1 lists some operators and algorithms implementing them together with their additional parameters.

Operator Tree. 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 operation. 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 5. A simple expression and its operator tree representation are shown in Figure 3.2(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. □

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

EXAMPLE 6. An access plan for the operator tree in Figure 3.2(a) is shown in Figure 3.2(b). Relations R_1 and R_2 are each retrieved using the File_scan algorithm, joined



(a) An expression and its corresponding operator tree

(b) A possible access plan for operator tree in (a)

Figure 3.2: Example of an operator tree and access plan

using Nested_loops, and finally sorted using Merge_sort. □

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 3.2 lists some typical properties that might be used in a descriptor. In general, descriptors for streams may be different from descriptors for stored files.[2]

The following notations will be useful in our subsequent discussions. If X_i is a stored file or stream, then $\mathbf{D_i}$ is its descriptor. Annotations of S_i are accessed by a structure member relationship, e.g., $\mathbf{D_i}$.cardinality. Also, let E be an expression and let $\mathbf{D}$ be its descriptor. This is written as $E : \mathbf{D}$.

EXAMPLE 7. The expression,

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

corresponds to the operator tree in Figure 3.2(a), and represents the join of two relations R_1 and R_2. The two relations are first RETrieved, then JOINed and finally SORTed. $\mathbf{D_1}$

[2] As an example, a stream may have a property join_predicate which is absent in a stored file's descriptor.

Property	Description
join_predicate	join predicate for JOIN operator
tuple_order	tuple order of resulting stream, DONT_CARE if none
cardinality	number of tuples of resulting stream
record_width	size of individual tuple in stream
attributes	list of attributes
cost	estimated cost of algorithm

Table 3.2: Properties of nodes in an operator tree in Prairie

and $\mathbf{D_2}$ are the descriptors of the stored files R_1 and R_2, respectively, $\mathbf{D_3}$ and $\mathbf{D_4}$ are the stream descriptors of the two RETs respectively, and $\mathbf{D_5}$ is the stream descriptor of the JOIN, and $\mathbf{D_6}$ is the stream descriptor of the SORT. Assuming that the descriptor fields for this expression are those given in Table 3.2, the selection predicate for the first RET is $\mathbf{D_3}$.selection_predicate, and that for the second RET is given by $\mathbf{D_4}$.selection_predicate. The join predicate of the JOIN node is given by $\mathbf{D_5}$.join_predicate, and the attributes that are output are given by $\mathbf{D_5}$.attributes. And so on. □

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

3.3 Prairie Optimization Paradigm

As shown in Figure 3.1, an optimizer is generated from a Prairie rule specification by converting them to a Volcano specification and then compiling it with the Volcano rule engine. Since the Volcano rule engine implements a top-down optimization search strategy (Section 2.1), this means that, currently, Prairie specifications can only generate top-down optimizers. Given an appropriate search engine, Prairie can potentially also be used with a bottom-up optimization strategy; however, we will not explore this topic in this dissertation.

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. For

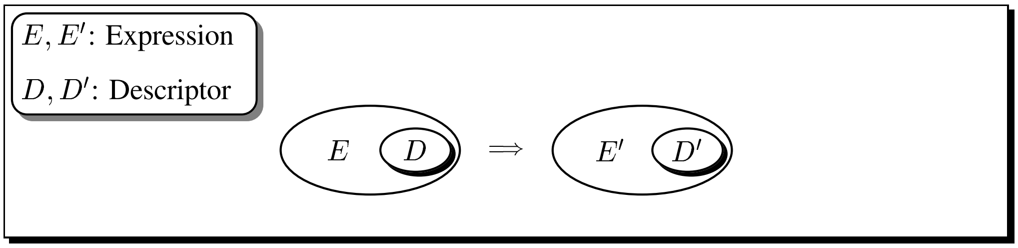


Figure 3.3: General form of a Prairie rule

this purpose, the DBI must define support functions (called init_descriptor) for each operator; these functions compute the descriptor properties for an operator given the descriptors of its input. For instance, init_descriptor_JOIN computes the descriptor properties (cardinality, record_width, attributes) from the descriptors of its two inputs. Our following discussions assume operator trees are initialized.

3.4 Rules in Prairie

Rules in Prairie correspond to rewrites between pairs of expressions, or between an expression and an access plan. Figure 3.3 shows the general format of a Prairie rule. This general rule results in 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, as Figure 3.3 shows. Thus, rules in Prairie represent units of *encapsulation* for operator tree and descriptor transformations.

T-rules and I-rules are defined precisely in Sections 3.5 and 3.6 and are illustrated with examples. The 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 3.1 and 3.2.

$$E(x_1, \ldots, x_n) : \mathbf{D_1} \Longrightarrow E'(x_1, \ldots, x_n) : \mathbf{D_2} \tag{3.1}$$

```
{{
        pre-test statements
}}
test
{{
        post-test statements
}}
```

Figure 3.4: General form of a Prairie T-rule

3.5 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 (3.1) (shown in Figure 3.4) 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[3] statements.

The left-hand sides of these statements refer to descriptors of expressions on the right-hand side of the T-rule (i.e., the descriptors whose values are to be computed); the right-hand sides of the statements can refer to any descriptor in the T-rule. As in procedural languages (like C), function calls can also appear on the right side of the assignment statements. These functions are called *helper* functions. Unlike support functions that are mandated by the language (e.g., init_descriptor), helper functions exist solely to simplify rule actions. 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

[3]The actions can be non-assignment statements (like helper function calls), but in this case, the P2V preprocessor (described in Chapter 4) needs some hints about the properties that are changed by the statement in order to correctly categorize each property. For simplicity, in this dissertation, we assume all actions consist of assignment statements.

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.[4] It is important to remember that the pre-test actions are carried out prior to the test; the post-test actions are performed only if a T-rule's test evaluates to TRUE, and all post-test actions are performed immediately, with no intermediate optimization of any descendant nodes of the root of E.

We now define the actions and tests of a T-rule more precisely. Let O_i be an abstract operator of E', and let $\mathbf{O_i}$ be its descriptor. Similarly, let I_i be an abstract operator of E and let $\mathbf{I_i}$ be its descriptor. (I_i is an operator that is input to the rule and O_i is an operator that is output by the rule.) Let M_j denote the jth descriptor property. Thus, $\mathbf{O_i}.M_j$ is the value of the jth property of descriptor $\mathbf{O_i}$. The left-hand side of an assignment refers to an output descriptor ($\mathbf{O_i}$) or a member of an output descriptor ($\mathbf{O_i}.M_j$). The right-hand side is an expression or a helper function call that only references input descriptors and/or their members. Here are a few examples:

$$
\begin{array}{rcll}
\mathbf{O_i} & = & \mathbf{I_k}\ ; & //\ \text{copy descriptor } \mathbf{I_k} \text{ to } \mathbf{O_i} \\
\mathbf{O_i}.M_j & = & \mathbf{I_k}.M_j + 4\ ; & //\ \text{expression defining } \mathbf{O_i}.M_j \\
\mathbf{O_3}.M_5 & = & \text{foo}\ (\mathbf{I_1}.M_5, \mathbf{I_2}.M_5)\ ; & //\ \text{helper function foo that computes } \mathbf{O_3}.M_5 \\
 & & & //\ \text{from inputs } \mathbf{I_1}.M_5 \text{ and } \mathbf{I_2}.M_5.
\end{array}
$$

The test for a T-rule's applicability is a boolean expression and normally involves checks on the values of output descriptors (e.g., $\mathbf{O_3}.M_5 > 6$); occasionally, helper functions may be needed.

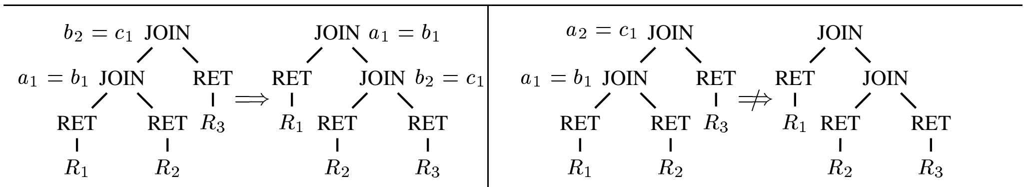
EXAMPLE 8. The associativity of JOINs is expressed by T-rule (3.2) in Figure 3.5(a). It rewrites a two-way join into an equivalent operator tree. The (single) pre-test statement computes the list of attributes of the new JOIN node on the right side. The test of the T-rule consists of a call to the helper function "is_associative", which returns TRUE or FALSE depending on whether the T-rule is applicable. If it is not, then the rule is rejected (e.g.,

[4] It may be possible to use data-flow analysis to partition the assignment statements automatically, but for now, we let the DBI do the partitioning.

$$\text{JOIN}(\text{JOIN}(S_1, S_2) : \mathbf{D_4}, S_3) : \mathbf{D_5} \Longrightarrow \text{JOIN}(S_1, \text{JOIN}(S_2, S_3) : \mathbf{D_6}) : \mathbf{D_7} \tag{3.2}$$

{{
    $\mathbf{D_6}$.attributes $=$ union ($\mathbf{D_2}$.attributes, $\mathbf{D_3}$.attributes) ;
}}
is_associative ($\mathbf{D_5}$.join_predicate, $\mathbf{D_2}$.attributes)
{{
    $\mathbf{D_7} = \mathbf{D_5}$;
    $\mathbf{D_7}$.join_predicate $=$ $\mathbf{D_4}$.join_predicate ;
    $\mathbf{D_6}$.join_predicate $=$ $\mathbf{D_5}$.join_predicate ;
    $\mathbf{D_6}$.record_width $=$ $\mathbf{D_2}$.record_width $+$ $\mathbf{D_3}$.record_width ;
    $\mathbf{D_6}$.cardinality $=$ compute_cardinality ($\mathbf{D_2}$, $\mathbf{D_3}$) ;
}}

(a) Join associativity T-rule



(b) Example of the associativity rule applied to an operator tree

(c) Example of an operator tree where the associativity rule does not apply because a cross-product would result

Figure 3.5: Join associativity T-rule

because it generates a cross-product), otherwise the post-test statements are executed. The post-test statements compute various other annotations of the new nodes that are generated by applying the T-rule. Note the use of helper functions "compute_cardinality" and "union" to compute descriptor properties.

Consider three relations R_1, R_2 and R_3, and let a_i, b_i and c_i be their respective sets of attributes. Figures 3.5(b) and 3.5(c) show, respectively, examples of the applicability and non-applicability of the join associativity T-rule. □

3.6 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

$E(x_1, \ldots, x_n) : \mathbf{D_1} \Longrightarrow A(x_1, \ldots, x_n) : \mathbf{D_2}$ (3.3)
test
{{
    pre-opt statements
}}
{{
    post-opt statements
}}

Figure 3.6: General form of a Prairie I-rule

E. The general form of an I-rule is given by Equation (3.3) (shown in Figure 3.6).

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 descendant of the root of E is 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 evaluated *after* all descendants x_i of the root of E are optimized. Normally, the post-opt statements compute properties that can only be determined once the inputs to the algorithm are completely optimized and their properties known.

EXAMPLE 9. I-rule (3.4) (shown in Figure 3.7) selects the Nested_loops algorithm to implement the JOIN operator. The test for this rule is TRUE since Nested_loops can be applied regardless of any property values. The pre-opt section consists of three assignment statements. The first statement sets the descriptor of Nested_loops to that of the JOIN. The next two statements express the fact that the tuple order of Nested_loops is the same as the tuple order of its left (outer) input; all other properties remain the same. The third statement in the pre-opt section ensures that this requirement is met by setting the tuple_order of S_1 on the right side.[5] The fourth statement computes the cost of using the Nested_loops algorithm.

[5]Actually, it is not enough to simply set the desired tuple order of S_1; it is also necessary to ensure that *after* optimization, S_1 does indeed have the required property. One way to satisfy this is to insert a SORT node in

```
JOIN(S_1, S_2) : D3 ==> Nested_loops(S_1 : D4, S_2) : D5                (3.4)
TRUE
{{
        D5 = D3 ;
        D4 = D1 ;
        D4.tuple_order = D3.tuple_order ;
        D5.cost = D4.cardinality * D2.cardinality ;
}}
{{
        D5.tuple_order = D4.tuple_order ;
}}
```

Figure 3.7: Nested loops I-rule

```
SORT(S_1) : D2 ==> Merge_sort(S_1) : D3                                 (3.5)
(D2.tuple_order != DONT_CARE)
{{
        D3 = D2 ;
        D3.cost = D3.cardinality * log(D3.cardinality) ;
}}
{{

}}
```

Figure 3.8: Merge sort I-rule

The post-opt section is executed after S_1 and S_2 are optimized; it consists of a single statement that assigns the tuple_order of the Nested_loops node. The Nested_loops algorithm returns its stream in the same order as its left input. □

EXAMPLE 10. Figure 3.8 shows the I-rule that implements the SORT operator using Merge_sort. I-rule (3.5) rewrites a stream such that it is sorted using the Merge_sort algorithm. The test for this I-rule is that the tuple order of the sorted stream must not be a

front of S_1 that can meet the sortedness requirement of S_1. Thus, in this case, we would need a T-rule (which introduces a new operator JOPR),

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

and an I-rule,

JOPR(S_1, S_2) : $\mathbf{D_3} \Longrightarrow$ Nested_loops$(S_1 : \mathbf{D_4}, S_2)$: $\mathbf{D_5}$.

In our discussions, this additional level of detail will be ignored for the sake of simplicity.

$$O(S_1) : \mathbf{D_2} \Longrightarrow \text{Null}(S_1 : \mathbf{D_3}) : \mathbf{D_4} \tag{3.6}$$

```
TRUE
{{
        D4 = D2 ;
        D3 = D1 ;
        D3.property = D2.property ;
}}
{{
        D4.cost = D3.cost ;
}}
```

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

$$\text{SORT}(S_1) : \mathbf{D_2} \Longrightarrow \text{Null}(S_1 : \mathbf{D_3}) : \mathbf{D_4} \tag{3.7}$$

```
TRUE
{{
        D4 = D2 ;
        D3 = D1 ;
        D3.tuple_order = D2.tuple_order ;
}}
{{
        D4.cost = D3.cost ;
}}
```

(b) Null sort I-rule

Figure 3.9: The "Null" algorithm concept

DONT_CARE order. The pre-opt section consists of two statements. The first copies the descriptor from the left side to the expression on the right, and the second computes the cost of using Merge_sort. The post-opt section is empty. □

3.6.1 The 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 (3.6) in Figure 3.9(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 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 11. Equation (3.7) (in Figure 3.9(b)) shows the I-rule that rewrites the SORT operator to use a Null algorithm. The third pre-opt statement sets the tuple order of S_1 on the right side to be the tuple order of the SORT node, thus ensuring that when S_1 is optimized on the right side, it will have the same tuple order as the SORT node. □

3.7 Advantages of Prairie

In the previous chapter, we described Volcano's rule specification language and some of its problems. Prairie addresses these problems:

- Prairie does not contain any implicit operators, algorithms, or rules (as in Volcano; see Section 2.8.1). This implies that extending a Prairie specification is simple, and property transformations are also simplified since an abstraction (e.g., algorithm) directly corresponds to a concept, instead of an implementation-level detail (e.g., enforcer in Volcano).

- Since properties are represented using single descriptors, it is trivial to add new properties, or change existing ones.

- Property transformations are vastly simplified, since, instead of a multitude of support functions (as required by Volcano, Table 2.3), property transformations in Prairie are accomplished on a per-rule basis. This is seen clearly from the forms of the general rules for Volcano (Figure 2.8) and Prairie (Figure 3.3). The only support function that Prairie requires is init_descriptor for each operator. Adding algorithms in Prairie don't require the DBI to define additional support functions as in Volcano.

3.8 Summary

In this chapter, we presented the rule specification language of Prairie. Prairie strives for high-level abstractions that insulate a DBI from implementation-level details of the optimizer. (Not all details of the underlying implementation can be hidden from a DBI. For example, the top-down strategy is evident in the general form of a Prairie I-rule (see Figure 3.6). However, other details such as enforcers are hidden.) The resulting environment is streamlined, robust, and easily extensible (as described in Section 3.7).

High-level abstractions like those that Prairie provides can lead to inefficient implementations of optimizers if one is not careful in compiling Prairie rules. In the next chapter, we describe the P2V preprocessor that automatically generates a Volcano specification from a Prairie specification. In this way, the advantages of Prairie are preserved while making use of Volcano's ability to generate efficient optimizers. Since Prairie provides only a rule specification environment, this also means that we can utilize Volcano's rule engine for the search strategy of a Prairie optimizer.

Chapter 4

The Prairie-to-Volcano Preprocessor

In the previous chapter, we described the rule specification language of Prairie and showed how it overcame some of Volcano's drawbacks concerning high-level abstractions and extensibility. We also mentioned that a Prairie specification could be translated into a Volcano specification to make use of Volcano's advantages, and to make it compatible with Volcano's rule engine. This chapter describes the Prairie-to-Volcano (or P2V) preprocessor that accomplishes this translation.

4.1 Overview

There are four main responsibilities of the P2V preprocessor:

- Establishing the correspondence between the various concepts of Prairie to similar ones in Volcano. Specifically, this means that the P2V preprocessor must translate stored files, 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 corresponding Volcano 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. As we will see in this chapter, translating an I-rule's actions is complicated by the fact that Volcano accomplishes most of its implementation rule actions via DBI-defined support functions.

- Generating a compact Volcano rule set from a Prairie specification. This means that the P2V preprocessor attempts to remove unused rules, and to consolidate 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.

The following sections describe each of these steps in greater detail. The P2V preprocessor is about 6000 lines of `flex` and `bison` code, and took about 3 man-months to implement.

4.2 Correspondence of Elements in Prairie and Volcano

4.2.1 Operators, Algorithms, and Enforcers

Operators and algorithms in Prairie correspond directly to operators and algorithms, respectively, in Volcano. That is, an operator in Prairie represents the same operator in Volcano. Similarly for algorithms. The difficulty arises when some algorithms in Prairie are identified by the P2V preprocessor as implicit (as defined in Section 2.8.1). These algorithms then correspond to enforcers in Volcano. This process is described in greater detail below.

In Chapter 2, we mentioned that enforcers in Volcano are really algorithms, except that they do not implement any particular operator. This means that enforcers can be inserted anywhere in an expression, depending on needed physical properties.

Now consider operators in Prairie. Some operators have the "Null" algorithm as one

$$
\begin{array}{lcl}
O(S_1) : \mathbf{D_2} & \Longrightarrow & A_1(S_1) : \mathbf{D_3} \\
 & \cdots & \\
O(S_1) : \mathbf{D_2} & \Longrightarrow & A_n(S_1) : \mathbf{D_3} \\
O(S_1) : \mathbf{D_2} & \Longrightarrow & \text{Null}(S_1 : \mathbf{D_3}) : \mathbf{D_4}
\end{array}
$$

Figure 4.1: Identifying implicit operators, algorithms, and rules

of their implementations (see Section 3.6.1). Whenever Null is used as an implementation, it has the same effect as if the algorithm were not present in the operator tree at all. On the other hand, when a non-Null algorithm is used as an implementation, an "actual" algorithm is introduced in the operator tree. Also, whenever a Null algorithm is used in Prairie, it serves to push its needed properties down to its inputs. In other words, it tries to *enforce* certain properties on its inputs. Thus, Null algorithms in Prairie are indications of enforcers in Volcano. Stated differently, a choice of the Null algorithm in Prairie implies that no enforcer is introduced (in an operator tree) in Volcano, whereas a choice of a non-Null algorithm introduces an enforcer.

More formally, consider the sequence of Prairie I-rules shown in Figure 4.1. Each I-rule specifies an implementation of the abstract operator O, in terms of algorithms A_1 through A_n, and the Null algorithm. The Null algorithm acts as a conduit to push all properties of the operator O down to its input S_1. The P2V preprocessor marks operator O as an implicit operator, algorithms A_1 through A_n as implicit algorithms, and all the rules in Figure 4.1 as implicit rules. For brevity, in this chapter, we will call the operator O in Prairie an *enforcer-operator*, and algorithms A_1 through A_n, *enforcer-algorithms*. Enforcer-operators are discarded when translating to Volcano, enforcer-algorithms in Prairie correspond to enforcers in Volcano, and all implicit rules are discarded when translating to Volcano. (We will discuss later how to translate the actions associated with implicit I-rules.) The Null algorithm does not appear in a Volcano specification at all. Expressions in Prairie which contain enforcer-operators must be modified so that they are legal expressions in Volcano; we will show how this is done later.

EXAMPLE 12. Consider I-rules 3.5 and 3.7 from Figures 3.8 and 3.9(b) (repeated below, without the associated tests or actions):

SORT(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ Merge_sort(S_1) : $\mathbf{D_3}$

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

These two I-rules together describe Merge_sort to be an enforcer-algorithm in Prairie (an enforcer in Volcano), and SORT as an enforcer-operator. □

4.2.2 Operator Trees and Access Plans

Operator trees in Prairie are expressed in a functional notation, and those in Volcano are expressed in a LISP prefix notation. Apart from this syntactic difference, operator trees and access plans in the two specifications bear a one-to-one correspondence.

EXAMPLE 13. The expression,

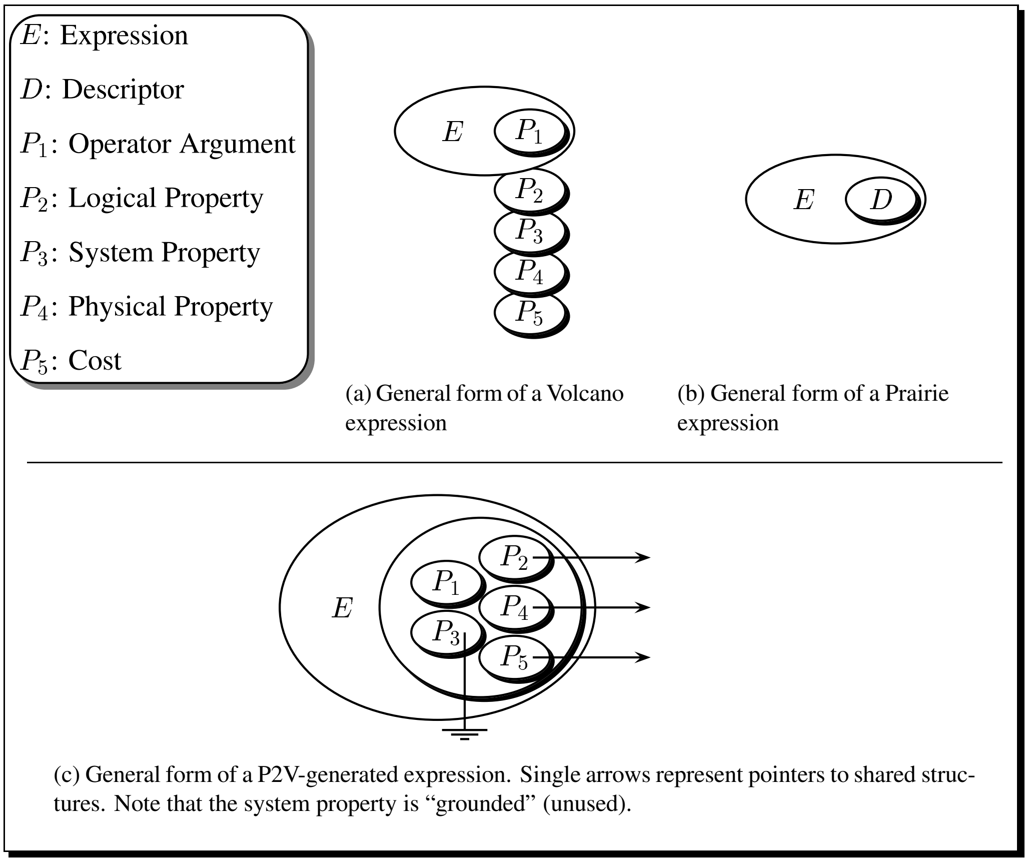
$$\text{JOIN}(\text{RET}(R_1) : \mathbf{D_3}, \text{RET}(R_2) : \mathbf{D_4}) : \mathbf{D_5}$$

in Prairie, which denotes the RETrieval of relations R_1 and R_2, and their subsequent JOIN, is represented in Volcano as

(JOIN ?op_arg5 ((RET ?op_arg3 ()) (RET ?op_arg4 ()))

The descriptors $\mathbf{D_3}$, $\mathbf{D_4}$, and $\mathbf{D_5}$ map to the operator arguments ?op_arg3, ?op_arg4, and ?op_arg5, respectively. In addition, since Volcano expressions represent only stream inputs explicitly, the relation names R_1 and R_2 are also incorporated in the operator arguments ?op_arg3 and ?op_arg4, respectively. A more precise correspondence between Prairie descriptors and Volcano properties is described in the next section. □

Access plans in Prairie correspond to physical expressions in Volcano; they have the same correspondence as do operator trees in Prairie and Volcano; plans are transformed from a functional notation to a LISP prefix notation with descriptors transformed into algorithm arguments.



(a) General form of a Volcano expression

(b) General form of a Prairie expression

(c) General form of a P2V-generated expression. Single arrows represent pointers to shared structures. Note that the system property is "grounded" (unused).

Figure 4.2: General expressions in Volcano, Prairie, and P2V-generated specifications

4.2.3 Descriptors and Properties

In Prairie, there is one structure that represents all the properties of a node in an operator tree, the descriptor. Volcano, on the other hand, represents properties of expressions in five different structures. Figures 4.2(a) and 4.2(b) show the general form of expressions in Volcano and Prairie (i.e., the forms visible to the DBI).

The advantage of having separate property structures in Volcano is that they correspond to the way the properties are used and stored internally by the Volcano rule engine. In other words, the partition of properties has *semantic* meaning for the internal representation. For example, logical properties in Volcano are computed before optimization begins.

Furthermore, the logical properties of operator trees in the same equivalence class (that is, logically equivalent expressions) are the same; thus, it makes sense to share a common logical property structure between all such equivalent operator trees, and indeed, this is what the Volcano rule engine does. This is easily done if logical properties are defined as a set of properties distinct from all other properties.

On the other hand, partition of properties has its drawbacks in the area of high-level abstractions. As listed in Section 2.8.2, these drawbacks complicate the task of specifying an optimizer by forcing the DBI to think of low-level (or internal) details of property representation. Also, as mentioned earlier, these property partitions can be brittle, and change when operators, algorithms, or properties are changed. As mentioned in Chapter 3, Prairie overcomes these drawbacks by representing properties in a single structure, a descriptor.

What is needed, then, is a translation from the high-level abstraction of Prairie descriptors to the lower-level representation of Volcano's property structures, where the advantage of property sharing is preserved. The P2V preprocessor accomplishes this by retaining a single property structure with every expression, but partitioning the descriptor *internally* into separate structures that correspond to Volcano's property structures. This is shown pictorially in Figure 4.2(c). Note how logical and physical properties are translated to pointers to property structures; this allows sharing between expressions by setting pointers to common structures. The cost property structure is not shared, but it is converted to a pointer since it makes sense only for an algorithm (and thus can be set to null for an operator). Also, as we mentioned in Section 2.2, the system property set behaves identically to the logical property set, so the P2V preprocessor "grounds" the system property set (and doesn't use it).

The problem now is deciding what property set (operator argument, logical property, cost, or physical property) each property in a descriptor belongs to. This is determined as follows.

Initially, all properties in a descriptor are logical properties, by default. Consider a T-rule that rewrites an expression E to E'. If property P of the root of E' is changed[1] in either

[1] The problem of detecting whether a property is changed is a difficult one since properties can be changed via assignment statements, helper function calls, etc. in C. To avoid implementing a full-blown C (or C++) parser,

$$
\begin{array}{ll}
\text{JOIN(JOIN}(S_1, S_2) : \mathbf{D_4}, S_3) : \mathbf{D_5} \Longrightarrow \text{JOIN}(S_1, \text{JOIN}(S_2, S_3) : \mathbf{D_6}) : \mathbf{D_7} & (4.1) \\
\{\{ & \\
\quad \mathbf{D_6}.\text{attributes} = \text{union}\ (\mathbf{D_2}.\text{attributes}, \mathbf{D_3}.\text{attributes})\ ; & \\
\}\} & \\
\text{is\_associative}\ (\mathbf{D_5}.\text{join\_predicate}, \mathbf{D_2}.\text{attributes}) & \\
\{\{ & \\
\quad \mathbf{D_7} = \mathbf{D_5}\ ; & \\
\quad \boxed{\mathbf{D_7}.\text{join\_predicate} = \mathbf{D_4}.\text{join\_predicate}\ ;} & \\
\quad \mathbf{D_6}.\text{join\_predicate} = \mathbf{D_5}.\text{join\_predicate}\ ; & \\
\quad \mathbf{D_6}.\text{record\_width} = \mathbf{D_2}.\text{record\_width} + \mathbf{D_3}.\text{record\_width}\ ; & \\
\quad \mathbf{D_6}.\text{cardinality} = \text{compute\_cardinality}\ (\mathbf{D_2}, \mathbf{D_3})\ ; & \\
\}\} &
\end{array}
$$

Figure 4.3: Identifying operator arguments

the pre-test or post-test sections of the T-rule, then P is identified as an operator argument. The reason is that T-rules only access operator arguments or logical properties, and, since the logical properties of E' are the same as those of E (and thus cannot be changed in a T-rule), P must be an operator argument.

To illustrate this, consider the join associativity T-rule from Chapter 3, repeated in Figure 4.3. Note the boxed statement: it changes the join predicate of $\mathbf{D_7}$, the descriptor of the root JOIN on the right side of the T-rule. Thus, join predicate is an operator argument.

Cost properties are easy to detect: like Volcano, Prairie requires a cost property to have the type COST. Thus, all properties in the descriptor with the type COST are cost properties (although, typically, there is only one cost property).

Physical properties are also easy to identify. The P2V preprocessor looks at all "Null I-rules" (i.e., I-rules where Null implements an operator). If any property of the input stream of the Null algorithm is changed in the pre-opt section of the I-rule, then that property is a physical property. The rationale for this is simple: properties that are copied from an operator to its input must be "needed" properties (in Volcano's terminology), and thus, are physical properties. As an example, consider the Null sort I-rule from Chapter 3, repeated in

P2V imposes a requirement on Prairie rules that all changes to descriptor properties be identified by an annotation to the statement. Assignments to descriptor properties are accomplished by a function called dcopy. Thus, the boxed statement in Figure 4.3 is actually written as dcopy($\mathbf{D_7}$.join_predicate, $\mathbf{D_4}$.join_predicate). It is important to note that this requirement does not sacrifice the full power of C.

$$
\begin{array}{l}
\text{SORT}(S_1) : \mathbf{D_2} \Longrightarrow \text{Null}(S_1 : \mathbf{D_3}) : \mathbf{D_4} \qquad (4.2) \\
\text{TRUE} \\
\{\{ \\
\qquad \mathbf{D_4} = \mathbf{D_2} \; ; \\
\qquad \mathbf{D_3} = \mathbf{D_1} \; ; \\
\qquad \boxed{\mathbf{D_3}.\text{tuple\_order} = \mathbf{D_2}.\text{tuple\_order} \; ;} \\
\}\} \\
\{\{ \\
\qquad \mathbf{D_4}.\text{cost} = \mathbf{D_3}.\text{cost} \; ; \\
\}\}
\end{array}
$$

Figure 4.4: Identifying physical properties

Figure 4.4. The boxed statement changes the tuple_order property of S_1, the input of Null. Thus, the preprocessor marks tuple_order as a physical property.

4.3 Translating T-rules

A T-rule in Prairie corresponds to a transformation rule in Volcano. The translation from a T-rule to a transformation rule is straightforward. Figure 4.5 shows the correspondence between a Prairie T-rule and a Volcano trans_rule. Consider the general form of a Prairie T-rule shown in Figure 4.5(a). X and Y are sets of C (or C++) statements, perhaps computing properties of the various expressions in the rule. T is a boolean expression, possibly containing property references. A Volcano transformation rule is derived from this T-rule as shown in Figure 4.5(b). X', Y' and T' are derived from X, Y, and T, respectively, by appropriately translating the property references in each. This is necessary, since, as described in Section 4.2.3, a Prairie descriptor is transformed into a structure containing multiple property sets. This is best illustrated with an example.

Consider the join associativity T-rule from Chapter 3, shown in Figure 4.6(a). The Volcano transformation rule generated by the P2V preprocessor is shown in Figure 4.6(b). The statements are numbered, and correspond to the same numbered statements in Figure 4.6(a). As shown in Figure 4.2, operator arguments are a separate structure in the translated descriptor. Hence the reference (in statement 4 in the application code of the trans_rule)

$$E(x_1, \ldots, x_n) : \mathbf{D_1} \Longrightarrow E'(x_1, \ldots, x_n) : \mathbf{D_2} \tag{4.3}$$

```
{{
        X
}}
T
{{
        Y
}}
```

(a) General Prairie T-rule

$$(E \text{ ?op\_arg1 } (?1 \ldots ?n)) \rightarrow (E' \text{ ?op\_arg2 } (?1 \ldots ?n)) \tag{4.4}$$

```
%cond_code
{{
        X'
        if (!(T')) REJECT;
}}
%appl_code
{{
        Y'
}}
```

(b) Volcano trans_rule corresponding to the T-rule in (a). $?i$ corresponds to x_i, and X', T', and Y' correspond to X, T, and Y, respectively, with property references suitably modified.

Figure 4.5: Translating T-rules

to op_arg as a C struct. On the other hand, since logical properties are pointers to shared structures, the reference in statement 2 to log_prop is as a pointer.

There are two interesting points to note in this translation. Note that a single descriptor copy ($\mathbf{D_7} = \mathbf{D_5}$) statement in Figure 4.6(a) results in two assignment statements in Figure 4.6(b), one for copying the operator argument (join predicate), and another for copying a *pointer* to a shared logical property structure. Since physical properties and cost are not relevant in a T-rule, they are not copied.[2]

Second, note that statements 1, 6, and 7 in Figure 4.6(a), which compute logical properties of $\mathbf{D_6}$, are reduced to NO_OP statements.[3] Again, this is because logical properties are computed before optimization begins (by DBI-defined support functions, init_descriptor), so, these statements are redundant.

[2] They are, however, initialized by DBI-defined init_descriptor functions.

[3] The P2V preprocessor macro-defines NO_OP as an empty statement.

JOIN(JOIN(S_1, S_2) : $\mathbf{D_4}, S_3$) : $\mathbf{D_5} \Longrightarrow$ JOIN(S_1, JOIN(S_2, S_3) : $\mathbf{D_6}$) : $\mathbf{D_7}$ (4.5)

{{
  1. $\mathbf{D_6}$.attributes = union ($\mathbf{D_2}$.attributes, $\mathbf{D_3}$.attributes) ;
}}
2. is_associative ($\mathbf{D_5}$.join_predicate, $\mathbf{D_2}$.attributes)
{{
  3. $\mathbf{D_7} = \mathbf{D_5}$;
  4. $\mathbf{D_7}$.join_predicate = $\mathbf{D_4}$.join_predicate ;
  5. $\mathbf{D_6}$.join_predicate = $\mathbf{D_5}$.join_predicate ;
  6. $\mathbf{D_6}$.record_width = $\mathbf{D_2}$.record_width + $\mathbf{D_3}$.record_width ;
  7. $\mathbf{D_6}$.cardinality = compute_cardinality ($\mathbf{D_2}, \mathbf{D_3}$) ;
}}

(a) Join associativity T-rule

(JOIN ?op_arg5 ((JOIN ?op_arg4 (?1 ?2)) ?3)) $\rightarrow$ (JOIN ?op_arg7 (?1 (JOIN ?op_arg6 (?2 ?3)))) (4.6)

```
%cond_code
{{
    1. NO_OP ;
    2. if (!(is_associative (?op_arg5 → op_arg.join_predicate, ?2 → expr.argument.log_prop → attributes)))
         REJECT ;
}}
%appl_code
{{
    3. ?op_arg7 → op_arg.join_predicate = ?op_arg5 → op_arg.join_predicate ;
    3. ?op_arg7 → log_prop = ?op_arg5 → log_prop ;
    4. ?op_arg7 → op_arg.join_predicate = ?op_arg4 → op_arg.join_predicate ;
    5. ?op_arg6 → op_arg.join_predicate = ?op_arg5 → op_arg.join_predicate ;
    6. NO_OP ;
    7. NO_OP ;
}}
```

(b) P2V-generated trans_rule corresponding to the T-rule in (a). Statement numbers correspond to the same numbered statements in (a).

Figure 4.6: Translating the join associativity T-rule

A final issue in translating T-rules is that of enforcer-operators. Recall that enforcer-operators are discarded by the preprocessor when translating Prairie rules into Volcano. This means that if an expression in a T-rule contains an enforcer-operator, the P2V preprocessor deletes the enforcer-operator when translating the expression to Volcano. What happens to the T-rule's actions when a statement references the descriptor of the (now-discarded) enforcer-operator? The answer becomes obvious when we consider that enforcer-operators have the same logical properties and operator arguments as their input. Thus, when an enforcer-operator like SORT(S_1) : $\mathbf{D_2}$ is deleted, references to properties of $\mathbf{D_2}$ are con-

JOIN(S_1, S_2) : $\mathbf{D_3} \Longrightarrow$ JOPR(SORT(S_1) : $\mathbf{D_4}$, SORT(S_2) : $\mathbf{D_5}$) : $\mathbf{D_6}$ (4.7)

```
{{
        1. D4 = D1 ;
        2. D5 = D2 ;
}}
3. (D4.cardinality < 10000)
{{
        ...
}}
```

(a) T-rule with enforcer-operator SORT

```
(JOIN ?op_arg3 (?1 ?2)) → (JOPR ?op_arg6 (?1 ?2))                    (4.8)
%cond_code
{{
        1. NO_OP ;
        2. NO_OP ;
        3. if (!((?1 → expr.argument.log_prop → cardinality < 10000))) REJECT ;
}}
%appl_code
{{
        ...
}}
```

(b) P2V-generated trans_rule corresponding to the T-rule in (a). Statement numbers correspond to the same numbered statements in (a).

Figure 4.7: Translating T-rules with enforcer-operators

verted to references to properties of $\mathbf{D_1}$ (the descriptor of S_1). This is shown in Figure 4.7 where the reference to the cardinality of a SORT (in the test of the T-rule) is translated to a reference to the cardinality of the input of the SORT in the Volcano trans_rule.

4.4 Translating I-rules

I-rules in Prairie correspond to Volcano's implementation rules. However, the translation of I-rules is more complicated than the translation of T-rules. This is primarily because the actions associated with a Prairie I-rule are all expressed concomitantly (pre-opt and post-opt) with the rule. In Volcano, on the other hand, there are *four* distinct support functions (as listed in Table 2.3) that are responsible for most of the property transformations associated with a implementation rule. This means that when translating a Prairie I-rule to Volcano's imple-

$E(x_1, \ldots, x_n) : \mathbf{D_1} \Longrightarrow A(x_1, \ldots, x_n) : \mathbf{D_2}$ (4.9)

```
T
{{
        X
}}
{{
        Y
}}
```

(a) General Prairie I-rule

$(E$?op_arg1 $(?1 \ldots ?n)) \rightarrow (A$?al_arg2 $(?1 \ldots ?n))$ (4.10)

```
%cond_code
{{
        if (!(T')) REJECT;
}}
%appl_code
{{
        // book-keeping statements
}}
```

Support function do_any_good_A

```
{
        X'
}
```

Support function derive_A_phy_prop

```
{
        Y'
}
```

(b) Volcano impl_rule corresponding to the I-rule in (a). $?i$ corresponds to x_i, and X', T', and Y' correspond to X, T, and Y, respectively, with property references suitably modified. Note that the general I-rule is translated to an implementation rule and two support functions corresponding to the algorithm A. The remaining support functions (get_input_pv and cost) are "grounded" (i.e., contain only book-keeping statements).

Figure 4.8: Translating I-rules

mentation rule, the actions of the I-rule have to be partitioned into the appropriate support functions. How this is accomplished is the most challenging problem.

Broadly speaking, the P2V preprocessor partitions the actions of an I-rule as follows. The test of an I-rule is translated to the condition code of a Volcano implementation rule. The pre-opt section of an I-rule is transformed into the do_any_good function for the appropriate algorithm, and the post-opt section is translated to the derive_phy_prop function. The two other support functions, get_input_pv and cost, are translated to dummy functions. That is, the statements in these two functions are meant for book-keeping purposes, and are

$\text{JOIN}(S_1, S_2) : \mathbf{D_3} \Longrightarrow \text{Nested\_loops}(S_1 : \mathbf{D_4}, S_2) : \mathbf{D_5}$ (4.11)

```
1. TRUE
{{
        2. D5 = D3 ;
        3. D4 = D1 ;
        4. D4.tuple_order = D3.tuple_order ;
        5. D5.cost = D4.cardinality * D2.cardinality ;
}}
{{
        6. D5.tuple_order = D4.tuple_order ;
}}
```

(a) Nested loops I-rule

(JOIN ?op_arg1 (?1 ?2)) → (Nested_loops ?al_arg2 (?1 ?2)) (4.12)

```
%cond_code
{{
        1. if (!(TRUE)) REJECT ;
}}
%appl_code
{{
        // book-keeping statements
}}
Support function do_any_good_Nested_loops
{
        // book-keeping statements
        2. ret_value → algorithm_argument.log_prop = operator_argument → log_prop ;
        3. NO_OP ;
        4. ret_value → algorithm_argument.al_arg → required[0] → tuple_order = needed_phy_prop → tuple_order ;
        5. ret_value → algorithm_argument.al_arg → cost =
              ret_value → algorithm_argument.al_arg → desc[0] → log_prop → cardinality
              * ret_value → algorithm_argument.al_arg → desc[1] → log_prop → cardinality ;
}
Support function derive_Nested_loops_phy_prop
{
        // book-keeping statements
        6. algorithm_argument → phy_prop → tuple_order = input_phy_prop[0] → tuple_order ;
}
```

(b) P2V-generated impl_rule and support functions corresponding to the I-rule in (a). Statement numbers correspond to the same numbered statements in (a).

Figure 4.9: Translating the nested loops I-rule

not taken from the actions of any I-rule. These book-keeping statements are primarily responsible for pointer manipulations and copying of shared property structures. This process is shown in Figure 4.8 which shows a general Prairie I-rule (Figure 4.8(a)) and the Volcano implementation rule and support functions that it is translated to (Figure 4.8(b)).

Most of the difficulty in an I-rule translation involves translating property references in Prairie to those in Volcano. Since the property translation process involves pointers (as shown in Figure 4.2), the generated code for the Volcano support functions contains a lot of pointer manipulation operations. It is illuminating to consider an example that illustrates the translation process.

Consider the nested loops I-rule from Chapter 3, shown in Figure 4.9(a). The Volcano implementation rule and support functions (do_any_good_Nested_loops and derive_Nested_loops_phy_prop) generated by the P2V preprocessor are shown in Figure 4.9(b). The statements are numbered, and correspond to the same numbered statements in Figure 4.9(a).

The interesting points to note are the translation of the pre-opt and the post-opt sections of the I-rule. As shown in Figure 4.8, these are translated to support functions do_any_good_Nested_loops and derive_Nested_loops_phy_prop, respectively. Notice how the third statement in the I-rule is translated to a NO_OP statement. This is because the only properties of D_4 that can be changed in the pre-opt section are the physical properties (this corresponds to Volcano's concept of "needed physical properties"; see Section 2.2), which is accomplished by the fourth statement. The physical properties of the inputs of Nested_loops are represented as pointers to physical property structures, as evidenced by the translation of the fourth statement.

In general, the translation of I-rule actions involves a lot of book-keeping statements and, as Figure 4.9(b) shows, a lot of pointer traversals. This is necessary because of the way the P2V preprocessor transforms Prairie descriptors into Volcano properties, and also because of the rigid structure of Volcano support functions (not all of them have access to all property structures because the parameter list of support functions is pre-defined).

4.4.1 Translating Enforcers

Figure 4.1 shows how the P2V preprocessor identifies enforcers (implicit algorithms) in Prairie. As described in Section 4.2.1, rules containing enforcers are discarded by the preprocessor. What becomes of the actions in such rules?

SORT(S_1) : $\mathbf{D_2} \Longrightarrow$ Merge_sort(S_1) : $\mathbf{D_3}$ (4.13)
1. ($\mathbf{D_2}$.tuple_order != DONT_CARE)
{{
    2. $\mathbf{D_3} = \mathbf{D_2}$;
    3. $\mathbf{D_3}$.cost = $\mathbf{D_3}$.cardinality $*$ log($\mathbf{D_3}$.cardinality) ;
}}
{{

}}

(a) Merge sort I-rule

Support function do_any_good_Merge_sort
{
    // book-keeping statements
    1. if (needed_phy_prop → tuple_order != DONT_CARE) {
    2.   ret_value → algorithm_argument.log_prop = operator_argument → log_prop ;
    3.   ret_value → algorithm_argument.al_arg → cost =
               ret_value → algorithm_argument.log_prop → cardinality
               $*$ log(ret_value → algorithm_argument.log_prop → cardinality) ;
    }
}
Support function derive_Merge_sort_phy_prop
{
    // book-keeping statements
}

(b) P2V-generated support functions corresponding to the I-rule in (a). Statement numbers correspond to the same numbered statements in (a).

Figure 4.10: Translating I-rules with enforcer-algorithms

The answer is obvious when one considers that implicit rules are basically I-rules. Thus, enforcers have support functions (just like explicit algorithms). This means that actions of an implicit rule can be translated as statements in the appropriate support functions. The general mechanism for translating I-rules (Figure 4.8) holds for enforcers with one difference: the test T of an implicit rule is translated to T', and is used as a condition to bracket the set of actions Y' (as depicted in Figure 4.8(b)) in the do_any_good support function. The rest of the translation works exactly as for other I-rules.

Figure 4.10 illustrates the translation of the merge sort I-rule into two support functions (and no impl_rules).

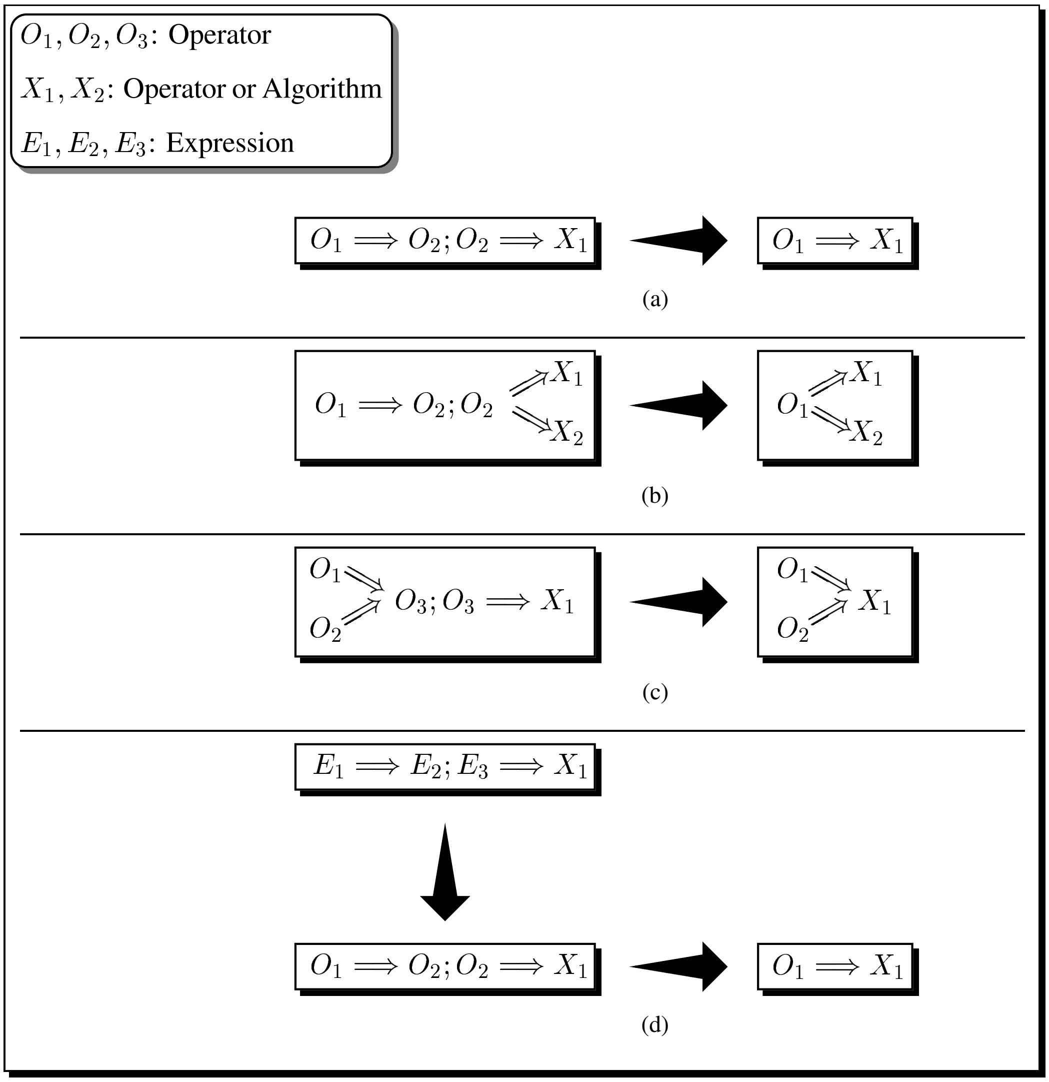


Figure 4.11: Rule compaction

4.5 Rule Compaction

Rule compaction is a technique by which the P2V preprocessor generates a smaller set of rules from a typically large rule set. This has a direct bearing on the efficiency of the generated optimizer, since a smaller rule set results in a more efficient optimizer (see Appendix B for a more formal argument for rule compaction).

The basic idea in rule compaction by the P2V preprocessor is that transitive rewrite rules can be replaced with a single rule. In other words, if a rule transforms one expression into another which is, in turn, transformed into a third expression by a second rule, then the two rules can be replaced with a single rule that directly transforms the original expression into the final one. This general principle of rule compaction results in four specific cases, as shown in Figure 4.11. The compaction algorithm is not as simple as it seems at first glance, since rules have actions which have to meet certain criteria to be compacted.

Rule compaction works by progressively reducing pairs of rules to single rules. In general, the first rule in the pair consists of a T-rule transforming an operator into another. The second rule in the pair can either be a T-rule or an I-rule. The actions of a compacted rule set are derived from the actions of the last of the sequence of the rules being compacted.

Consider Figure 4.11(b) as an example. The Prairie rule set being compacted can be viewed as consisting of two pairs of rules,

$$O_1 \Longrightarrow O_2; O_2 \Longrightarrow X_1$$
$$O_1 \Longrightarrow O_2; O_2 \Longrightarrow X_2,$$

where the second rule in each pair is either a T-rule or an I-rule (depending on whether X_1 and X_2 are operators or algorithms). It is clear from the above that each pair of rules represents a transitive mapping: the first, from O_1 to X_1, and the second, from O_1 to X_2. The P2V preprocessor, thus, reduces the three Prairie rules shown above, to two Volcano rules, as Figure 4.11(b) shows. Figures 4.11(a) and 4.11(c)[4] depict similar compactions.

An interesting case of rule compaction is shown in Figure 4.11(d). The Prairie rule set being compacted consists of a T-rule transforming E_1 to E_2, and another rule that transforms E_3 to X_1. Expressions E_1, E_2, E_3 may consist of enforcer-operators. Now, as described in Section 4.2.1, enforcer-operators are discarded when expressions and rules are translated from Prairie to Volcano by the P2V preprocessor. It is, thus, possible, that when enforcer-operators are deleted from E_1, E_2, and E_3, that expression E_1 is transformed to an

[4]This compaction is currently not implemented in the P2V preprocessor because, in our experience, it usually does not occur. However, there is no technical difficulty in implementing it.

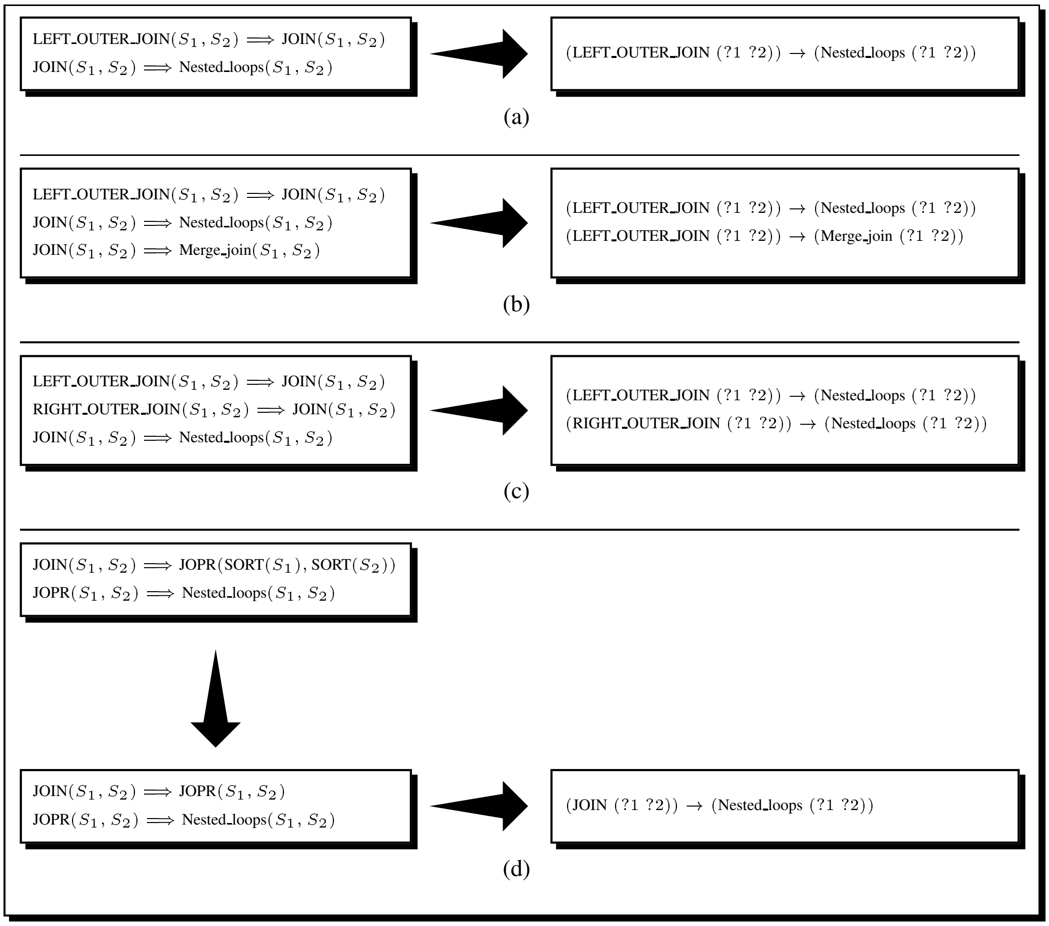


Figure 4.12: Examples of rule compaction (cf. Figure 4.11). The left sides show Prairie rule sets, the right sides are Volcano rule sets. For clarity, descriptors in the Prairie rules, and operator and algorithm arguments in the Volcano rules, are omitted, as are all rule actions.

operator O_1, and expressions E_2 and E_3 are transformed into operator O_2. This means that E_2 and E_3 *unify*[5] to the operator O_2. As shown in Figure 4.11(d), this compaction reduces to the one in Figure 4.11(a).

As mentioned earlier, there are certain restrictions on the form and actions of the first T-rule in each pair that is compacted. First, the arity (i.e., the number of inputs) of the opera-

[5]This process is called unification because it is similar to the technique used in automatic theorem proving by resolution in first-order logic (see, e.g., [18]).

tors on each side of the rule must be the same. Second, the test for the T-rule must be TRUE (i.e., the rule must be unconditionally applicable). Third, the actions (pre-test and post-test) in the T-rule must consist either of (1) descriptor assignment statements that copy the descriptors of the corresponding expressions on either side of the rule, or (2) assignment statements that are translated to NO_OP by the preprocessor (an example is the statement numbered 6 in Figure 4.6(a)). The reason for these restrictions is simple. Since rule compaction generates a smaller rule set in the *most general case*, it must always be the case that applying the uncompacted rule set to *any* expression must have the same effect as applying the compacted rule set to the same expression (i.e., the two rule sets must be *equivalent*). Since the P2V preprocessor does not know the semantics of rule actions, it checks for syntactic hints (a TRUE test, descriptor assignment statements, NO_OP statements) that indicates that compaction yields an equivalent rule set. It may be possible to incorporate semantic knowledge of rules in the P2V preprocessor to produce even greater rule compaction; however, this is beyond the scope of this dissertation.

Figure 4.12 shows an example of rule compaction corresponding to each of the four general cases in Figure 4.11. Apart from the familiar relational operators and algorithms we have already encountered, Figure 4.12 introduces some new operators. An outer join [41] is a generalization of the traditional join operator (or natural join). It produces a stream that contains tuples whose join attributes are unmatched according to the join predicate. The outer join operation has two special cases depending on whether the outer or inner streams contribute unmatched tuples. These two operators are called LEFT_OUTER_JOIN and RIGHT_OUTER_JOIN, respectively. As shown on the left side of Figure 4.12(c), these two outer join operators can be transformed into the JOIN operator, if it is known that no unmatched tuples result.

The operator JOPR is exactly like JOIN, except that it expects its inputs to be sorted. The JOPR operator is needed in Prairie in order to introduce the explicit operator SORT. However, the P2V preprocessor identifies SORT as an enforcer-operator (as described in Section 4.2.1), and deletes it, resulting in the rule compaction shown in Figure 4.12(d).

4.6 Summary

In this chapter, we described the Prairie-to-Volcano preprocessor that generates a Volcano rule specification from a Prairie specification. The obvious advantage of using the preprocessor is that a DBI can specify rules in a high-level abstract framework (Prairie), while the preprocessor can generate a lower-level specification (Volcano) that takes into account implementation details. This generator-based approach means that the DBI is insulated from the actual implementation and internal representation of the features of the optimizer. In fact, it is possible to imagine a scenario where the preprocessor translates a Prairie specification suitable for another (non-Volcano) rule engine, if it turns out that the Volcano rule engine is not appropriate for the DBI. Another advantage of the P2V preprocessor, as described in Section 4.5, is that it is possible to "optimize" the rule set itself, i.e., to add "intelligence" in the preprocessor to generate small rule sets. This approach carries great potential, and, in particular, as we will see in Chapter 6, can be used to generate compact rule sets from the building-blocks used in reconfigurable optimizers.

High-level abstractions, like Prairie, can result in inefficient optimizers unless the P2V preprocessor translates Prairie rules into the appropriate implementation-level details of Volcano. In this chapter, we described the translation process. In the next chapter, we evaluate the efficiency of the resulting Volcano optimizer by optimizing some benchmark queries.

Chapter 5

Performance Results

In the last chapter, we described the P2V preprocessor that translates a Prairie rule set specification into a Volcano specification. One important aspect of this translation is performance. Namely, how does an optimizer generated from a Prairie specification compare with an optimizer generated from a hand-written Volcano specification? In this chapter, we present some experimental evidence that performance is not sacrificed by specifying rules in Prairie.

5.1 Overview

Performance of rule-based optimizers is one of the four goals that we identified in Chapter 1. This becomes clear when we consider that Prairie's programming environment has limited impact if it sacrifices performance in its quest for abstractions and extensibility. Consequently, some evidence is required that efficient optimizers can be generated from Prairie specifications.

In this chapter, we present experimental results using two different optimizers that are each specified using Prairie and Volcano. To avoid claims of "doctored up" Volcano specifications (to give Prairie an unfair advantage), we use two Volcano rule sets written by other researchers as points of comparison. The first rule specification is that of a simple centralized optimizer that contains a subset of the relational database operators, and has a fairly simple cost model. The second rule set specifies an optimizer for a large-scale object-oriented database. Both of these rule sets were re-engineered (i.e., rewritten) into Prairie specifications, from which Volcano specifications were obtained by using the P2V preprocessor described in the last chapter.[1] The two optimizers (P2V-generated and hand-written) were then compared for efficiency by optimizing a random set of operator trees.

The problem of developing a set of representative benchmark queries to test a query processing system is a well-researched topic [13, 14, 32, 42]. However, standard benchmark queries were not very useful in our case for various reasons. For example, the Wisconsin benchmark [13] consists of no queries with more than 3 joins; moreover, this benchmark was designed to test the performance of query processors on queries consisting of the entire set of relational operators (select, project, retrieve, join, update, insert, delete, aggregation) and with a variety of algorithms implementing the various operators. Since we are using pre-written Volcano rule sets that only contain a subset of these operators, we had to design our own benchmark queries.

In the following sections, we describe our experiments using the two rule specifications, the generation of random queries, and the productivity gains in using Prairie.

5.2 A Centralized Relational Query Optimizer

A simple centralized relational optimizer is bundled (as an example of a rule specification) with the Volcano optimizer generator; it was written by Graefe and his co-workers. It consists of 2 operators (RET and JOIN), 2 algorithms (File_scan and Merge_join), and 1 enforcer (Merge_sort). The semantics of these operators and algorithms are as defined in Chapter 2. There are 2 transformation rules, and 2 implementation rules. All support functions (as listed in Table 2.3) required for the operators, algorithms, and enforcers, and data structures representing the properties are defined in separate files (the host language is C). The entire rule specification package consists of approximately $1,400$ lines.

[1] Because of the way the P2V preprocessor represented properties, the P2V-generated Volcano specification was quite different from the hand-written Volcano specification.

Re-engineering the Volcano rule specification into Prairie involved several steps. First, we had to fix a few bugs in the original Volcano rule specification that precluded the optimizer from optimizing certain legal queries and that also produced non-optimal plans for some queries. Second, we transformed the multiple property sets into a single descriptor structure for the Prairie specification. This, in turn, required modifying property references in rule actions. Third, actions in support functions had to be specified as rule actions (since a Prairie rule contains *all* its actions). In the whole process of reconstituting the centralized optimizer specification, the behavior of the optimizer had to be kept unchanged (i.e., the Prairie optimizer had to generate the same optimal access plan as the Volcano optimizer, for the same operator tree).

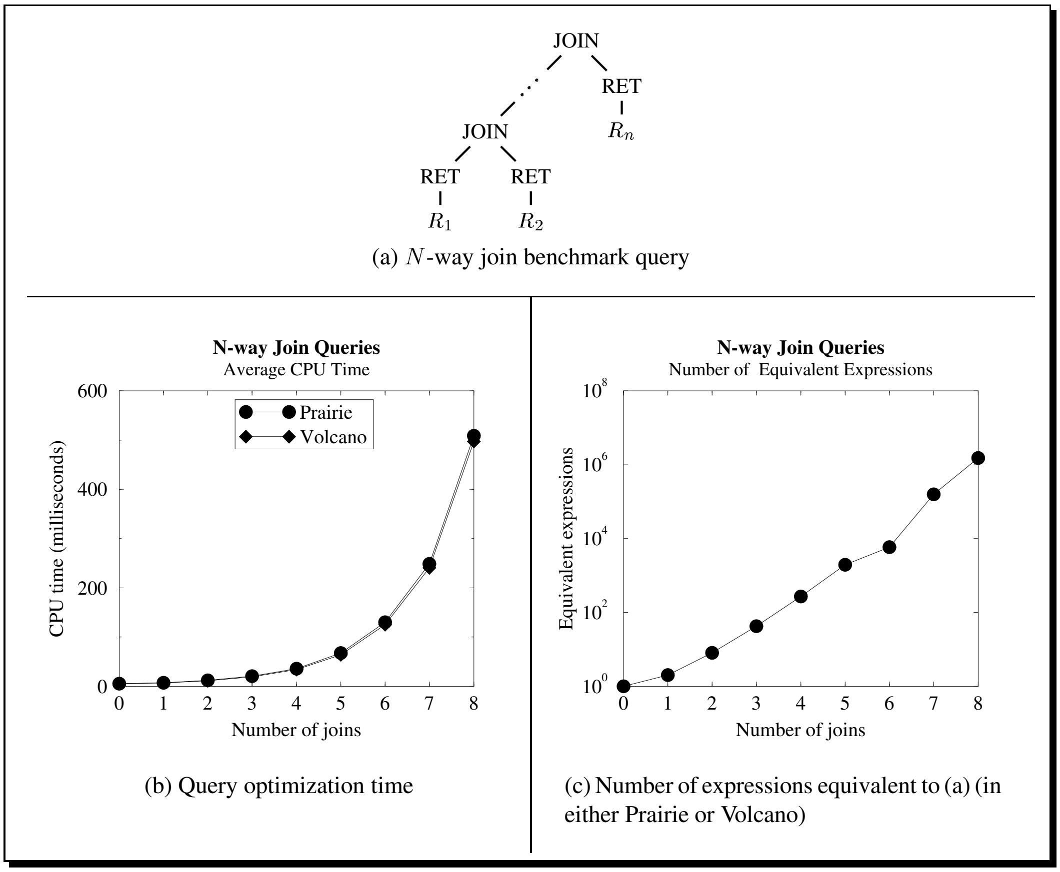
5.2.1 Programmer Productivity

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 suggests that Prairie excels on the latter, while offering modest reductions in the volume of code that needs to be written. The re-engineered Prairie specification of the simple centralized optimizer consisted of approximately $1,000$ lines, a savings of about 30% over the Volcano specification.[2] As mentioned above, however, savings in lines of code do not fully 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.

5.2.2 Generating Benchmark Queries

Our experiments using the centralized optimizer consisted of optimizing randomly generated queries using the two optimizers generated, respectively, using the Prairie and the hand-written Volcano specifications. (In the remainder of this chapter, we will use "Prairie"

[2] As reported in [21], the savings are even greater if the Volcano rule engine is modified slightly to remove calls to support functions that are grounded by the P2V preprocessor.



(a) N-way join benchmark query

(b) Query optimization time

(c) Number of expressions equivalent to (a) (in either Prairie or Volcano)

Figure 5.1: Benchmarking a simple centralized optimizer

and "Volcano" to denote these two approaches.) The queries represented left-deep N-way join operator trees, for varying values of N, as shown in Figure 5.1(a). Each operator tree also required its output to be returned sorted, but since Volcano considered SORT to be an implicit operator, it is not shown in Figure 5.1(a).

Random queries were generated for optimization using a uniform random number generator as follows. The set of stored files (relations) was fixed, as were the number of attributes in each file, their cardinalities, and record widths. Each query contained all operators in the Volcano specification. The queries generated varied in the order of the relations in the (left-deep) operator tree. Each join in the tree had an equijoin predicate. The two join attributes were chosen, at random, from the set of attributes in the outer and inner streams, respectively. The (single) tuple order for the (implicit) sort operator at the root was chosen as an attribute of one of the relations in the operator tree.

5.2.3 Performance Results Using the Centralized Optimizer

For each value N of the number of joins, we generated 10 *different* queries (as described above), and submitted it to both the Prairie and Volcano optimizers for optimization. Both optimizers produced the same optimal plan for each query. The run times of each optimizer were measured[3] using the GNU `time` command, and averaged over the 10 queries to generate the per-query optimization time. Thus, each point in our graphs represents the average CPU time for optimizing 10 different queries. All experiments were performed on a lightly loaded DECstation 5000/200 running Ultrix 4.2.

The optimization time for both approaches (Prairie and Volcano) are shown in Figure 5.1(b). The number N of joins was varied from 0 through 8. The results, as seen from the graph, are virtually identical for both Prairie and Volcano. That is, there is no loss of performance of the optimizer if it is specified using Prairie instead of Volcano.

It is interesting to note the number of equivalent expressions that are generated from the original N-way join query in Figure 5.1(a). This is shown in Figure 5.1(c). Both optimizers (Prairie and Volcano) result in the same number of equivalent expressions, which grows exponentially with N, the number of joins in the query. This corresponds to the exponential rise in optimization time as seen in Figure 5.1(b).

From these experiments with the simple centralized optimizer, it was evident to us that Prairie offers a modest reduction in code size, better programmability and clarity, with no performance degradation. It is important to show, however, that Prairie can also be used to specify rule sets for large-scale optimizers. This is described in the next section.

[3] To reduce measurement errors, the *same* query was optimized 10 times (in a loop) and the total time was divided by 10 to obtain the optimization time for any query.

5.3 The Texas Instruments Open OODB Query Optimizer

The Texas Instruments Open Object-Oriented Database Management System [50] is an open, extensible, object-oriented database system which provides users an architectural framework that is configurable in an incremental manner. It consists of three sets of modules: a core set providing low-level primitives for creating new environments, a set of functional modules that facilitates extensibility using functional requirements, and a meta-architecture module housing the extensibility concepts of Open OODB. Examples of the core set are communication and address space management, while examples of functional modules are persistence, distribution, and query processing. The meta-architecture module consists of events, sentries, and policy manager interfaces.

The query processing module provides users with a query language (OQL[C++]) based on SQL and C++. A query expressed in this high-level format is parsed and transformed into an operator tree suitable for optimization. From this operator tree, the query optimizer generates an optimal access plan, which is then transformed into a C++ program ready for execution.

The query optimizer in Open OODB [15] is generated using Volcano. It consists of 6 operators (SELECT, PROJECT, JOIN, RET, UNNEST (for set-valued attributes), and MAT (MATerialize; it is used for representing path expressions in a query)), 8 algorithms (Filter, Project, Hash_join, Ptr_hash_join, File_scan, Index_scan, Unnest, and Alg_assembly), and 1 enforcer (Enf_assembly). Currently, the Open OODB rule set consists of 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 DBI for specific needs. There are also *catalogs* which contain information about base classes that are used by the optimizer. The complete rule set for the Open OODB optimizer is shown in Appendix C.

5.3.1 Programmer Productivity

As in the case of the centralized optimizer, we re-engineered the Open OODB rule set into Prairie, and then generated a Volcano specification by using the P2V preprocessor. Using lines of code as a crude metric of programmer productivity, there was about 10% savings realized in Prairie over Volcano.[4] However, an even more important increase in programmer productivity resulted in using the abstractions and encapsulations in Prairie; it took almost 4 man-months to reconstitute the Prairie specifications for the Open OODB optimizer because of the widespread use of implementation-level details in the Volcano specification of the optimizer, a specification that ought to be programmed at the *conceptual* level.

As a sidenote, the Prairie specification of the Open OODB optimizer consisted of 22 T-rules and 11 I-rules (the increased number of rules is because, as mentioned in Chapter 3, Prairie does not allow implicit rules). The P2V-generated Volcano rule set contained 17 transformation rules and 9 implementation rules, the same as the original hand-written Volcano specification.

5.3.2 Generating Benchmark Queries

Our experiments using the Open OODB consisted of optimizing 8 different queries using the Prairie and Volcano optimizers. There were 4 distinct expressions that were used to generate the queries used in the experiments; these are shown in Figure 5.2. Each expression represents an N-way join query, for varying values of N.

The first expression, E1, is a simple retrieval and join of stored files; these files represent 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 scope) before the join. (This is an abstract representation of path expressions in object-oriented queries.) 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).[5]

[4] 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.

[5] 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 implementation rule and no transformation 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 transformation rule and one implementation rule; including it in our queries would have increased the number of parameters that could affect our run times. We preferred to concentrate on

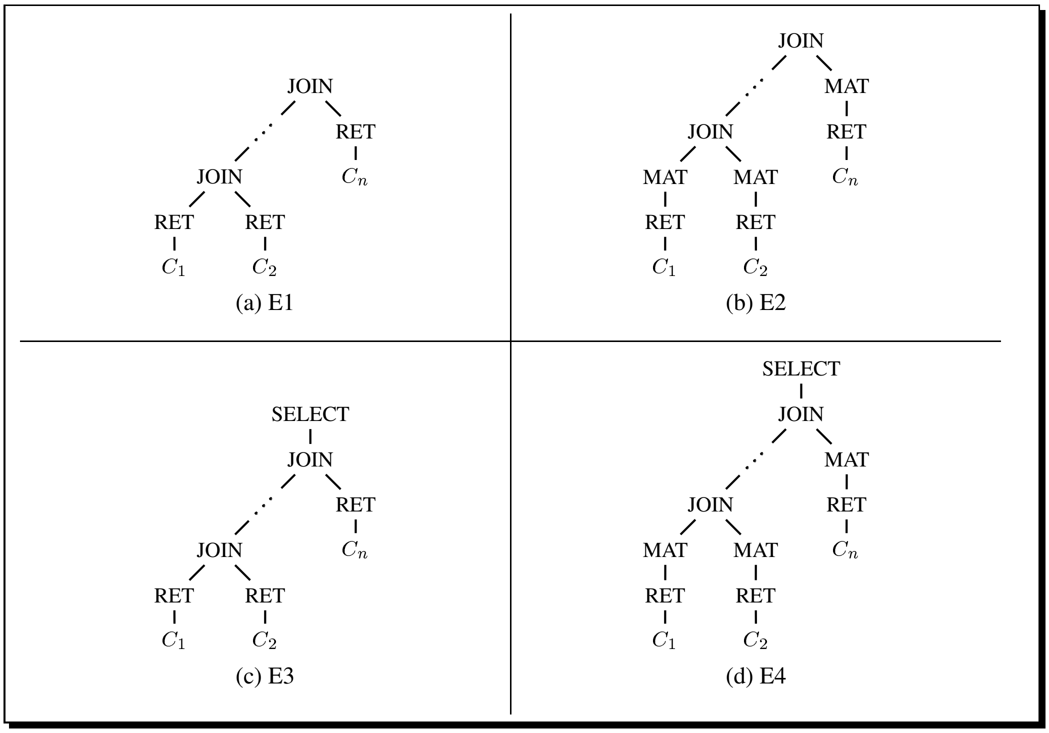


Figure 5.2: Expressions used in generating benchmark queries for Open OODB

Query	Indices?	Expression	Rules matched	
			transformation rules	**implementation rules**
E11	No	E1	3	3
E12	Yes			
E21	No	E2	8	4
E22	Yes			
E31	No	E3	9	5
E32	Yes			
E41	No	E4	16	7
E42	Yes			

Table 5.1: Queries used in benchmarking the Open OODB optimizer

As in the case of the centralized optimizer, there are many parameters that can be varied when benchmarking a query optimizer. Since our objective was to verify that the Prairie

simple join expressions.

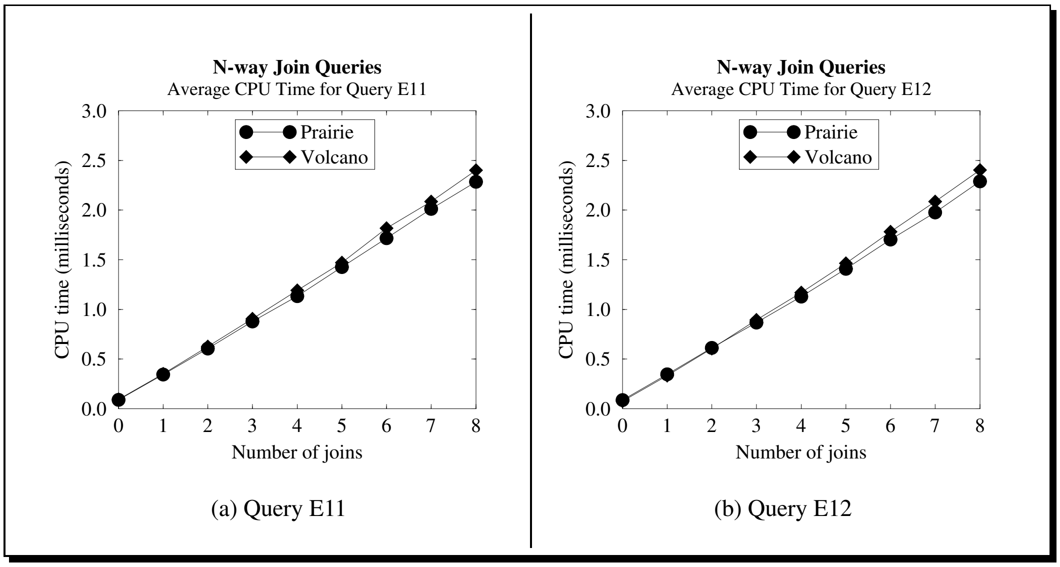
approach did not sacrifice efficiency, our criteria for the queries was that they test a majority of the rules, with varying properties of the base classes. To this end, we tested the Prairie (and Volcano) optimizer with 8 different queries (shown in Table 5.1). The eight queries, E11 through E42, are derived from the 4 expressions in Figure 5.2. 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 was in the properties of the base classes: the first query did not contain any indices on any classes, whereas the second one contained a single index on each base class occurring in the expression.

In expressions where a SELECT is present (E3 and E4), the selection predicate was a conjunction of equality predicates $bc_i == const_i$, where bc_i was an attribute of class C_i, and $const_i$ was 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 (E32 and E42, in Table 5.1), the (single) indexed attribute 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 equijoin predicates. The choice of join predicates was such that the queries corresponded to linear query graphs; furthermore, each join predicate corresponded to an implementation using a one-way pointer between the join attributes of the base classes. For example, if e is an object of the Employee class, and p an object of the Person class, and the spouse method (i.e., attribute) of Employee returns an instance of Person, then the join predicate $e.\text{spouse} == p$ was implemented as a pointer from the spouse method of Employee to the Person class.

Table 5.1 also shows the number of Volcano transformation and implementation rules that are matched by each query. These are the rules whose left hand sides match a sub-expression. However, not all the rules were necessarily applicable. For instance, an implementation rule with an index scan would not apply to E21, although it might apply to E22.

Random queries were generated for optimization using a random number generator



(a) Query E11

(b) Query E12

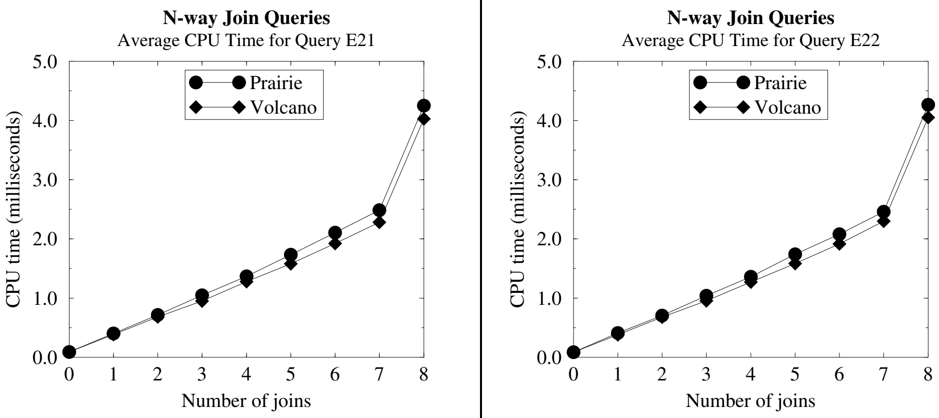
Figure 5.3: Benchmarking the Open OODB optimizer — queries E11 and E12

as follows. The set of stored files (classes) was fixed. Each query contained a random order of classes in the (left-deep) operator trees. For each query, the catalog information (cardinality of classes, record widths, etc.) was varied. The join and selection predicates (where applicable) were generated subject to the constraints described above.

5.3.3 Performance Results Using the Open OODB Optimizer

Queries E11 through E42 were optimized for increasing number N of joins. For each query and a fixed value of N, we varied the cardinalities, record widths, etc. 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[6] using the GNU `time` command. All experiments were performed on a lightly loaded DECstation 5000/200 running Ultrix 4.2.

[6] 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 obtain optimization time for a given query.



(a) Query E21

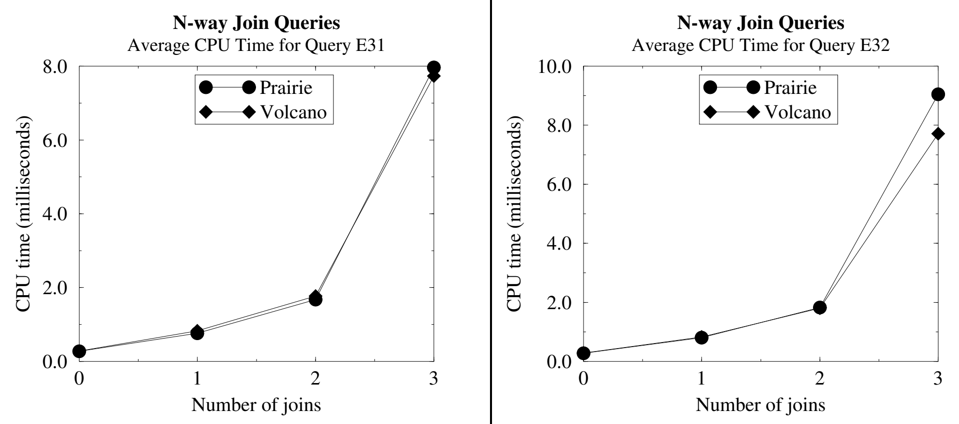
(b) Query E22

Figure 5.4: Benchmarking the Open OODB optimizer — queries E21 and E22

The optimization times for each query for both approaches (Prairie and Volcano) are shown in Figures 5.3 through 5.6. The number of joins N in each set of graphs was varied to a maximum of 8, or until virtual memory was exhausted.

The first set of graphs (Figures 5.3(a) and 5.3(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 rule set had only two join algorithms (Ptr_hash_join and Hash_join), neither of which makes use of any indices.

The second set of graphs (Figures 5.4(a) and 5.4(b)) shows the results of optimizing E21 and E22. Here, as in Figures 5.3(a) and 5.3(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) for storing the search space results in a lot of thrashing at this point. We speculate that in systems with more virtual memory, the graphs will be smoother.



(a) Query E31 (b) Query E32

Figure 5.5: Benchmarking the Open OODB optimizer — queries E31 and E32

The third and fourth sets of graphs in Figures 5.5 and 5.6 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 in our synthetically generated queries). 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 preprocessor introduces (because of shared property structures using pointers; see Figure 4.2). 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.

It is instructive to compare the memory requirements in the optimization of the various queries. As mentioned earlier, the Volcano rule engine implements its optimization in main memory, i.e., it stores the search space of a query in main memory. This means that memory can become a bottleneck in the size of queries that can be optimized. Figure 5.7 compares the number of equivalent expressions generated for each of the expressions E1 through E4. It is important to note that not all equivalent expressions lead to valid access

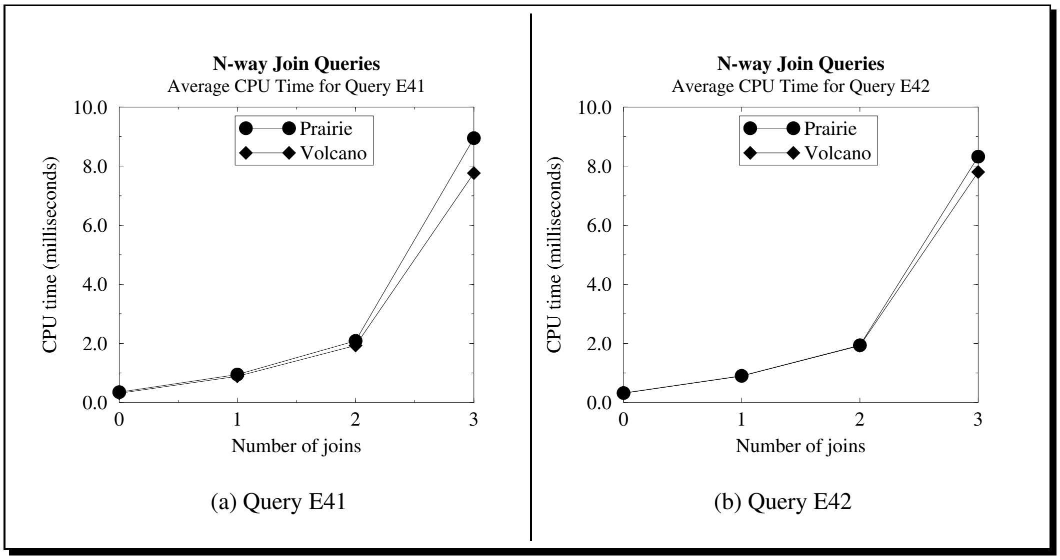


Figure 5.6: Benchmarking the Open OODB optimizer — queries E41 and E42

plans, but an exhaustive search strategy (like Volcano) requires the expressions to be stored because they might be reused in other expressions.

We can see from Figure 5.7 that the search space size increases by large magnitudes from E1 through E4. For instance, the number of equivalent expressions for E4 is about a million, for 3-way joins. This is the reason why, for example, we could not optimize expressions containing more than 3 joins in E4 (Figure 5.6). The main reason for this tremendous use of memory is the presence of the SELECT and MAT operators which increases the number of rules that an expression matches (see Table 5.1). These rules basically permute the SELECT and MAT operators with the others (JOIN and RET), thus creating a large number of equivalent expressions.

It is also interesting to compare the memory usage in the Open OODB optimizer with the simple centralized optimizer in Section 5.2. Expression E1 in Figure 5.2(a) (which only contains the relational operators JOIN and RET) is similar to the expression used in the centralized optimizer (Figure 5.1(a)). However, the memory usage, as measured by the number of equivalent expressions (Figures 5.7 and Figures 5.1(c)), is vastly different in the two

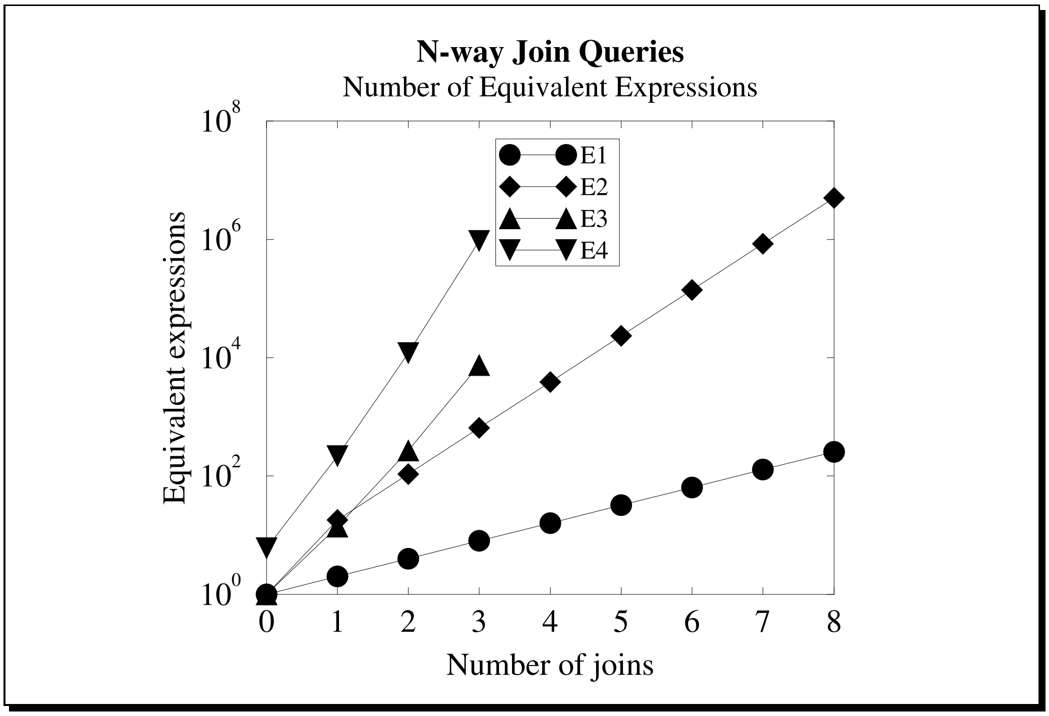


Figure 5.7: Equivalent expressions in the Open OODB optimizer

cases (256 for E1 versus 1.5 million for the centralized optimizer, for 8-way joins). What accounts for this huge difference? The key lies in the implementation of the join predicates. In the case of Open OODB optimizer, we designed the database such that join predicates were implemented by one-way pointers[7] (links), thus, the join commutativity transformation rule did not apply; this cuts down on the number of equivalent expressions tremendously. In the simple centralized optimizer, attributes in a join predicate were implemented as normal columns; thus, the join commutativity rule could permute join streams and generate equivalent expressions.

[7]The reason for this choice was that if join predicates were implemented as set-valued attributes, then join commutativity would increase the number of equivalent expressions to an even larger explosion than that shown in Figure 5.7. This would mean, for example, that it might not have been possible to optimize queries E31 and E32 (generated from E3) even for 3-way joins.

5.4 Summary

In this chapter, we presented experimental evidence that Prairie specifications do not sacrifice performance compared to Volcano. Two sets of experiments were performed, one with a simple centralized optimizer and another with a large-scale object-oriented optimizer; each demonstrated comparable performances between the Prairie and Volcano optimizers. There were also gains in lines of code, and, most importantly, gains in abstractions and extensibility. This was made possible by using the P2V preprocessor to bridge the gap between Prairie and Volcano.

In the next chapter, we describe how to use Prairie to build reconfigurable optimizers. This is done by using rule sets that are constructed using reusable, recombinable building-blocks. We will also show how the P2V preprocessor can generate monolithic optimizers from these reconfigurable specifications, optimizers that are as efficient as any hand-written rule set.

Chapter 6

Reconfigurable Optimizers

In the previous chapter, we presented experimental results to evaluate the performance of Prairie optimizers. In this chapter, we describe how the Prairie framework can be extended to construct optimizers that satisfy the fourth goal listed in Chapter 1, reconfigurability. That is, how can optimizers be changed quickly and seamlessly for use in a different database system? One technique to do this is presented in this chapter. The extensions described here were made easier by the clean design and abstractions embodied in Prairie.

6.1 Overview

Large-scale software development is an expensive undertaking. An approach that has been proposed is that of using software components [9, 11, 26, 27]. These components, typically used as building-blocks to construct larger systems, have well-defined and standardized imported (input) and exported (output) interfaces. A system is developed by fitting together components whose interfaces match one another. This approach not only has the advantage of being able to assemble modular systems from parts, it also allows the reuse of the same components in other software systems.

Database management systems (DBMSs) are large-scale software systems that can also benefit from a component-based approach. The first system to demonstrate the feasibility of this idea was Genesis [2, 3, 5, 6, 12]. A DBI created a customized DBMS by composing several pre-defined components with standard interfaces. The result was a DBMS that was tailored to the needs of a particular application.

The ideas incorporated in Genesis are not unique to databases; in fact, the GenVoca paradigm [7–9, 11] has shown that the same component-based approach also applies to networks, avionics, file systems, and data structures. The P2 generator [10, 11], in particular, has demonstrated that very efficient container data structures can be generated from a small family of plug-compatible components.

Query optimizers are integral to most DBMSs. They also tend to be a significant and complex part of a DBMS. The earliest query optimizer proposed, System R [46], was monolithic in nature. That is, the three features of a query optimizer, namely, search space, cost model, and search strategy, were all hard-wired into a single module that represented the optimizer. Many contemporary optimizers still follow this model. However, database systems are increasingly being used to store traditional relational data as well as non-traditional data (e.g., a database could contain relational as well as object-oriented and multimedia data); this requires a query optimizer that can be quickly and efficiently modified to deal with the changing properties of the underlying data. For example, in a cost model that is a weighted sum of the CPU and I/O costs, it might be realistic to weight the I/O cost more than the CPU cost when optimizing multimedia queries (since multimedia objects tend to be extremely large, I/O costs might dominate CPU cost) than when dealing with relational data. As another example, stored files with pointers between them would suggest specific join orders than when no pointers are present. Thus, it might make sense to modify or change certain rules depending on the specific application at hand.

In this chapter, we describe a building-blocks approach to the construction of rule-based optimizers using Prairie. The goal is to generate families of rule sets automatically where each rule set corresponds to some classical relational optimizer (centralized, distribution, replication, etc.). We discuss the model, together with a few simple examples, and show how the P2V preprocessor, described in Chapter 4, can be used to quickly generate efficient rule sets from component-based specifications. Currently, the P2 generator [10, 11] does not have a framework for quickly generating efficient full-fledged optimizers. The ultimate goal of the research described in this chapter is to integrate the Prairie specification language into P2 to enable automatic generation of rule-based optimizers. Thus, the goal of this chapter is to lay a foundation for using Prairie to build optimizers for novel applications (e.g., container data structures using P2).

6.2 Layered Rule-Based Optimizers

6.2.1 Layers

In the Prairie framework described in Chapter 3, 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, except for tests in a rule, the search space is determined entirely by the particular search strategy implemented by the rule engine.

Prairie allows a DBI to specify rules in encapsulated components. These components are called *layers* and can either be defined by a DBI, or 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 operations. The general form of a layer is shown in Figure 6.1(a). A layer translates an abstract expression consisting of *abstract* operators $\{O_1, \ldots, O_n\}$ to a set of concrete 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.

Layers in Prairie follow the GenVoca paradigm [9]. Thus, layers with the same in-

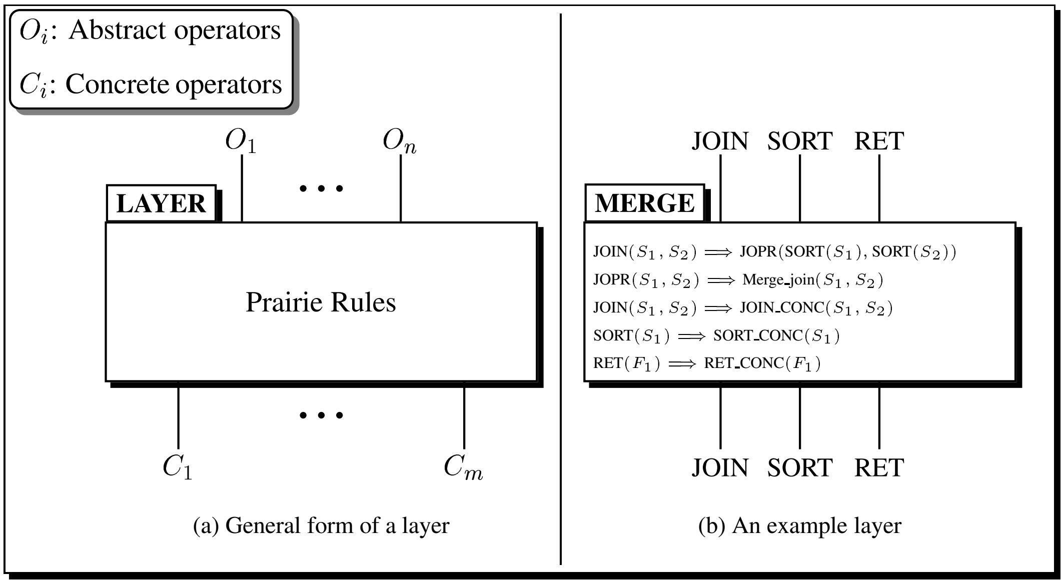


Figure 6.1: General form of a Prairie layer and an example

terfaces constitute a *realm*; each layer is plug-compatible and interchangeable with the other layers in its realm. Layers in Prairie are usually *symmetric*; that is, they have export interfaces that are the same as their import interfaces [9]. Typically, these interfaces are comprised of all the abstract operators in the particular database schema at hand. Symmetric layers can, thus, be composed in arbitrary ways; this provides DBIs many ways to construct optimizers using the same layers. However, not all the compositions are necessarily meaningful or correct. Batory and Geraci [8] describe methods for validating the correctness of component compositions. We will assume that compositions of our layers are always correct. We will not address the problem of validating layer compositions; we are aware these problems are present.

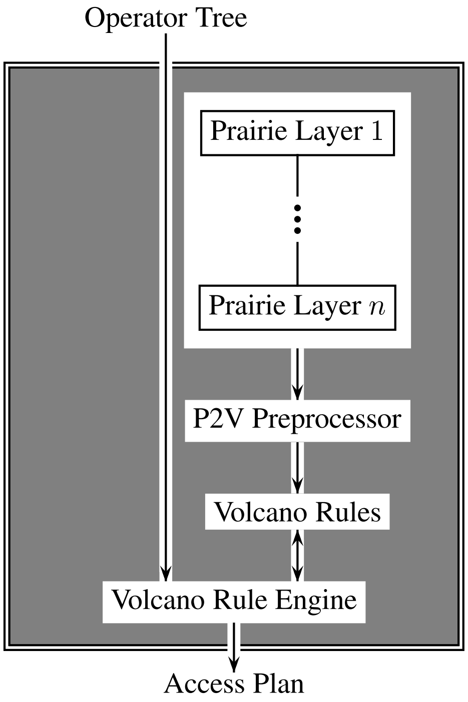
To allow optimizer specifications using layers, the Prairie specification language defined in Chapter 3 was extended in two ways. First, rules can now be declared as belonging to a specific layer (a layer declaration demarcates rule definitions). Second, the optimizer can be defined as a composition of layers. This composition is defined as a type expression [9] expressed as a linear sequence of layers. This is described in more detail in the next section.

An example layer is shown in Figure 6.1(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). Note that the concrete operators bear the same name as the abstract operators, implying that the MERGE layer is symmetric. 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 JOIN operator that will be transformed into an algorithm by another lower layer. Since the MERGE layer is symmetric (i.e., has the same exported and imported interfaces), we need a way to distinguish the abstract and concrete operators in the rules. Prairie allows a DBI to append "_CONC" to an operator to refer to a concrete operator. Thus, "JOIN" refers to the abstract join operator and "JOIN_CONC" refers to the concrete join operator.

6.2.2 Composing Layers

In a layered Prairie optimizer specification, a DBI can define layers, each consisting of T-rules and I-rules. An optimizer can then be constructed by composing layers in a linear order. That is, the specification of an optimizer now consists of layers stacked upon one another, instead of a set of rules specified in a single component. This is shown schematically in Figure 6.2(a). The Prairie syntax for specifying rules in individual layers and the layer composition are shown in Figure 6.2(b). The composition shown in this figure, for example, represents an optimizer with the SEQUENTIAL, MERGE, SORT, and RET layers arranged in that order. This can be viewed as composing *parameterized* layers, where each layer (except the terminal layer, RET) has a single parameter that is another layer. Note that not all layers defined by the DBI are used in the composition; the P2V preprocessor deletes unused layers. A more general framework of layer composition in the future will involve hierarchical compositions of layers (not just linear compositions).

The semantics of layer composition is as follows. For *each new* expression generated by the optimizer, the rule engine applies rules to the expression in the order of layering.



(a) Schematic representation (cf. Figure 3.1)

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

%layer SEQUENTIAL
%trule JOIN(S_1, S_2) $\Longrightarrow$! JOIN(S_2, S_1)
%trule JOIN(JOIN(S_1, S_2), S_3) $\Longrightarrow$ JOIN(S_1, JOIN(S_2, S_3))
%trule JOIN(S_1, S_2) $\Longrightarrow$ JOIN_CONC(S_1, S_2)
%trule SORT(S_1) $\Longrightarrow$ SORT_CONC(S_1)
%trule RET(F_1) $\Longrightarrow$ RET_CONC(F_1)

%layer SORT
%irule SORT(S_1) $\Longrightarrow$ Merge_sort(S_1)
%irule SORT(S_1) $\Longrightarrow$ Null(S_1)
%trule JOIN(S_1, S_2) $\Longrightarrow$ JOIN_CONC(S_1, S_2)
%trule SORT(S_1) $\Longrightarrow$ SORT_CONC(S_1)
%trule RET(F_1) $\Longrightarrow$ RET_CONC(F_1)

%layer MERGE
%trule JOIN(S_1, S_2) $\Longrightarrow$ JOPR(SORT(S_1), SORT(S_2))
%irule JOPR(S_1, S_2) $\Longrightarrow$ Merge_join(S_1, S_2)
%trule JOIN(S_1, S_2) $\Longrightarrow$ JOIN_CONC(S_1, S_2)
%trule SORT(S_1) $\Longrightarrow$ SORT_CONC(S_1)
%trule RET(F_1) $\Longrightarrow$ RET_CONC(F_1)

%layer DISTRIBUTION
%trule JOIN(S_1, S_2) $\Longrightarrow$ JOIN_CONC(XFER(S_1), XFER(S_2))
%irule XFER(S_1) $\Longrightarrow$ Ship(S_1)
%irule XFER(S_1) $\Longrightarrow$ Null(S_1)
%trule JOIN(S_1, S_2) $\Longrightarrow$ JOIN_CONC(S_1, S_2)
%trule SORT(S_1) $\Longrightarrow$ SORT_CONC(S_1)
%trule RET(F_1) $\Longrightarrow$ RET_CONC(F_1)

%layer REPLICATION
%trule RET(F_1) $\Longrightarrow$ RET_CONC(F_{11})
%trule RET(F_1) $\Longrightarrow$ RET_CONC(F_{12})

%layer RET
%irule RET(F_1) $\Longrightarrow$ File_scan(F_1)

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

Figure 6.2: The Prairie layered optimizer paradigm

Thus, rules in a layer are applied to an expression until no more applicable rules remain in that layer; the rule engine then applies rules in the next layer, and so on. Layer composition, thus, defines a hierarchy of rules that determines the order in which rules are applied. This hierarchy provides the DBI with another degree of control over the search space of the optimizer.

One question that might arise 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), all the layered optimizers that we constructed (and that are described in this chapter) 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 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.

6.3 Examples of Layered Optimizers

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

6.3.1 Example Layers

Some examples of layers in a Prairie specification are shown in Figure 6.3.[1] 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 6.3, the SORT operator is transformed to either the Merge_sort or the Null algorithm. Other sort

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

SEQUENTIAL

JOIN(S_1, S_2) $\Longrightarrow$! JOIN(S_2, S_1)
JOIN(JOIN(S_1, S_2), S_3) $\Longrightarrow$ JOIN(S_1, JOIN(S_2, S_3))
JOIN(S_1, S_2) $\Longrightarrow$ JOIN_CONC(S_1, S_2)
SORT(S_1) $\Longrightarrow$ SORT_CONC(S_1)
RET(F_1) $\Longrightarrow$ RET_CONC(F_1)

SORT

SORT(S_1) $\Longrightarrow$ Merge_sort(S_1)
SORT(S_1) $\Longrightarrow$ Null(S_1)
JOIN(S_1, S_2) $\Longrightarrow$ JOIN_CONC(S_1, S_2)
SORT(S_1) $\Longrightarrow$ SORT_CONC(S_1)
RET(F_1) $\Longrightarrow$ RET_CONC(F_1)

MERGE

JOIN(S_1, S_2) $\Longrightarrow$ JOPR(SORT(S_1), SORT(S_2))
JOPR(S_1, S_2) $\Longrightarrow$ Merge_join(S_1, S_2)
JOIN(S_1, S_2) $\Longrightarrow$ JOIN_CONC(S_1, S_2)
SORT(S_1) $\Longrightarrow$ SORT_CONC(S_1)
RET(F_1) $\Longrightarrow$ RET_CONC(F_1)

DISTRIBUTION

JOIN(S_1, S_2) $\Longrightarrow$ JOIN_CONC(XFER(S_1), XFER(S_2))
XFER(S_1) $\Longrightarrow$ Ship(S_1)
XFER(S_1) $\Longrightarrow$ Null(S_1)
JOIN(S_1, S_2) $\Longrightarrow$ JOIN_CONC(S_1, S_2)
SORT(S_1) $\Longrightarrow$ SORT_CONC(S_1)
RET(F_1) $\Longrightarrow$ RET_CONC(F_1)

REPLICATION

RET(F_1) $\Longrightarrow$ RET_CONC(F_{11})
RET(F_1) $\Longrightarrow$ RET_CONC(F_{12})

RET

RET(F_1) $\Longrightarrow$ File_scan(F_1)

Figure 6.3: Example layers. For clarity, all rule actions are omitted.

algorithms can either be introduced in this or other SORT layers.[2] The remaining rules transform abstract operators into concrete operators.

The MERGE layer transforms the JOIN operator into the Merge_join algorithm.[3] Other join algorithms can either be encapsulated in the MERGE layer, or in a separate layer.

The DISTRIBUTION layer encapsulates the distribution of stored files 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. As in the MERGE layer, the XFER operator (denoting the transfer of streams between sites) is an enforcer-operator, explicitly introduced via a T-rule. One

[2] In fact, we can use any sort algorithm in this layer. This is true for other layers as well.

[3] Note that the JOPR operator in the MERGE layer is required because all enforcer-operators (like SORT) are explicit in Prairie. However, since the JOPR operator (presumably) does not occur in other layers, it is not an exported operator of the MERGE layer, and thus, is not transformed into a concrete operator.

algorithm that implements the XFER operator is Ship. The Ship algorithm here is assumed to be a block transfer of streams (as in R* [20, 37, 45]); other transfer strategies (e.g., tuple-at-a-time) could be defined in this or other layers encapsulating distribution transformations.

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 physical file for each stored file in the database. The REPLICATION layer translates a stored file reference into a reference to one of the physical replicas of the file. (In Figure 6.3, we assume each stored file 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 imports the RET operator, but doesn't export any concrete operator. This, in turn, implies that the RET layer, as defined, is typically last in a layer composition.

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

6.3.2 An Optimizer for a Centralized Database

An optimizer for a centralized database is shown in Figure 6.4. It is formed by composing the SEQUENTIAL, MERGE, SORT, and RET layers. 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 6.4. (Horizontal 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

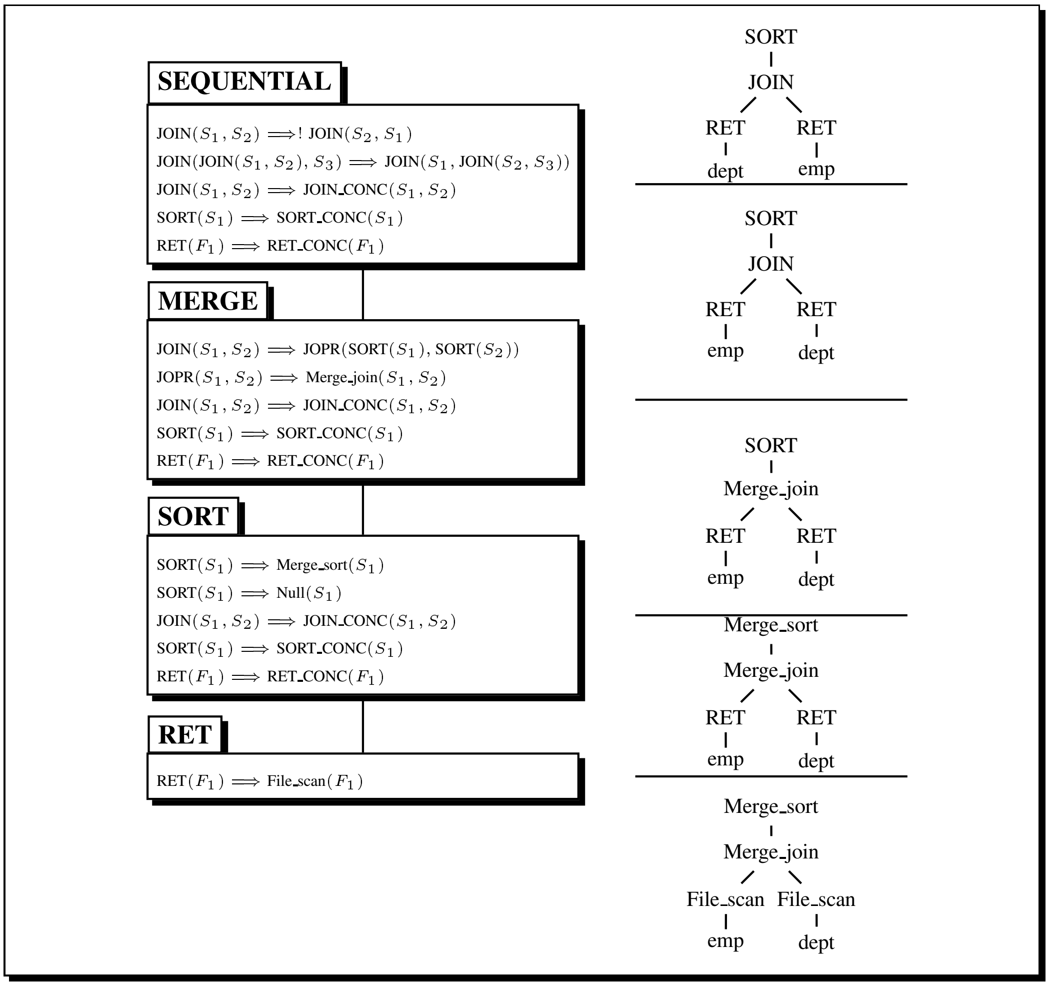


Figure 6.4: An optimizer for a centralized database and an example transformation

unchanged to the MERGE layer. The example transformation shown in Figure 6.4 is, thus, *one* of many that can be 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 is described in Section 6.4.

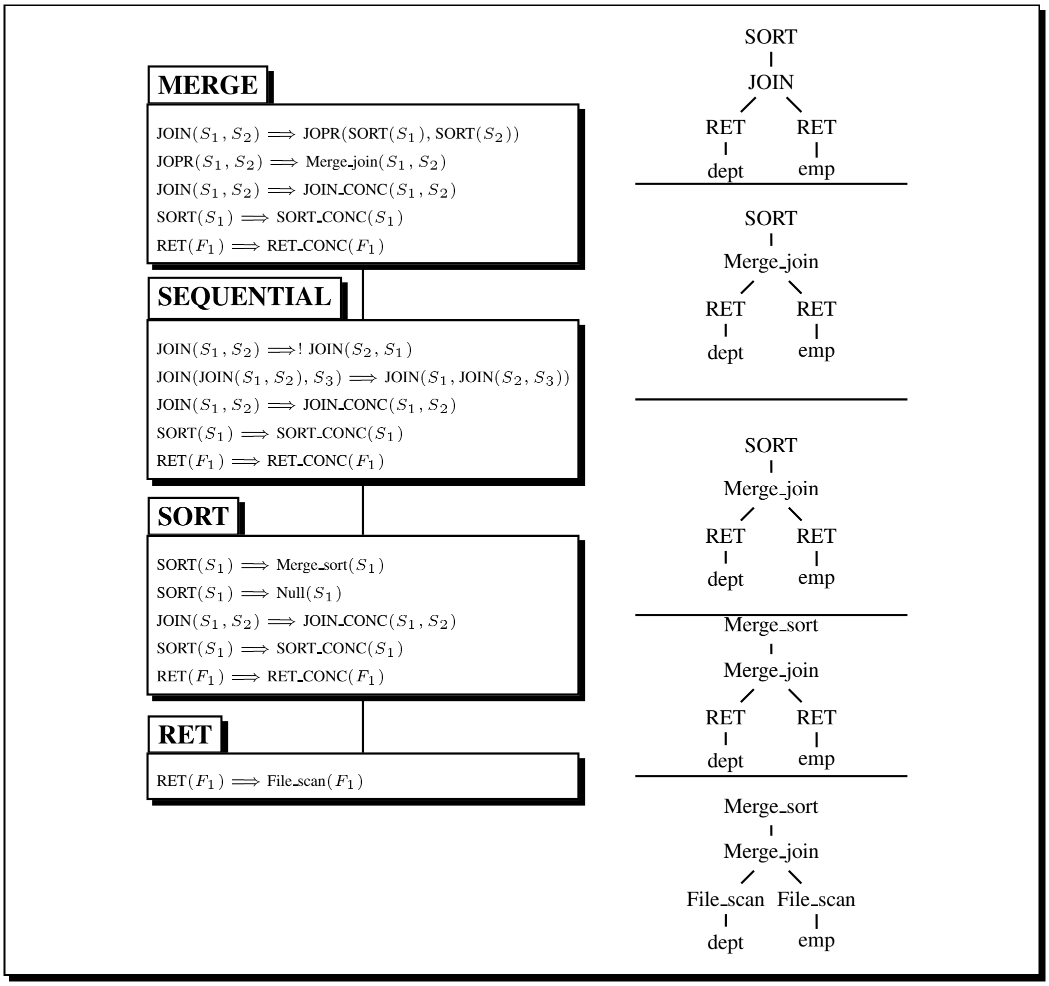


Figure 6.5: An alternative optimizer for a centralized database and an example transformation

6.3.3 Another Optimizer for a Centralized Database

An alternative optimizer for a centralized database is shown in Figure 6.5. It is similar to the optimizer in Figure 6.4 with the SEQUENTIAL and MERGE layers reversed. Again, an example transformation of a two-way join expression is shown in Figure 6.5. The important point to note here is that the MERGE layer first transforms a JOIN operator into the Merge_join algorithm, then the SEQUENTIAL layer applies the join associativity and commutativity rules. However, any remaining JOIN operators beyond the SEQUENTIAL layer are not transformed into any join algorithms (since the two lower layers, SORT and RET only transform the SORT and RET operators, respectively). Thus, the transformations in the SEQUENTIAL layer are never applied to generate valid access plans. In other words, the layered optimizer specification shown in Figure 6.5 would remain unchanged in its behavior if the SEQUENTIAL layer were removed.[4]

Comparison of the two alternative centralized optimizers in Figures 6.4 and 6.5 shows that the one in Figure 6.4 is more general since its search space is larger. This simple example shows that the order of layer stacking can alter the search space without any change in the search strategy. This, in turn, implies that the optimal access plan depends on the order of layers.

6.3.4 An Optimizer for a Distributed Database

An optimizer for a distributed database is shown in Figure 6.6. It is formed by composing the SEQUENTIAL, DISTRIBUTION, MERGE, SORT, and RET layers. In other words, it is obtained by adding the DISTRIBUTION layer to the centralized optimizer in Figure 6.4. 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. An example transformation is shown in Figure 6.6.

6.3.5 An Optimizer for a Replicated Database

An optimizer for a replicated database is obtained from a centralized optimizer by inserting the REPLICATION layer. This is shown in Figure 6.7. It is formed by composing the SE-

[4]The layer ordering of Figure 6.5 is actually an example of a composition error, in that the SEQUENTIAL layer should lie above all join layers for it to be useful. Errors like this can be detected using Batory and Geraci's algorithms [8].

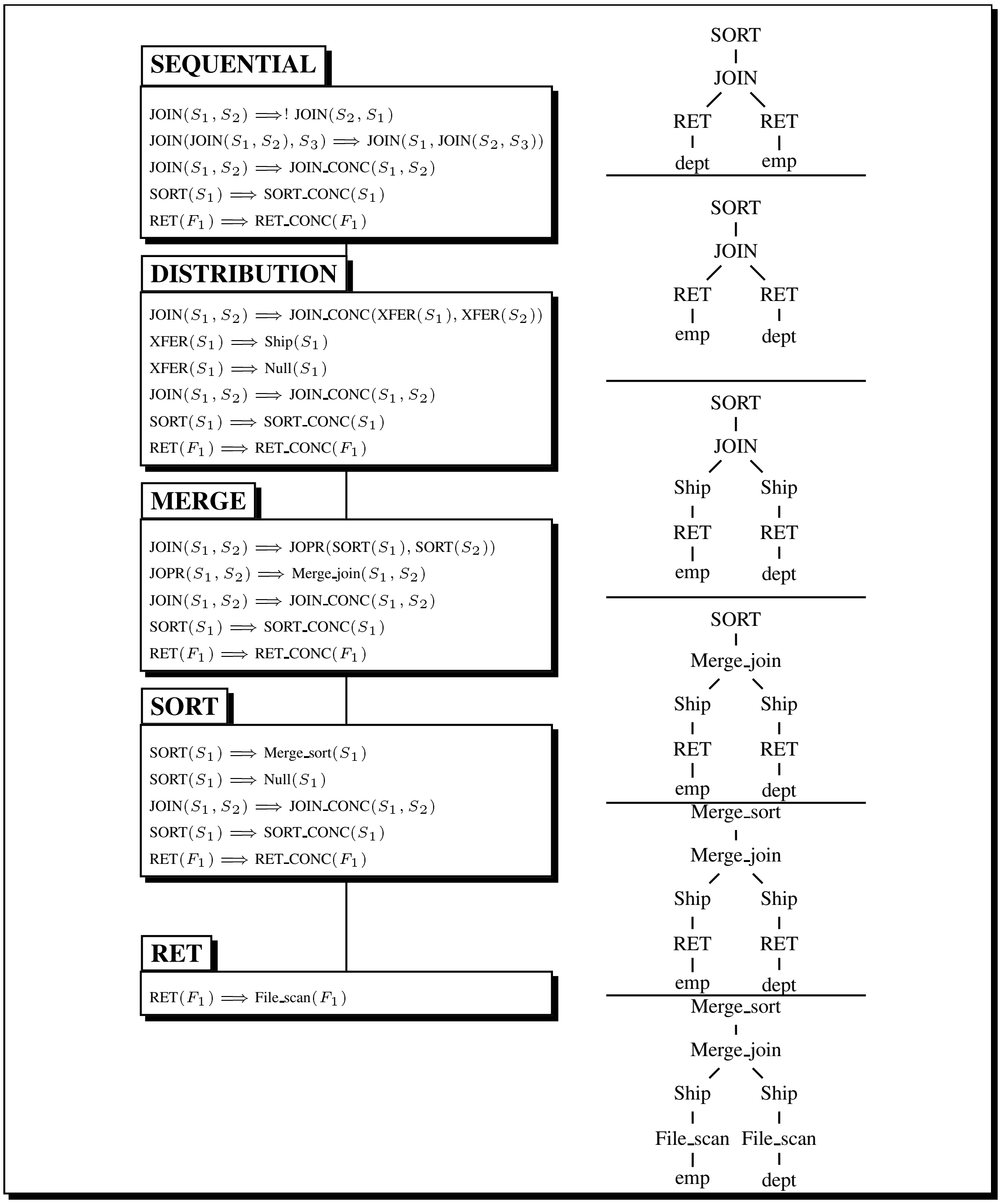


Figure 6.6: An optimizer for a distributed database and an example transformation

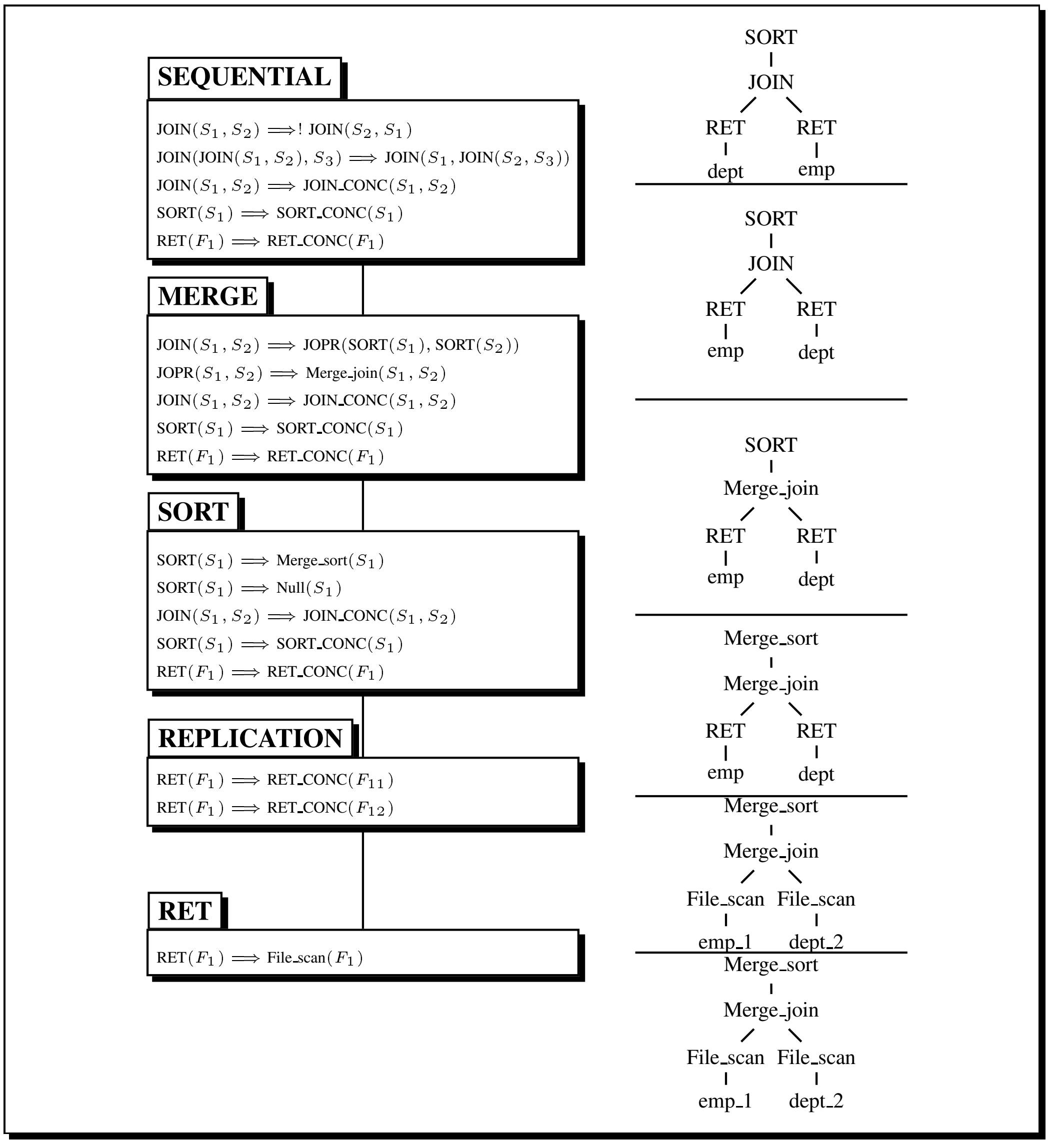


Figure 6.7: An optimizer for a replicated database and an example transformation

QUENTIAL, MERGE, SORT, REPLICATION, and RET layers. The SEQUENTIAL layer applies the join associativity and commutativity T-rules to a join expression. 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. The REPLICATION layer transforms references to logical stored files into their physical replicas. Finally, the RET layer transforms the RET operator into the File_scan algorithm. An example transformation is shown in Figure 6.7. Note that references to the stored files dept and emp are translated by the REPLICATION layer into references to their corresponding physical stored files.

6.4 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 6.3, 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. In this section, we discuss how the P2V preprocessor can be used to compact layered optimizers 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 define 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 6.4, the JOIN_CONC operator in the SEQUENTIAL layer corresponds to the JOIN operator in

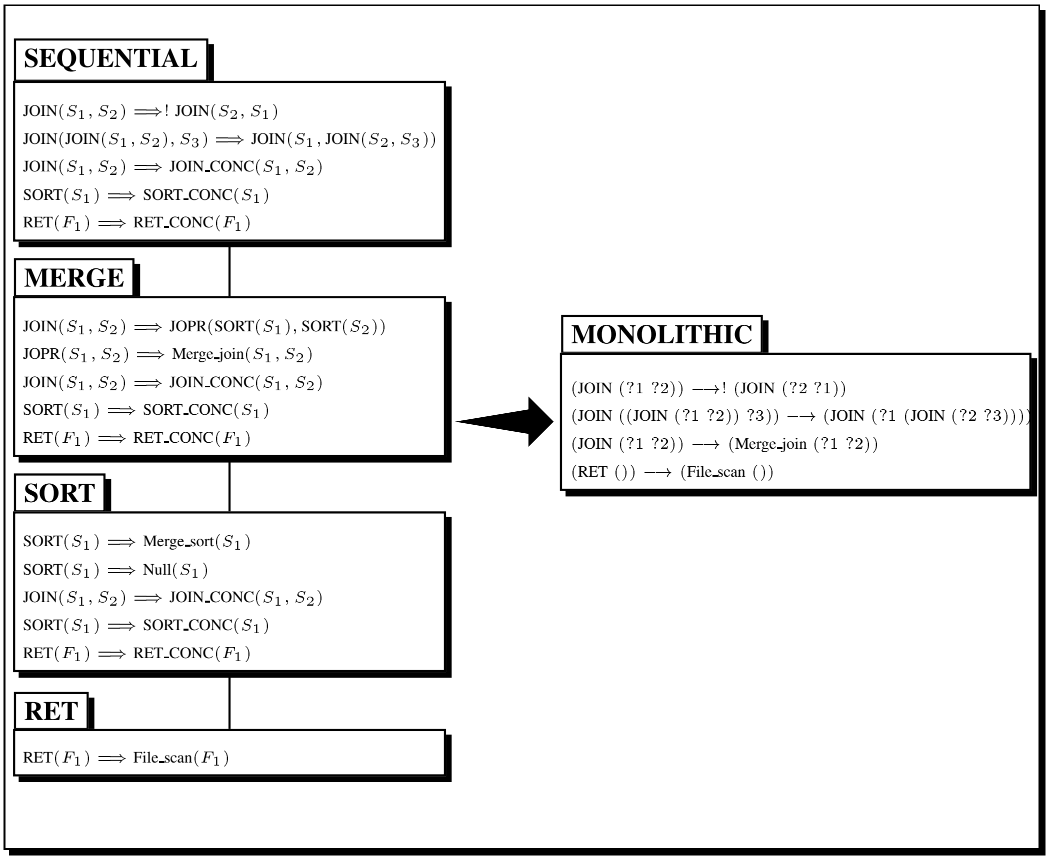


Figure 6.8: Compacting the layered centralized Prairie rule set in Figure 6.4

the MERGE layer. Once this correspondence is established, the P2V preprocessor can use the same rule compaction techniques described in Section 4.5 (and depicted in Figure 4.11) to combine all the layers together into a single, monolithic rule specification. For the centralized optimizer shown in Figure 6.4, this process results in the compaction illustrated in Figure 6.8. It is interesting to note that the monolithic rule set obtained by layer compaction is basically the same[5] (except for rule actions as shown in Figure 6.9) as one that might have been hand-written by a DBI; the layered specification, however, affords more degrees of ex-

[5] Note, especially, that rules (e.g., the JOIN to JOIN_CONC transformation in the MERGE layer) that simply transform abstract operators to concrete operators are discarded as a direct consequence of the compaction process.

$\text{JOIN}(\text{JOIN}(S_1, S_2) : \mathbf{D_4}, S_3) : \mathbf{D_5} \Longrightarrow \text{JOIN}(S_1, \text{JOIN}(S_2, S_3) : \mathbf{D_6}) : \mathbf{D_7}$ (6.1)

```
{{
        1. D6.attributes = union (D2.attributes, D3.attributes) ;
}}
2. is_associative (D5.join_predicate, D2.attributes)
{{
        3. D7 = D5 ;
        4. D7.join_predicate = D4.join_predicate ;
        5. D6.join_predicate = D5.join_predicate ;
        6. D6.record_width = D2.record_width + D3.record_width ;
        7. D6.cardinality = compute_cardinality (D2, D3) ;
}}
```

(a) Join associativity T-rule

```
(JOIN ?op_arg5 ((JOIN ?op_arg4 (?1 ?2)) ?3)) → (JOIN ?op_arg7 (?1 (JOIN ?op_arg6 (?2 ?3))))     (6.2)
%cond_code
{{
        1. NO_OP ;
        2. if (!(is_associative (?op_arg5 → op_arg.join_predicate, ?2 → expr.argument.log_prop → attributes))
              &&(?op_arg5 → op_arg.layer <= 1) )
              REJECT ;
}}
%appl_code
{{
        3. ?op_arg7 → op_arg.join_predicate = ?op_arg5 → op_arg.join_predicate ;
        3. ?op_arg7 → log_prop = ?op_arg5 → log_prop ;
        3. ?op_arg7 → op_arg.layer = ?op_arg5 → op_arg.layer ;
        4. ?op_arg7 → op_arg.join_predicate = ?op_arg4 → op_arg.join_predicate ;
        5. ?op_arg6 → op_arg.join_predicate = ?op_arg5 → op_arg.join_predicate ;
        6. NO_OP ;
        7. NO_OP ;
        8. ?op_arg6 → op_arg.layer = 1 ;
}}
```

(b) P2V-generated trans_rule corresponding to the T-rule in (a). Statement numbers correspond to the same numbered statements in (a). Boxed clauses and statements are specific to layered optimizers (cf. Figure 4.6(b)).

Figure 6.9: Translating the join associativity T-rule in the SEQUENTIAL layer (from the centralized optimizer in Figure 6.8)

tensibility and reusability.

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 the P2V preprocessor as follows. The preprocessor automatically introduces a new property, called layer, that records the layer number in which each expression was generated.[6] Initially, all *new* expressions have a layer property of 1 (meaning that all rules in layers 1 and higher can be applied to them). As expressions propagate through the layers, their layer property values are assigned by the P2V preprocessor in rule actions. In other words, the P2V preprocessor augments rule actions with statements that set the layer property value for each new expression. The test of a rule is also modified to ensure that an expression generated in a lower layer is not transformed in a layer above. This is best illustrated by an example.

Consider the join associativity T-rule in the SEQUENTIAL layer shown in Figure 6.9(a). When a layered optimizer consisting of the SEQUENTIAL layer (e.g., the centralized optimizer in Figure 6.4) is compacted, the actions of the join associativity rule are modified to assign the layer property values of any new expressions. This is shown in Figure 6.9(b). The boxed statements and clause (in the test) are introduced by the P2V preprocessor in addition to the other transformations described in Chapter 4. Notice that the root of a new expression obtained by a rule has the same layer number as the root of the old expression; this is because they belong to the same equivalence class. New sub-expressions, however, have their layer values set to 1; thus, all rules, beginning from the first layer, are applicable to them.

The translation of I-rules works similarly, i.e., the test of an I-rule is augmented to check if the operator being rewritten was generated by a higher or lower layer.

6.5 Benchmarking Layered Optimizers

In the previous section, we described how layered optimizer specifications are compacted by the P2V preprocessor to generate monolithic rule sets. The question that naturally arises is whether this method of optimizer construction (layered specification followed by compaction) results in a loss of efficiency of the generated optimizer. In this section, we present

[6] Layers are numbered in ascending order, beginning from 1 for the topmost layer.

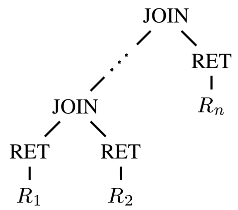
preliminary experimental results that demonstrate that the efficiency of an optimizer is not sacrificed by using a layered specification.

Consider the layered centralized optimizer shown in Figure 6.4. As shown in Figure 6.8, this can be compacted to a monolithic rule set. However, rules in the monolithic set do not have exactly the same rule actions as a non-layered Prairie specification (since the P2V preprocessor adds statements to track layer numbers), or a hand-written Volcano specification (since the abstractions in Prairie and Volcano are different). To verify that the efficiency of optimizers generated from layered specifications is not sacrificed, we conducted some 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.

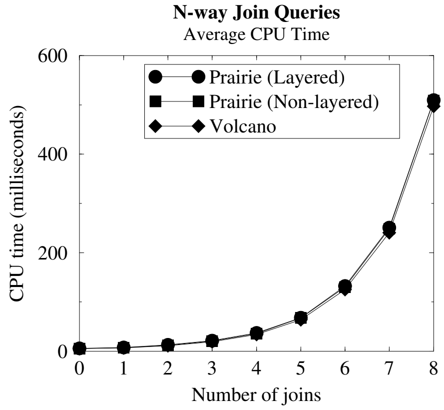
The benchmarking of the layered optimizers consisted of optimizing left-deep N-way join operator trees, for varying values of N, as shown in Figure 6.10(a). Random queries were generated using a uniform random number generator, as in Section 5.2. The set of relations, along with their attributes, cardinalities, and record widths was fixed. The order of relations in the left-deep tree was varied. Each join had an 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 optimizers, we simulated a database with four sites; each relation was randomly assigned to a site. For the replicated DBMS case, there were two replicas for each stored file referenced in an operator tree; to make the replication meaningful, each replica of a given stored file 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 opti-

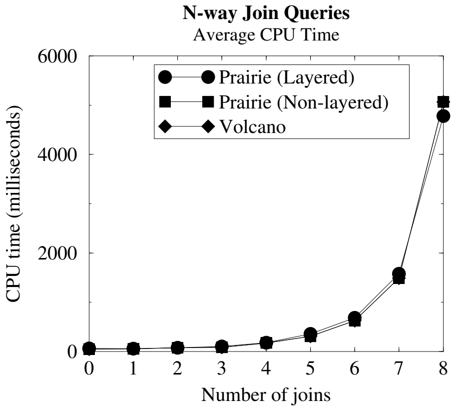
[7] We chose 10 instead of a larger number since the average run time was not substantially different for more queries.



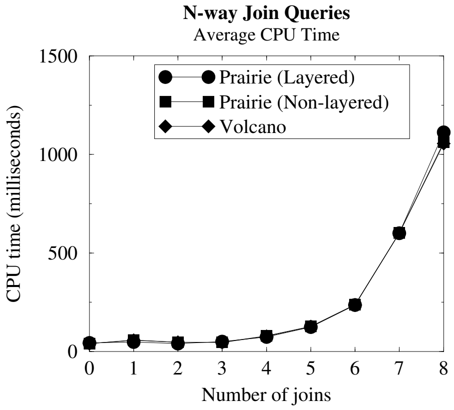
(a) N-way join benchmark query



(b) Query optimization time for the centralized DBMS optimizer (Figure 6.4)



(c) Query optimization time for the distributed DBMS optimizer (Figure 6.6)



(d) Query optimization time for the replicated DBMS optimizer (Figure 6.7)

Figure 6.10: Benchmarking layered optimizers

mizing 10 different queries. As before, 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 6.10(b)[8], 6.10(c), and 6.10(d) for the centralized

[8]The optimization times for both non-layered Prairie and Volcano for the centralized optimizer are the same

(Figure 6.4), distributed (Figure 6.6), and replicated (Figure 6.7) DBMS optimizers, respectively. In each case, we can see that the three specifications are virtually equally efficient, confirming the hypothesis that layered optimizers need not sacrifice any performance. More experiments are needed, however, to verify that the performance is good when scaled to *large* layered optimizers.

Another metric which can measure the usefulness of layered optimizer specifications is the ease with which they can be tailored to different applications. In this case, the layered approach is useful 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 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 layered configurations. It is in this respect that we believe that the layered approach will yield the most productivity gains.

6.6 Related Work

Optimizer design and implementation using pre-defined building-blocks has been proposed by other researchers. Most of these proposals follow a rule-based paradigm. In this section, we briefly describe some of these approaches. The main problem with these ideas is that they are paper proposals, so it is not immediately clear how well they might work in practice. Moreover, as we have seen in Chapter 4, it is important to have a compiler that can generate efficient, compact optimizers from a specification constructed using components. None of the proposals discussed in this section describe how that is done, or even if it is possible.

Sciore and Sieg [44] describe an optimizer generator model that allows a DBI to construct a rule-based optimizer using *modules*. Each module consists of *term rewrite rules*, where rewrite rules transform terms in a relational algebra. These rules can have *conditions* associated with them. Each module has exported and imported interfaces which consist of terms.

as those shown in Figure 5.1(b).

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 is not 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 unwieldy to use. If so, then it defeats the purpose of an extensible optimizer generator paradigm.

Mitchell, Dayal, and Zdonik [39, 40] 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.

6.7 Summary

In this chapter, we described how layered optimizers can be constructed using Prairie, and how the P2V preprocessor can generate monolithic optimizers from a layered specification. We also presented preliminary experimental evidence from simple layered optimizers that demonstrate that optimizer performance is not sacrificed by using reusable, extensible layers. More experiments are necessary to conclusively validate that a layered optimizer can be just as efficient as (and more extensible than) monolithic optimizers. Our primary goal in this chapter 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 [10, 11].

Chapter 7

Conclusion

This dissertation described Prairie, an algebraic framework for rule-specification in query optimizers. This chapter summarizes the contributions of our research, and suggests avenues of future work.

7.1 Contributions of Dissertation

To design open DBMSs that are easily tailorable and retargetable to different architectures and applications, the major components of a DBMS should be specified using abstractions that are independent of the underlying implementation details. To that end, we have proposed, designed, and implemented Prairie, an algebraic specification framework for constructing rule-based query optimizers. In Chapter 1, we enumerated four critical requirements in such a framework. Below, we summarize and describe how Prairie meets all of these intended goals.

- **Abstractions.** Well-defined and high-level abstractions are critical to the success of an open system. Prairie provides abstractions that are central to the specification of any optimizer; these abstractions capture the operation of an optimizer without regard to their implementation. Operators, algorithms, rules, descriptors, and layers represent the various abstractions that are key to Prairie's success. Operators represent the abstract computations on streams. Algorithms describe the implementations of operators. Rules (T-rules and I-rules) specify the transformations that an optimizer applies to generate its search space; the internal representation of the search space is left as an implementation-level detail. Descriptors are used to encapsulate the properties of expressions in the search space. Unlike other rule-based optimizers which use multiple structures to specify properties (based on their use), descriptors represent all the properties used in an optimizer; the classification of properties based on their usage is an internal detail, better left to an implementation. Layers encapsulate larger rule sets; each layer typically represents a rule specification in its own right.

- **Extensibility.** Extensibility refers to the ease with which an existing system can be changed. The language constructs that Prairie provides to specify the abstractions mentioned above are designed to enable a DBI to quickly and efficiently add or delete operators, algorithms, rules, properties, or layers to construct an optimizer for a new query environment. This is an improvement over existing rule-based optimizers because any changes made by the DBI are seamless. Thus, modifying an optimizer using Prairie need not result in a system with lots of patches applied haphazardly; brittleness of the resulting optimizer is less of a concern.

- **Performance.** Performance of an optimizer is critical since the search spaces are typically exponential in the size of an operator tree. The experiences of many researchers have shown that a very general framework of system construction can result in inefficient implementations either because the abstractions do not adequately capture the system semantics, or because the translation of the specification into an executable system does not exploit the properties of the target application or architecture. In our research, we have used Volcano as the target framework for translating Prairie specifications. We have implemented the P2V preprocessor that transforms the abstractions embodied in Prairie to the lower-level details expected by the Volcano rule engine. We took care to ensure that the translation process generated efficient internal representations for the various abstractions. To validate the performance of this approach, we presented various benchmark experiments, both with small and large optimizer specifications using Prairie. The results confirm our belief that the preprocessor approach is a valid step, and that Prairie abstractions have efficient implementations.

- **Reconfigurability.** Layers in Prairie encapsulate a set of rules; they can be used as subsystems to construct a larger system (optimizer). We have extended Prairie to allow DBIs to specify layers, and to build optimizers as a linear composition of layers. The P2V preprocessor uses the composition semantics to generate an efficient monolithic optimizer from a composition of layers. Again, this process results in little or no loss in performance of the resulting optimizer as our preliminary experimental evidence shows.

7.2 Future Work

Prairie provides an open environment for the high-level specification of query optimizers. Together with the P2V preprocessor to generate executable optimizers, this approach results in extensible specifications. Future work will make this paradigm even more useful. In this section, we describe avenues for further research.

- Continuing validation of Prairie is important to ensure that the abstractions embodied within it are well-defined and represent important concepts of optimization. In this dissertation, we have presented experimental evidence using simple optimizers and the Open OODB optimizer. In the future, interesting experiments will involve distributed and parallel databases, multi-query optimization, and real-time queries. Each application introduces a set of constraints concerning the evaluation of queries and experimentation is needed to verify that Prairie specifications can adequately capture the semantics of the application. An example of such an application is extending the Open OODB optimizer (described in Chapter 5) to build distributed or parallel object-oriented optimizers.

- Layered Prairie specifications present several opportunities for future work. Currently, Prairie allows DBIs to encapsulate a set of rules in a layer. Encapsulation of other abstractions in Prairie is highly desirable. For example, a layer can define new properties in a descriptor, or new helper functions. The site information of streams in an optimizer for a distributed DBMS, for instance, can be defined as a new property in the DISTRIBUTION layer (see Chapter 6). The P2V preprocessor can be modified to compact layers and generate a monolithic rule set, descriptor, and a single set of helper functions.

- In this dissertation, we presented preliminary experimental evidence to validate the performance of layered optimizers. Future work will include more experiments with large layered optimizer specifications. For instance, the Open OODB optimizer is a good candidate for such a specification. It will also be instructive to extend the layered Open OODB specification to distributed or replicated databases.

- Extending the Prairie specification language to allow hierarchical, non-linear compositions (as opposed to linear compositions) will allow a DBI to specify more optimizers than are currently possible. This means that layers can have multiple parameters, each of which can be instantiated by a layer. For example, a retrieval layer implementing an abstract RET operator with the index scan algorithm may require an additional parameter to define the implementation of the index. Experience with Genesis [4] has shown that such parameterized layers can capture the semantics of many widely used optimization algorithms. Extending Prairie to allow such compositions will necessitate a modification of P2V's layer compaction algorithm.

- Extensible search strategies is another important area of future work. Currently, the P2V preprocessor translates Prairie specifications into Volcano specifications to be compiled with Volcano's rule engine. However, Volcano's rule engine implements a hard-wired search strategy that is very rigid. Moreover, the implementation itself does not lend itself to easy change; modifying it will (we hypothesize) require major changes. Since Prairie provides abstractions that are high-level and (mostly) independent of the underlying search strategy, we believe that Prairie specifications can be easily translated to another rule engine with an appropriate preprocessor. Moreover, since Prairie allows encapsulation of rule sets in layers, we can foresee each layer encapsulating a rule engine. Thus, the need for a global search strategy disappears, and each layer implements its own search strategy (in addition to its own search space and cost model). This is similar to the frameworks proposed in [39, 40, 44]. One interesting problem is the compaction of layers — namely, how can we generate a monolithic rule set with a search strategy that is semantically equivalent to the composition of the search strategies in the different layers? Since each layer can have a different rule engine, any compaction of layers should generate a monolithic rule engine that is provably equivalent to the actions of the hierarchical ordering of the individual rule engines.

- In Chapter 6, we mentioned that layers in Prairie are mostly symmetric. This affords a much greater flexibility in layer composition. However, not all legal compositions are necessarily valid or meaningful. This implies a need for some validation techniques that can detect invalid compositions. Batory and Geraci [8] describe a useful method for design rule checking in layered systems that validates layer compositions. A future goal is to integrate this validation into the P2V preprocessor.

- Multi-phase query optimizers are being investigated as a means of limiting the enormous size of the search spaces in most optimizers. In this approach, the optimizer is implemented as a composition of separate phases, each phase optimizing the output of another. Systems like XPRS [48] and Mariposa [47] are some examples of this approach. The Mariposa distributed DBMS, for instance, implements a three-phase optimizer; these phases are compilation, parallelization, and distribution. The compilation phase optimizes an operator tree assuming a centralized DBMS, the parallelization phase introduces intra-operator parallelism, and the distribution phase selects sites for the execution of the various algorithms. As Stonebraker et al point out in [47], partitioning the optimization process into multiple phases can result in sub-optimal plans. However, careful design and implementation of each phase can significantly reduce the search space of the optimizer (and, consequently, the optimization time), while still generating a good access plan. The design of such multi-phase optimizers can be greatly facilitated by ensuring that each phase is reconfigurable; this will enable DBIs to fine-tune each phase to generate plans that are most likely to yield the best access plan for queries in a particular application. Prairie may be a valuable tool in building multi-phase optimizers, where each phase is layered and specified using Prairie.

- Since Prairie depends on Volcano's rule engine, and since Volcano implements a top-down search strategy, a future area of research would involve investigating the use of Prairie for specifying bottom-up optimizers. Since most of Prairie's abstractions do not depend on the underlying search strategy, this implies that Prairie might serve as a useful framework for both top-down and bottom-up optimizers.

7.3 Retrospective

Our research concentrated on developing a high-level framework for rule specification in query optimizers, and a preprocessor for the generation of efficient optimizers. We have used Volcano as the backend rule engine for our preprocessor and for our experiments.

Most of the abstractions in Prairie capture optimizer actions succintly. The descriptor abstraction, for instance, not only allows a DBI to concisely represent all properties, but also for the same expression to have different descriptors (i.e., different properties) depending on which side of a rule it occurs on. Abstract operators and algorithms that are all treated as first-class objects allows Prairie specifications to scale gracefully. We feel, thus, that Prairie is a good beginning for constructing extensible, scalable, and efficient optimizers.

Although Volcano's rule engine uses an efficient, exhaustive search strategy, its implementation leaves it extremely inextensible. Moreover, some of the details of the rule engine propagate up to the Volcano rule specification language. For example, the data structures used to store operator trees and their properties are optimized for maximum storage efficiency. Unfortunately, this also means that the DBI has to be careful and knowledgable enough to partition properties appropriately. Another example of this inflexibility is the multitude of support functions with specific parameter lists that is required by Volcano. This not only disperses the optimizer actions over a wider area, but also constrains the DBI since the parameter lists of the support functions restrict the properties accessible in the function. Since Prairie is designed to be independent of many of these implementation details, it is the P2V preprocessor that currently has to bear the brunt of the burden of generating many of the low-level "abstractions" of Volcano. A better approach to our research would probably have been to implement our own rule engine (instead of using Volcano) with the same four goals (abstractions, extensibility, performance, and reconfigurability) as the rule specification language.

Cost-based query optimization is a well-studied area of research. Even though the techniques are well understood, building an optimizer is still an expensive proposition. This problem will only be magnified as DBMSs are called upon to support larger amounts of data as well as unstructured, novel, or heterogeneous data. What is needed are tools that can help in a streamlining the process of optimizer development; these tools have to be general and contain abstractions that are applicable to a wide variety of applications. This dissertation is a step in that direction.

Appendix A

Complexity of the System R Optimizer

In this appendix, we sketch a proof for the asymptotic complexity of the System R optimizer [46] for joining n relations. This complexity holds when the query graph is fully connected or the heuristic of delaying cross-products is dropped. This is the worst-case scenario.

The System R optimizer only generates left-deep operator trees. That is, the inner input of a join operator is the retrieval of a stored file. It is easy to see that, under this assumption, the total number of distinct expressions is $n!$. However, System R uses dynamic programming [19] to substantially reduce this complexity.

Below, we derive the complexity of the System R optimizer using dynamic programming. The optimizer constructs optimal plans for each possible join of j relations, for $j = 0, \ldots, n$. At each stage, only the best plan for joining a given subset of relations is maintained. Thus, the complexity of the System R optimizer is determined by the total number of operator trees that need to be examined at each stage.

The number of operator trees with 1 relation is $n = \binom{n}{0}$. The number of operator trees with 1 relation after pruning is $n = \binom{n}{1}$.

The number of operator trees with 2 relations is $\binom{n}{1}(n-1)$; $\binom{n}{1}$ is the number of 1-relation operator trees, and $(n-1)$ is the number of 2-relation operator trees generated from each 1-relation tree. The number of operator trees with 2 relations after pruning is $\binom{n}{2}$.

In general, the number of operator trees with j relations is $\binom{n}{j-1}(n-j+1)$. The number of operator trees with j relations after pruning is $\binom{n}{j}$.

The complexity of the System R optimization algorithm is, thus, given by the sum

$$\sum_{j=0}^{n-1} \binom{n}{j}(n-j) = n2^{n-1}$$

In other words, the System R optimizer has a worst-case complexity that is exponential in the number of relations to be joined.

Appendix B

Benefits of Rule Compaction

In this appendix, we present an informal argument that the run-time of an optimizer decreases when rules are compacted using the techniques described in Chapter 4.

Assume that O original rules are compacted to C rules ($C < O$). If the compacted rule set (with C rules) runs in time T, then the original rule set (with O rules) runs in $\frac{O}{C}T$ time. To see this, if a compacted rule c is obtained by compacting k original rules, then one application of the compacted rule c requires the equivalent of the k original rule applications. That is, a linear speedup is achieved by compacting rules.

Note, however, that T (the time it takes to optimize a query) is actually exponential in n (the number of relations to be joined), as proved in Appendix A. Thus, the benefits of rule compaction also extend to reducing memory requirements: each time a rule is applied, a new operator tree is introduced. By compacting rules, intermediate trees are not generated. There might be a very large (e.g., exponential) number of such intermediate trees whose creation are eliminated through rule compaction.

Appendix C

The Open OODB Rule Set

In this appendix, we list the rules that appear in the Open OODB optimizer. Figure C.1 shows the 17 transformation rules and 9 implementation rules using the Volcano optimizer-generator. Figure C.2 shows the 22 transformation rules and 11 implementation rules in the Prairie framework. Note that the first five Prairie T-rules (Figure C.2) have no counterparts on the Volcano side; these additional T-rules are required to introduce the enforcer-operator PR_ASSEMBLY into the various operator trees. Similarly, the last two Prairie I-rules (Figure C.2) are required to introduce the Null algorithm and the enforcer-algorithm Enf_assembly. The remaining Prairie rules have a one-to-one correspondence with the Volcano rules.

Most of the rules in the Open OODB optimizer are quite complex with substantial portions of code for manipulating properties. We compare two simple transformation rules in Figure C.3, which shows the corresponding rules for delaying the materialization operation. Figure C.3(a) shows the Volcano rule, and Figure C.3(b) shows the Prairie rule for this transformation. Note that the Prairie T-rule has more statements that compute properties of new expressions. This is because some properties in Volcano are computed automatically (by DBI-written support functions) when rules are applied, as mentioned in Chapter 2. Prairie requires a DBI to specify all property transformations explicitly.

Transformation rules

```
(OP_SELECT ?op_arg1 (?1)) −→ (OP_SELECT ?op_arg2 ((OP_SELECT ?op_arg3 (?1))))
(OP_MAT ?op_arg1 ((OP_SELECT ?op_arg2 (?1)))) −→!(OP_SELECT ?op_arg3 ((OP_MAT op_arg4 (?1))))
(OP_SELECT ?op_arg1 ((OP_MAT ?op_arg2 (?1)))) −→!(OP_MAT ?op_arg3 ((OP_SELECT op_arg4 (?1))))
(OP_SELECT ?op_arg1 ((OP_UNNEST ?op_arg2 (?1)))) −→ (OP_UNNEST ?op_arg3 ((OP_SELECT op_arg4 (?1))))
(OP_MAT ?op_arg1 (?1)) −→ (OP_JOIN ?op_arg2 (?1 (OP_GET ?op_arg3 ())))
(OP_SELECT ?op_arg1 ((OP_JOIN ?op_arg2 (?1 ?2)))) −→ (OP_JOIN ?op_arg3 (?1 (OP_SELECT op_arg4 (?2))))
(OP_SELECT ?op_arg1 ((OP_JOIN ?op_arg2 (?1 ?2)))) −→ (OP_JOIN ?op_arg3 ((OP_SELECT op_arg4 (?1)) ?2))
(OP_JOIN ?op_arg1 (?1 (OP_MAT ?op_arg2 (?2)))) −→ (OP_MAT ?op_arg3 ((OP_JOIN op_arg4 (?1 ?2))))
(OP_JOIN ?op_arg1 ((OP_SELECT ?op_arg2 (?1)) ?2)) −→!(OP_SELECT ?op_arg3 ((OP_JOIN op_arg4 (?1 ?2))))
(OP_MAT ?op_arg1 ((OP_JOIN ?op_arg2 (?1 ?2)))) −→ (OP_JOIN ?op_arg3 ((OP_MAT op_arg4 (?1)) ?2))
(OP_JOIN ?op_arg1 (?1 ?2)) −→!(OP_JOIN ?op_arg2 (?2 ?1))
(OP_SELECT ?op_arg1 ((OP_SELECT ?op_arg2 (?1)))) −→!(OP_SELECT ?op_arg3 ((OP_SELECT op_arg4 (?1))))
(OP_MAT ?op_arg1 ((OP_MAT ?op_arg2 (?1)))) −→!(OP_MAT ?op_arg3 ((OP_MAT op_arg4 (?1))))
(OP_SELECT ?op_arg1 ((OP_SELECT ?op_arg2 (?1)))) −→!(OP_SELECT ?op_arg3 (?1))
(OP_JOIN ?op_arg1 ((OP_JOIN ?op_arg2 (?1 ?2)) ?3)) −→ (OP_JOIN ?op_arg3 (?1 (OP_JOIN op_arg4 (?2 ?3))))
(OP_MAT ?op_arg1 ((OP_MAT ?op_arg2 (?1)))) −→ (OP_MAT ?op_arg3 (?1))
(OP_JOIN ?op_arg1 (?1 ?2)) −→!(OP_JOIN ?op_arg2 (?2 ?1))
```

Implementation rules

```
(OP_JOIN ?op_arg1 (?1 ?2)) −→ (Al_hh_join ?al_arg1 (?1 ?2))
(OP_JOIN ?op_arg1 (?1 (OP_GET ?op_arg2 ()))) −→ (Al_ptr_hh_join ?al_arg1 (?1))
(OP_SELECT ?op_arg1 (?1)) −→ (Al_filter ?al_arg1 (?1))
(OP_MAT ?op_arg1 (?1)) −→ (Al_assembly ?al_arg1 (?1))
(OP_GET ?op_arg1 ()) −→ (Al_file_scan ?al_arg1 ())
(OP_UNNEST ?op_arg1 (?1)) −→ (Al_unnest ?al_arg1 (?1))
(OP_PROJECT ?op_arg1 (?1)) −→ (Al_project ?al_arg1 (?1))
(OP_SELECT ?op_arg1 ((OP_MAT ?op_arg2 ((OP_GET ?op_arg3 ()))))) −→ (Al_index_scan ?al_arg1 ())
(OP_SELECT ?op_arg1 ((OP_GET ?op_arg2 ()))) −→ (Al_index_scan ?al_arg1 ())
```

Figure C.1: Volcano rules for the Open OODB optimizer

Transformation rules

OP_SELECT(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ PR_SELECT(PR_ASSEMBLY(S_1) : $\mathbf{D_3}$) : $\mathbf{D_4}$
OP_PROJECT(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ PR_PROJECT(PR_ASSEMBLY(S_1) : $\mathbf{D_3}$) : $\mathbf{D_4}$
OP_JOIN(S_1, S_2) : $\mathbf{D_3}$ $\Longrightarrow$ PR_JOIN(PR_ASSEMBLY(S_1) : $\mathbf{D_4}$, PR_ASSEMBLY(S_2) : $\mathbf{D_5}$) : $\mathbf{D_6}$
OP_MAT(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ PR_MAT(PR_ASSEMBLY(S_1) : $\mathbf{D_3}$) : $\mathbf{D_4}$
OP_UNNEST(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ PR_UNNEST(PR_ASSEMBLY(S_1) : $\mathbf{D_3}$) : $\mathbf{D_4}$
OP_SELECT(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ OP_SELECT(OP_SELECT(S_1) : $\mathbf{D_3}$) : $\mathbf{D_4}$
OP_MAT(OP_SELECT(S_1) : $\mathbf{D_2}$) : $\mathbf{D_3}$ $\Longrightarrow$ OP_SELECT(OP_MAT(S_1) : $\mathbf{D_4}$) : $\mathbf{D_5}$
OP_SELECT(OP_MAT(S_1) : $\mathbf{D_2}$) : $\mathbf{D_3}$ $\Longrightarrow$! OP_MAT(OP_SELECT(S_1) : $\mathbf{D_4}$) : $\mathbf{D_5}$
OP_SELECT(OP_UNNEST(S_1) : $\mathbf{D_2}$) : $\mathbf{D_3}$ $\Longrightarrow$ OP_UNNEST(OP_SELECT(S_1) : $\mathbf{D_4}$) : $\mathbf{D_5}$
OP_MAT(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ OP_JOIN(S_1, OP_GET(F_3) : $\mathbf{D_4}$) : $\mathbf{D_5}$
OP_SELECT(OP_JOIN(S_1, S_2) : $\mathbf{D_3}$) : $\mathbf{D_4}$ $\Longrightarrow$ OP_JOIN(S_1, OP_SELECT(S_2) : $\mathbf{D_5}$) : $\mathbf{D_6}$
OP_SELECT(OP_JOIN(S_1, S_2) : $\mathbf{D_3}$) : $\mathbf{D_4}$ $\Longrightarrow$ OP_JOIN(OP_SELECT(S_1) : $\mathbf{D_5}$, S_2) : $\mathbf{D_6}$
OP_JOIN(S_1, OP_MAT(S_2) : $\mathbf{D_3}$) : $\mathbf{D_4}$ $\Longrightarrow$ OP_MAT(OP_JOIN(S_1, S_2) : $\mathbf{D_5}$) : $\mathbf{D_6}$
OP_JOIN(S_1, OP_SELECT(S_2) : $\mathbf{D_3}$) : $\mathbf{D_4}$ $\Longrightarrow$! OP_SELECT(OP_JOIN(S_1, S_2) : $\mathbf{D_5}$) : $\mathbf{D_6}$
OP_MAT(OP_JOIN(S_1, S_2) : $\mathbf{D_3}$) : $\mathbf{D_4}$ $\Longrightarrow$ OP_JOIN(OP_MAT(S_1) : $\mathbf{D_5}$, S_2) : $\mathbf{D_6}$
OP_JOIN(S_1, S_2) : $\mathbf{D_3}$ $\Longrightarrow$! OP_JOIN(S_2, S_1) : $\mathbf{D_4}$
OP_SELECT(OP_SELECT(S_1) : $\mathbf{D_2}$) : $\mathbf{D_3}$ $\Longrightarrow$! OP_SELECT(OP_SELECT(S_1) : $\mathbf{D_4}$) : $\mathbf{D_5}$
OP_MAT(OP_MAT(S_1) : $\mathbf{D_2}$) : $\mathbf{D_3}$ $\Longrightarrow$! OP_MAT(OP_MAT(S_1) : $\mathbf{D_4}$) : $\mathbf{D_5}$
OP_SELECT(OP_SELECT(S_1) : $\mathbf{D_2}$) : $\mathbf{D_3}$ $\Longrightarrow$! OP_SELECT(S_1) : $\mathbf{D_4}$
OP_JOIN(OP_JOIN(S_1, S_2) : $\mathbf{D_4}$, S_3) : $\mathbf{D_5}$ $\Longrightarrow$ OP_JOIN(S_1, OP_JOIN(S_2, S_3) : $\mathbf{D_6}$) : $\mathbf{D_7}$
OP_MAT(OP_MAT(S_1) : $\mathbf{D_2}$) : $\mathbf{D_3}$ $\Longrightarrow$ OP_MAT(S_1) : $\mathbf{D_4}$
OP_JOIN(S_1, S_2) : $\mathbf{D_3}$ $\Longrightarrow$! OP_JOIN(S_2, S_1) : $\mathbf{D_4}$

Implementation rules

PR_JOIN(S_1, S_2) : $\mathbf{D_3}$ $\Longrightarrow$ Al_hh_join(S_1 : $\mathbf{D_4}$, S_2 : $\mathbf{D_5}$) : $\mathbf{D_6}$
PR_JOIN(S_1, OP_GET(F_2) : $\mathbf{D_3}$) : $\mathbf{D_4}$ $\Longrightarrow$ Al_ptr_hh_join(S_1 : $\mathbf{D_6}$) : $\mathbf{D_5}$
PR_SELECT(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ Al_filter(S_1 : $\mathbf{D_4}$) : $\mathbf{D_3}$
PR_MAT(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ Al_assembly(S_1 : $\mathbf{D_4}$) : $\mathbf{D_3}$
OP_GET(F_1) : $\mathbf{D_2}$ $\Longrightarrow$ Al_file_scan(F_1) : $\mathbf{D_3}$
PR_UNNEST(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ Al_unnest(S_1 : $\mathbf{D_4}$) : $\mathbf{D_3}$
PR_PROJECT(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ Al_project(S_1 : $\mathbf{D_4}$) : $\mathbf{D_3}$
PR_SELECT(PR_MAT(OP_GET(F_1) : $\mathbf{D_2}$) : $\mathbf{D_3}$) : $\mathbf{D_4}$ $\Longrightarrow$ Al_index_scan(F_1) : $\mathbf{D_5}$
PR_SELECT(OP_GET(F_1) : $\mathbf{D_2}$) : $\mathbf{D_3}$ $\Longrightarrow$ Al_index_scan(F_1) : $\mathbf{D_4}$
PR_ASSEMBLY(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ Null(S_1 : $\mathbf{D_3}$) : $\mathbf{D_4}$
PR_ASSEMBLY(S_1) : $\mathbf{D_2}$ $\Longrightarrow$ Enf_assembly(S_1 : $\mathbf{D_3}$) : $\mathbf{D_4}$

Figure C.2: Prairie rules for the Open OODB optimizer

(OP_JOIN ?op_arg1 (?1 (OP_MAT ?op_arg2 (?2)))) $\longrightarrow$ (OP_MAT ?op_arg3 ((OP_JOIN op_arg4 (?1 ?2)))) (C.1)
%cond_code
{{
1. if (dependent_mat_and_pred(?op_arg2, ?op_arg1, PRED_DEPENDS_ON_MAT) == TRUE)
REJECT ;
}}
%appl_code
{{
2. copy_operator_arg(?op_arg4, ?op_arg1) ;
3. copy_operator_arg(?op_arg3, ?op_arg2) ;
}}

(a) Volcano rule

OP_JOIN(S_1, OP_MAT(S_2) : $\mathbf{D_3}$) : $\mathbf{D_4} \Longrightarrow$ OP_MAT(OP_JOIN(S_1, S_2) : $\mathbf{D_5}$) : $\mathbf{D_6}$ (C.2)
{{

}}
1. dependent_mat_and_pred(&($\mathbf{D_3}$.oa), &($\mathbf{D_4}$.oa), PRED_DEPENDS_ON_MAT)! = TRUE
{{
2. dcopy($\mathbf{D_5}$.oa, $\mathbf{D_4}$.oa) ;
3. dcopy($\mathbf{D_6}$.oa, $\mathbf{D_3}$.oa) ;
4. dcopy($\mathbf{D_6}$.lp, $\mathbf{D_4}$.lp);
5. init_descriptor_OP_JOIN(&($\mathbf{D_5}$), &($\mathbf{D_1}$), &($\mathbf{D_2}$)) ;
}}

(b) Prairie rule corresponding to the rule in (a). Statement numbers correspond to the same numbered statements in (a).

Figure C.3: Comparison of an Open OODB rule

Bibliography

[1] M. M. Astrahan et al. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.

[2] Don Batory. Modeling the storage architectures of commercial database systems. *ACM Transactions on Database Systems*, 10(4):463–528, December 1985.

[3] Don Batory. Extensible cost models and query optimization in GENESIS. *Database Engineering*, 9(4):30–36, December 1986.

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

[5] Don Batory. On the reusability of query optimization algorithms. *Information Systems*, 49(1/3), 1989.

[6] Don Batory. The Genesis database system compiler: User manual. Technical Report TR 90–27, The University of Texas at Austin, August 1990.

[7] Don Batory, Lou Coglianese, Mark Goodwin, and Steve Shafer. Creating reference architectures: An example from avionics. In *Proceedings Symposium on Software Reusability*, Seattle, April 1995.

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

[9] 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.

[10] 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.

[11] 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.

[12] Don Batory and Devang Vasavada. Software components for object-oriented database systems. *International Journal of Software Engineering and Knowledge Engineering*, 3(2):165–192, 1993.

[13] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking database systems: A systematic approach. In *Proceedings 9th International Conference on Very Large Data Bases*, pages 8–19, Florence, November 1983.

[14] Dina Bitton and Carolyn Turbyfill. A retrospective on the wisconsin benchmark. In Michael Stonebraker, editor, *Readings in Database Systems*. Morgan Kaufmann, San Mateo, second edition, 1994.

[15] 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.

[16] M. W. Blasgen and K. P. Eswaran. Storage and access in relational data bases. *IBM Systems Journal*, 16(4):363–377, 1977.

[17] Michael J. Carey, David J. DeWitt, Daniel Frank, Goetz Graefe, M. Muralikrishna, Joel E. Richardson, and Eugene J. Shekita. The architecture of the EXODUS exten-

sible DBMS. In *Proceedings International Workshop on Object-Oriented Database Systems*, pages 52–65, Asilomar, September 1986.

[18] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[19] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, 1990.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Leonidas Fegaras, David Maier, and Tim Sheard. Specifying rule-based query optimizers in a reflective framework. In *Proceedings Third International Conference on Deductive and Object-Oriented Databases*, pages 146–168, Phoenix, December 1993.

[25] 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.

[26] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Micro-Architectures for Reusable Object-Oriented Design*. Addison-Wesley, Reading, 1994.

[27] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriloa and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Company, 1993.

[28] Goetz Graefe. *Rule-Based Query Optimization in Extensible Database Systems*. PhD thesis, University of Wisconsin–Madison, 1987.

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

[30] 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.

[31] Goetz Graefe and William McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings 9th International Conference on Data Engineering*, pages 209–218, Vienna, April 1993.

[32] Jim Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufman, San Mateo, second edition, 1993.

[33] 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.

[34] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, June 1984.

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

[36] 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.

[37] Guy M. Lohman, C. Mohan, Laura M. Haas, Bruce G. Lindsay, Patricia G. Selinger, Paul F. Wilms, and Dean Daniels. Query processing in R^*. Research Report RJ 4272, IBM Research Laboratory, San Jose, April 1984.

[38] William McKenna. *Efficient Search in Extensible Database Query Optimization: The Volcano Optimizer Generator*. PhD thesis, University of Colorado at Boulder, 1993.

[39] 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.

[40] 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.

[41] Patrick O'Neil. *Database: Principles, Programming, and Performance*. Morgan Kaufmann, San Mateo, 1994.

[42] W. B. Rubenstein, M. S. Kubicar, and R. G. G. Cattell. Benchmarking simple database operations. In *Proceedings 1987 ACM SIGMOD International Conference on Management of Data*, pages 387–394, San Francisco, May 1987.

[43] P. Schwarz, W. Chang, J. C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the Starburst database system. In *Proceedings International Workshop on Object-Oriented Database Systems*, pages 85–92, Asilomar, September 1986.

[44] 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.

[45] P. G. Selinger and M. Adiba. Access path selection in distributed data base management systems. In Deen and Hammersly, editors, *Proceedings International Conference on Databases*, pages 204–215, University of Aberdeen, July 1980.

[46] 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.

[47] Michael Stonebraker, Paul M. Aoki, Robert Devine, Witold Litwin, and Michael Olson. Mariposa: A new architecture for distributed data. In *Proceedings 10th International Conference on Data Engineering*, pages 54–65, Houston, February 1994.

[48] Michael Stonebraker, Randy Katz, David Patterson, and John Ousterhout. The design of XPRS. In *Proceedings 14th International Conference on Very Large Data Bases*, pages 318–330, Los Angeles, 1988.

[49] 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.

[50] David L. Wells, José A. Blakeley, and Craig W. Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–82, October 1992.

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

Vita

Dinesh Das was born in Kharagpur, India on August 4, 1967, the son of Joshi Markandeya Das and Annapurna Das. After graduating from Central School in 1984, he entered the Indian Institute of Technology, Kharagpur, where he received the degree of Bachelor of Technology in Computer Science and Engineering with honors in June 1988. In September 1988, he entered the Graduate School of The University of Texas at Austin. He received the degree of Master of Science in Computer Sciences from The University of Texas at Austin in December 1990.

Permanent Address: P.O. Box 8734
Austin, TX 78713

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.